1 NAME

TXR – Programming Language (Version 285)

2 SYNOPSIS

`txr [ options ] [ script-file [ arguments ... ]]`

3 DESCRIPTION

TXR is a general-purpose, multi-paradigm programming language. It comprises two languages integrated into a single tool: a text scanning and extraction language referred to as the **TXR** Pattern Language (sometimes just "TXR"), and a general-purpose dialect of Lisp called **TXR Lisp**.

TXR can be used for everything from "one liner" data transformation tasks at the command line, to data scanning and extracting scripts, to full application development in a wide range of areas.

A script written in the **TXR** Pattern Language, also referred to in this document as a *query*, specifies a pattern which matches one or more sources of inputs, such as text files. Patterns can consist of large chunks of multiline free-form text, which is matched literally against material in the input sources. Free variables occurring in the pattern (denoted by the `@` symbol) are bound to the pieces of text occurring in the corresponding positions. Patterns can be arbitrarily complex, and can be broken down into named pattern functions, which may be mutually recursive.

In addition to embedded variables which implicitly match text, the **TXR** pattern language supports a number of directives, for matching text using regular expressions, for continuing a match in another file, for searching through a file for the place where an entire subquery matches, for collecting lists, and for combining subqueries using logical conjunction, disjunction and negation, and numerous others.

Patterns can contain actions which transform data and generate output. These actions can be embedded anywhere within the pattern-matching logic. A common structure for small **TXR** scripts is to perform a complete matching session at the top of the script, and then deal with processing and reporting at the bottom.

The **TXR Lisp** language can be used from within **TXR** scripts as an embedded language, or completely standalone. It supports functional, imperative and object-oriented programming, and provides numerous data types such as symbols, strings, vectors, hash tables with weak reference support, lazy lists, and arbitrary-precision ("bignum") integers. It has an expressive foreign function interface (FFI) for calling into libraries and other software components that support C-language-style calls.

TXR Lisp source files as well as individual functions can be optionally compiled for execution on a virtual machine that is built into **TXR**. Compiled files execute and load faster, and resist reverse-engineering. Standalone application delivery is possible.

TXR is free software offered under the two-clause BSD license which places almost no restrictions on redistribution, and allows every conceivable use, of the whole software or any constituent part, royalty-free, free of charge, and free of any restrictions.

4 ARGUMENTS AND OPTIONS

If **TXR** is given no arguments, it will enter into an interactive mode. See the INTERACTIVE LISTENER section for a description of this mode. When **TXR** enters interactive mode this way, it prints a one-line banner announcing the program name and version, and one line of help text instructing the user how to exit.

If **TXR** is invoked under the name `txrlisp`, it behaves as if the `--lisp` option had been specified

before any other option. Similarly, if **TXR** is invoked under the name `txrvm`, it behaves as if the `--compiled` option had been given.

Unless the `-c` or `-f` options are present, the first non-option argument is treated as a *`script-file`* which is executed. This is described after the following descriptions of all of the options. Any additional arguments have no fixed meaning; they are available to the **TXR** query or **TXR Lisp** application for specifying input files to be processed, or other meanings under the control of the application.

Options which don't take an argument may be combined together. The `-v` and `-q` options are mutually exclusive. Of these two, the one which occurs in the rightmost position in the argument list dominates. The `-c` and `-f` options are also mutually exclusive; if both are specified, it is a fatal error.

`-D`*`var=value`*
: Bind the variable *`var`* to the value *`value`* prior to processing the query. The name is in scope over the entire query, so that all occurrences of the variable are substituted and match the equivalent text. If the value contains commas, these are interpreted as separators, which give rise to a list value. For instance `-Dvar=a,b,c` binds `var` to the list of the strings `"a"`, `"b"` and `"c"`. (See the `@(collect)` directive.) List variables provide a multiple match. That is to say, if a list variable occurs in a query, a successful match occurs if any of its values matches the text. If more than one value matches the text, the first one is taken.

`-D`*`var`* Binds the variable *`var`* to an empty string value prior to processing the query.

`-q`
: Quiet operation during matching. Certain error messages are not reported on the standard error device (but if the situations occur, they still fail the query). This option does not suppress error generation during the parsing of the query, only during its execution.

`-i`
: If this option is present, then **TXR** will enter into an interactive interpretation mode after processing all options, and the input query if one is present. See the INTERACTIVE LISTENER section for a description of this mode.

`-d`

`--debugger`
: Invoke the interactive **TXR** debugger. See the DEBUGGER section. Implies `--backtrace`.

`--backtrace`
: Turns on the establishment of backtrace frames for function calls so that a backtrace can be produced when an unhandled exception occurs, and in other situations. Backtraces are helpful in identifying the causes of errors, but require extra stack space and slow down execution.

`-n`

`--noninteractive`
: This option affects behavior related to **TXR**'s `*stdin*` stream. It also has a another, unrelated effect, on the behavior of the interactive listener; see below.

  Normally, if this stream is connected to a terminal device, it is automatically marked as having the real-time property when **TXR** starts up (see the functions `stream-set-prop` and `real-time-stream-p`). The `-n` option suppresses this behavior; the `*stdin*` stream remains ordinary.

  The **TXR** pattern language reads standard input via a lazy list, created by applying the `lazy-`

`stream-cons` function to the `*stdin*` stream. If that stream is marked real-time, then the lazy list which is returned by that function has behaviors that are better suited for scanning interactive input. A more detailed explanation is given under the description of this function.

If the `-n` option is effect and **TXR** enters into the interactive listener, the listener operates in *plain mode*. The listener reads buffered lines from the operating system without any character-based editing features or history navigation. In plain mode, no prompts appear and no terminal control escape sequences are generated. The only output is the results of evaluation, related diagnostic messages, and any output generated by the evaluated expressions themselves.

`-v` Verbose operation. Detailed logging is enabled.

`-b` This option binds a Lisp global lexical variable (as if by the `defparml` function) to an object described by Lisp syntax. It requires an argument of the form *`sym=value`* where *`sym`* must be, syntactically, a token denoting a bindable symbol, and *`value`* is arbitrary **TXR Lisp** syntax. The *`sym`* syntax is converted to the symbol it denotes, which is bound as a global lexical variable, if it is not already a variable. The *`value`* syntax is parsed to the Lisp object it denotes. This object is not subject to evaluation; the object itself is stored into the variable binding denoted by *`sym`*. Note that if *`sym`* already exists as a global variable, then it is simply overwritten. If *`sym`* is marked special, then it stays special.

`-B` If the query is successful, print the variable bindings as a sequence of assignments in shell syntax that can be *eval*-ed by a POSIX shell. II the query fails, print the word "false". Evaluation of this word by the shell has the effect of producing an unsuccessful termination status from the shell's *eval* command.

`-l`

`--lisp-bindings`

This option implies `-B`. Print the variable bindings in Lisp syntax instead of shell syntax.

`-a` *`num`*

This option implies `-B`. The decimal integer argument *`num`* specifies the maximum number of array dimensions to use for list-valued variable bindings. The default is 1. Additional dimensions are expressed using numeric suffixes in the generated variable names. For instance, consider the three-dimensional list arising out of a triply nested collect: `((("a" "b") ("c" "d")) (("e" "f") ("g" "h")))`. Suppose this is bound to a variable *`V`*. With `-a 1`, this will be reported as:

```
V_0_0[0]="a"
V_0_1[0]="b"
V_1_0[0]="c"
V_1_1[0]="d"
V_0_0[1]="e"
V_0_1[1]="f"
V_1_0[1]="g"
V_1_1[1]="h"
```

With `-a 2`, it comes out as:

```
V_0[0][0]="a"
V_1[0][0]="b"
V_0[0][1]="c"
```

```
V_1[0][1]="d"
V_0[1][0]="e"
V_1[1][0]="f"
V_0[1][1]="g"
V_1[1][1]="h"
```

The leftmost bracketed index is the most major index. That is to say, the dimension order is: `NAME_m_m+1_..._n[1][2]...[m-1]`.

`-c` *`query`*

Specifies the query in the form of a command-line argument. If this option is used, the *`script-file`* argument is omitted. The first non-option argument, if there is one, now specifies the first input source rather than a query. Unlike queries read from a file, (nonempty) queries specified as arguments using -c do not have to properly end in a newline. Internally, **TXR** adds the missing newline before parsing the query. Thus `-c "@a"` is a valid query which matches a line.

Example:

Shell script which uses **TXR** to read two lines `"1"` and `"2"` from standard input, binding them to variables `a` and `b`. Standard input is specified as `-` and the data comes from shell "here document" redirection:

code:
```
#!/bin/sh

txr -B -c "@a
@b" - <<!
1
2
!
```

output:
```
a=1
b=2
```

The `@;` comment syntax can be used for better formatting:

```
txr -B -c "@;
@a
@b"
```

`-f` *`script-file`*

Provides a way to specify the file from which the query is to be read, as an alternative to using the main *`script-file`* argument. This is useful in `#!` ("hash bang") scripts. (See Hash-Bang Support below.) Use of this option does not affect the order of processing. All of the options are processed first, before the *`script-file`* is read, as if it were specified by the main *`script-file`* argument. If the argument to `-f` is `-` (dash) then the script will be read from standard input instead of a file. If this option is used, the first non-option argument, if there is one, no longer specifies the *`script-file`*. It is an argument to the script, such as the name of an input source.

`-e` *`expression`*

Evaluates a **TXR Lisp** expression for its side effects, without printing its value. Can be specified more than once. The *`script-file`* argument becomes optional if at least one `-e`, `-p`, `-P` or `-t` option is processed. If the evaluation of every *`expression`* evaluated this way terminates normally, and there is no *`script-file`* argument, then **TXR** terminates with a successful status, instead of entering the interactive listener. The `-i` option can be used to request the listener.

`-p` *`expression`*
: Just like `-e` but prints the value of *`expression`* using the `prinl` function.

`-P` *`expression`*
: Like `-p` but prints using the `pprinl` function.

`-t` *`expression`*
: Like `-p` but prints using the `tprint` function.

`-C` *`number`*

`--compat=`*`number`*
: Requests **TXR** to behave in a manner that is compatible with the specified version of **TXR**. This makes a difference in situations when a release of **TXR** breaks backward compatibility. If some version N+1 deliberately introduces a change which is backward incompatible, then `-C N` can be used to request the old behavior.

  The requested value of N can be too low, in which case **TXR** will complain and exit with an unsuccessful termination status. This indicates that **TXR** refuses to be compatible with such an old version. Users requiring the behavior of that version will have to install an older version of **TXR** which supports that behavior, or even that exact version.

  If the option is specified more than once, the behavior is not specified.

  Compatibility can also be requested via the `TXR_COMPAT` environment variable instead of the `-C` option.

  For more information, see the COMPATIBILITY section.

`--gc-delta=`*`number`*
: The *`number`* argument to this option must be a decimal integer. It represents a megabyte value, the "GC delta": one megabyte is 1048576 bytes. The "GC delta" controls an aspect of the garbage collector behavior. See the `gc-set-delta` function for a description.

`--debug-autoload`
: This option turns on debugging, like `--debugger` but also requests stepping into the autoload processing of **TXR Lisp** library code. Normally, debugging through the evaluations triggered by autoloading is suppressed. Implies `--backtrace`.

`--debug-expansion`
: This option turns on debugging, like `--debugger` but also requests stepping into the parse-time macro-expansion of **TXR Lisp** code embedded in **TXR** queries. Normally, this is suppressed. Implies `--backtrace`.

`--help`
: Prints usage summary on standard output, and terminates successfully.

`--license`
: Prints the software license. This depends on the software being installed such that the LICENSE file is in the data directory. Use of **TXR** implies agreement with the liability disclaimer in the license.

`--version`
: Prints a message on standard output which includes the program version, and then immediately causes **TXR** to terminate with a successful status.

`--build-id`
: If **TXR** was built with an embedded build ID string, this option prints that string. Otherwise nothing is printed. In either case, **TXR** then immediately terminates with a successful status.

`--args`
: The `--args` option provides a way to encode multiple arguments as a single argument, which is useful on some systems which have limitations in their implementation of the hash-bang mechanism. For details about its special syntax, see Hash-Bang Support below. It is also useful in standalone application deployment. See the section STANDALONE APPLICATION SUPPORT, in which example uses of `--args` are shown.

`--eargs`
: The `--eargs` option (extended `--args`) is like `--args` but must be followed by an argument. The argument is removed from the argument list and substituted in place of occurrences of `{}` among the arguments expanded from the `--eargs` syntax.

`--lisp`

`--compiled`
: These options influence the treatment of query files which do not have a recognized suffix indicating their type. The `--lisp` option causes a file with an unrecognized suffix, or no suffix, to be treated as Lisp source; `--compiled` causes it to be treated as a compiled **TXR Lisp** file. Moreover, `--lisp` and `--compiled` influence the suffix search. By default, when a query file name does not have a recognizable suffix, and the file does not exist, **TXR** adds the `".txr"` suffix to the name and tries opening that name, and in a similar way tries `".tlo"`, `".tlo.gz"` and finally `".tl"`. In this situation, if either of these two options is specified, **TXR** tries only the `".tlo"`, `".tlo.gz"` and `".tl"` suffixes, in that order, avoiding the `".txr"` suffix. The search order is always `".tlo"` first, then `".tl"` regardless of whether `--lisp` or `--compiled` is specified.

  Note that `--lisp` and `--compiled` influence how the argument of the `-f` option is treated, but only if they precede that option.

  If the file has a recognized suffix: `".tl"`, `".tlo"`, `".tlo.gz"`, `".txr"` or `".txr_profile"`, then these options have no effect. The suffix determines the interpretation of the content. Moreover, no suffix search takes place: only the given path name is tried.

`--reexec`
: On platforms which support the POSIX `exec` family of functions, this option causes **TXR** to re-execute itself. The re-executed image receives the remaining arguments which follow the `--reexec` argument. Note: this option is useful for supporting setuid operation in hash-hang scripts. On some platforms, the interpreter designated by a hash-bang script runs without altered privilege, even if that interpreter is installed setuid. If the interpreter is executed directly, then setuid applies to it, but not if it is executed via hash bang. If the `--reexec` option is used in the interpreter command line of such a script, the interpreter will re-execute itself, thereby gaining the setuid privilege. The re-executed image will then obtain the script name from the arguments which are passed to it and determine whether that script will run setuid. See the section SETUID/SETGID OPERATION.

`--noprofile`
: If entering the interactive listener, suppress the reading of the `.txr_profile` in the home directory. See the Interactive Profile File subsection in the INTERACTIVE LISTENER section of the manual.

`--gc-debug`
: This option enables a behavior which stresses the garbage collector with frequent garbage collection requests. The purpose is to make it more likely to reproduce certain kinds of bugs. Use of this option severely degrades the performance of **TXR**.

`--vg-debug`
: If **TXR** is enabled with Valgrind support, then this option is available. It enables code which uses the Valgrind API to integrate with the Valgrind debugger, for more accurate tracking of garbage collected objects. For example, objects which have been reclaimed by the garbage collector are marked as inaccessible, and marked as uninitialized when they are allocated again.

`--free-all`
: This option specifies that all memory allocated by **TXR** should be freed upon normal termination. This behavior is useful for debugging memory leaks. An accurate leak detection tool, such as the one built into Valgrind, should report zero leaked or still reachable memory if `--free-all` has been used and **TXR** has terminated normally. that indicates either a leak in **TXR**, a leak or global object retention in a platform library, or else a a leak introduced due to misuse of FFI.

`--dv-regex`
: If this option is used, then regular expressions are all treated using the derivative-based back-end. The NFA-based regex implementation is disabled. Normally, only regular expressions which require the intersection and complement operators are handled using the derivative back-end. This option makes it possible to test that back-end on test cases that it wouldn't normally receive.

`--in-package=`*`name`*
: This option changes to the specified package, by finding the package of the specified *`name`* and assigning that to the `*package*` special variable. If the package is not found, a diagnostic is issued, and **TXR** terminates unsuccessfully. The package thus specified is visible to the subsequent occurrences of the `-e` family of options as well as of the `--compile` option. It does not affect the value of `*package*` which is in effect when a *`script-file`* is executed or when the interactive listener is entered.

`--compile=`*`source-file`*`[:`*`target-file`*`]`
: This option invokes the `compile-update-file` on *`source-file`*. If *`target-file`* is specified, it is passed to `compile-update-file` as the target argument; otherwise, that argument is defaulted. The option can be used multiple times to process multiple files. Unsuccessful compilation throws an exception, causing **TXR** to terminate abnormally. Similarly to the `-e` option, if this option is used at least once, and all of the invocations are successful, and there is no *`script-file`* argument, then **TXR** terminates with a successful status instead of entering the interactive listener. The `-i` option can be used request the listener.

`--`
: Signifies the end of the option list.

`-`
: This argument is not interpreted as an option, but treated as a filename argument. After the first such argument, no more options are recognized. Even if another argument looks like an option, it is treated as a name. This special argument - means "read from standard input" instead of a file. The *`script-file`*, or any of the data files, may be specified using this option. If two or more

files are specified as −, the behavior is system-dependent. It may be possible to indicate EOF from the interactive terminal, and then specify more input which is interpreted as the second file, and so forth.

After the options, the remaining arguments are treated as follows.

If neither the `-f` nor the `-c` options were specified, then the first argument is treated as the *`script-file`*. If no arguments are present, then **TXR** enters interactive mode, provided that none of the `-e`, `-p`, `-P` or `-t` options had been processed, in which case it instead terminates.

The **TXR** Pattern Language has features for implicitly treating the subsequent command-line arguments as input files. It follows the convention that an argument consisting of a single − (dash) character specifies that standard input is to be used, instead of opening a file. If the query does not use the `@(next)` directive to select an alternative data source, and a pattern-matching construct is processed which demands data, then the first argument will be opened as a data source. Arguments not opened as data sources can be assigned alternative meanings and uses, or can be ignored entirely, under control of the query.

Specifying standard input as a source with an explicit − argument is unnecessary. If no arguments are present, then **TXR** scans standard input by default. This was not true in versions of **TXR** prior to 171; see the COMPATIBILITY section.

TXR begins by reading the script, which is given as the contents of the argument of the `-c` option, or else as the contents of an input source specified by the `-f` option or by the *`script-file`* argument. If `-f` or the *`script-file`* argument specify − (dash) then the script is read from standard input.

In the case of the **TXR** pattern language, the entire query is scanned, internalized, and then begins executing, if it is free of syntax errors. (**TXR Lisp** is processed differently, form by form.) On the other hand, the pattern language reads data files in a lazy manner. A file isn't opened until the query demands material from that file, and then the contents are read on demand, not all at once.

The suffix of the *`script-file`* is significant. If the name has no suffix, or if it has a `".txr"` suffix, then it is assumed to be in the **TXR** pattern language. If it has the `".tl"` suffix, then it is assumed to be **TXR Lisp**. The `--lisp` and `--compiled` options change the treatment of unsuffixed script file names, causing them to be interpreted as **TXR Lisp** source or compiled **TXR Lisp**, respectively.

If a file name is specified which does not have a recognized suffix, and names a file which doesn't exist, then **TXR** adds the `".txr"` suffix and tries again. If that doesn't exist, another attempt is made with the `".tlo"` suffix, which will be treated as as a **TXR Lisp** compiled file. If that doesn't exist, then `".tlo.gz"` is tried, expected to be a file compressed in `gzip` format. Finally, if that doesn't exist, the `".tl"` suffix is tried, which will be treated as containing **TXR Lisp** source. If either the `--lisp` or `--compiled` option has been specified, then **TXR** skips trying the `".txr"` suffix, and tries only `".tlo"` followed by `".tlo.gz"` and `".tl"`.

A **TXR Lisp** file is processed as if by the `load` macro: forms from the file are read and evaluated. If the forms do not terminate the **TXR** process or throw an exception, and there are no syntax errors, then **TXR** terminates successfully after evaluating the last form. If syntax errors are encountered in a form, then **TXR** terminates unsuccessfully. **TXR Lisp** is documented in the section TXR LISP.

If a query file is specified, but no file arguments, it is up to the query to open a file, pipe or standard input via the `@(next)` directive prior to attempting to make a match. If a query attempts to match text, but has run out of files to process, the match fails.

5 STATUS AND ERROR REPORTING

TXR sends errors and verbose logs to the standard error device. The following paragraphs apply when **TXR** is run without enabling verbose mode with `-v`, or the printing of variable bindings with `-B` or `-a`.

If the command-line arguments are incorrect, **TXR** issues an error diagnostic and terminates with a failed status.

If the *`script-file`* specifies a query, and the query has a malformed syntax, **TXR** likewise issues error diagnostics and terminates with a failed status.

If the query fails due to a mismatch, **TXR** terminates with a failed status. No diagnostics are issued.

If the query is well-formed, and matches, then **TXR** issues no diagnostics, and terminates with a successful status.

In verbose mode (option `-v`), **TXR** issues diagnostics on the standard error device even in situations which are not erroneous.

In bindings-printing mode (options `-B` or `-a`), **TXR** prints the word `false` if the query fails, and exits with a failed termination status. If the query succeeds, the variable bindings, if any, are output on standard output.

If the *`script-file`* is **TXR Lisp**, then it is processed form by form. Each top-level Lisp form is evaluated after it is read. If any form is syntactically malformed, **TXR** issues diagnostics and terminates unsuccessfully. This is somewhat different from how the pattern language is treated: a script in the pattern language is parsed in its entirety before being executed.

6 BASIC TXR SYNTAX

6.1 Comments

A query may contain comments which are delimited by the sequence `@;` and extend to the end of the line. Whitespace can occur between the `@` and `;`. A comment which begins on a line swallows that entire line, as well as the newline which terminates it. In essence, the entire comment line disappears. If the comment follows some material in a line, then it does not consume the newline. Thus, the following two queries are equivalent:

1.
```
@a@; comment: match whole line against variable @a
@; this comment disappears entirely
@b
```

2.
```
@a
@b
```

The comment after the `@a` does not consume the newline, but the comment which follows does. Without this intuitive behavior, line comment would give rise to empty lines that must match empty lines in the data, leading to spurious mismatches.

Instead of the `;` character, the `#` character can be used. This is an obsolescent feature.

6.2 Hash-Bang Support

TXR has several features which support use of the hash-bang convention for creating apparently standalone executable programs.

6.2.1 Basic Hash Bang

Special processing is applied to **TXR** query or **TXR Lisp** script files that are specified on the command line via the `-f` option or as the first non-option argument. If the first line of such a file begins with the characters `#!`, that entire line is consumed and processed specially.

This removal allows for **TXR** queries to be turned into standalone executable programs in the POSIX environment using the hash-bang mechanism. Unlike most interpreters, **TXR** applies special processing to the `#!` line, which is described below, in the section **Argument Generation with the Null Hack**.

Shell session example: create a simple executable program called `"twoline.txr"` and run it. This assumes **TXR** is installed in `/usr/bin`.

```
$ cat > hello.txr
#!/usr/bin/txr
@(bind a "Hey")
@(output)
Hello, world!
@(end)
$ chmod a+x hello.txr
$ ./hello.txr
Hello, world!
```

When this plain hash-bang line is used, **TXR** receives the name of the script as an argument. Therefore, it is not possible to pass additional options to **TXR**. For instance, if the above script is invoked like this

```
$ ./hello.txr -B
```

the `-B` option isn't processed by **TXR**, but treated as an additional argument, just as if `txr` *`scriptfile`* `-B` had been executed directly.

This behavior is useful if the script author wants not to expose the **TXR** options to the user of the script.

However, the hash-bang line can use the `-f` option:

```
#!/usr/bin/txr -f
```

Now, the name of the script is passed as an argument to the `-f` option, and **TXR** will look for more options after that, so that the resulting program appears to accept **TXR** options. Now we can run

```
$ ./hello.txr -B
Hello, world!
a="Hey"
```

The `-B` option is honored.

6.2.2 Argument Generation with `--args` and `--eargs`

On some operating systems, it is not possible to pass more than one argument through the hash-bang mechanism. That is to say, this will not work.

```
#!/usr/bin/txr -B -f
```

To support systems like this, **TXR** supports the special argument `--args`, as well as an extended version, `--eargs`. With `--args`, it is possible to encode multiple arguments into one argument. The `--args`

option must be followed by a separator character, chosen by the programmer. The characters after that are split into multiple arguments on the separator character. The `--args` option is then removed from the argument list and replaced with these arguments, which are processed in its place.

Example:

```
#!/usr/bin/txr --args:-B:-f
```

The above has the same behavior as

```
#!/usr/bin/txr -B -f
```

on a system which supports multiple arguments in the hash-bang line. The separator character is the colon, and so the remainder of that argument, `-B:-f`, is split into the two arguments `-B -f`.

The `--eargs` option is similar to `--args`, but must be followed by one more argument. After `--eargs` performs the argument splitting in the same manner as `--args`, any of the arguments which it produces which are the two-character sequence `{}` are replaced with that following argument. Whether or not the replacement occurs, that following argument is then removed.

Example:

```
#!/usr/bin/txr --eargs:-B:{}:--foo:42
```

This has an effect which cannot be replicated in any known implementation of the hash-bang mechanism. Suppose that this hash-bang line is placed in a script called `script.txr`. When this script is invoked with arguments, as in:

```
script.txr a b c
```

then **TXR** is invoked similarly to:

```
/usr/bin/txr --eargs:-B:{}:--foo:42 script.txr a b c
```

Then, when `--eargs` processing takes place, firstly the argument sequence

```
-B {} --foo 42
```

is produced by splitting into four fields using the `:` (colon) character as the separator. Then, within these four fields, all occurrences of `{}` are replaced with the following argument `script.txr`, resulting in:

```
-B script.txr --foo 42
```

Furthermore, that `script.txr` argument is removed from the remaining argument list.

The four arguments are then substituted in place of the original `--eargs:-B:{}:--foo:42` syntax.

The resulting **TXR** invocation is, therefore:

```
/usr/bin/txr -B script.txr --foo 42 a b c
```

Thus, `--eargs` allows some arguments to be encoded into the interpreter script, such that script name is inserted anywhere among them, possibly multiple times. Arguments for the interpreter can be encoded, as well as arguments to be processed by the script.

6.2.3 Argument Generation with the Null Hack

The `--args` and `--eargs` mechanisms do not solve the following problem: the POSIX `env` utility is often exploited for its `PATH` searching capability, and used to express hash-bang scripts in the following way:

```
#!/usr/bin/env txr
```

Here, the `env` utility searches for the `txr` program in the directories indicated by the `PATH` variable, which liberates the script from having to encode the exact location where the program is installed. However, if the operating system allows only one argument in the hash-bang mechanism, then no arguments can be passed to the program.

To mitigate this problem, **TXR** supports a special feature in its hash-bang support. If the hash-bang line contains a null byte, then the text from after the null byte until the end of the line is split into fields using the space character as a separator, and these fields are inserted into the command line. This manipulation happens during command-line processing, i.e. prior to the execution of the file. If this processing is applied to a file that is specified using the `-f` option, then the arguments which arise from the special processing are inserted after that option and its argument. If this processing is applied to the file which is the first non-option argument, then the options are inserted before that argument. However, care is taken not to process that argument a second time. In either situation, processing of the command-line options continues, and the arguments which are processed next are the ones which were just inserted. This is true even if the options had been inserted as a result of processing the first non-option argument, which would ordinarily signal the termination of option processing.

In the following examples, it is assumed that the script is named, and invoked, as `/home/jenny/foo.txr`, and is given arguments `--bar abc`, and that `txr` resolves to `/usr/bin/txr`. The `<NUL>` code indicates a literal ASCII NUL character (the zero byte).

Basic example:

```
#!/usr/bin/env txr<NUL>-a 3
```

Here, `env` searches for `txr`, finding it in `/usr/bin`. Thus, including the executable name, **TXR** receives this full argument list:

```
/usr/bin/txr /home/jenny/foo.txr --bar abc
```

The first non-option argument is the name of the script. **TXR** opens the script, and notices that it begins with a hash-bang line. It consumes the hash-bang line and finds the null byte inside it, retrieving the character string after it, which is `"-a 3"`. This is split into the two arguments `-a` and `3`, which are then inserted into the command line ahead of the the script name. The effective command line then becomes:

```
/usr/bin/txr -a 3 /home/jenny/foo.txr --bar abc
```

Command-line option processing continues, beginning with the `-a` option. After the option is processed, `/home/jenny/foo.txr` is encountered again. This time it is not opened a second time; it signals the end of option processing, exactly as it would immediately do if it hadn't triggered the insertion of any arguments.

Advanced example: use `env` to invoke `txr`, passing options to the interpreter and to the script:

```
#!/usr/bin/env txr<NUL>--eargs:-C:175:{}:--debug
```

This example shows how `--eargs` can be used in conjunction with the null hack. When `txr` begins

executing, it receives the arguments

```
/usr/bin/txr /home/jenny/foo.txr
```

The script file is opened, and the arguments delimited by the null character in the hash-bang line are inserted, resulting in the effective command line:

```
/usr/bin/txr --eargs:-C:175:{}:--debug /home/jenny/foo.txr
```

Next, `--eargs` is processed in the ordinary way, transforming the command line into:

```
/usr/bin/txr -C 175 /home/jenny/foo.txr --debug
```

The name of the script file is encountered, and signals the end of option processing. Thus `txr` receives the `-C` option, instructing it to emulate some behaviors from version 175, and the `/home/jenny/foo.txr` script receives `--debug` as **its** argument: it executes with the `*args*` list containing one element, the character string `"--debug"`.

The hash-bang null-hack feature was introduced in **TXR** 177. Previous versions ignore the hash-bang line, performing no special processing. Where a risk exists that programs which depend on the feature might be executed by an older version of **TXR**, care must be taken to detect and handle that situation, either by means of the `txr-version` variable, or else by some logic which infers that the processing of the hash-bang line hasn't been performed.

6.2.4 Passing Options to TXR via Hash-Bang Null Hack

It is possible to use the Hash-Bang Null Hack, such that the resulting executable program recognizes **TXR** options. This is made possible by a special behavior in the processing of the `-f` option.

For instance, suppose that the effect of the following familiar hash-bang line is required:

```
#!/path/to/txr -f
```

However, suppose there is also a requirement to use the `env` utility to find **TXR**. Furthermore, the operating system allows only one hash-bang argument. Using the Null Hack, this is rewritten as:

```
#!/usr/bin/env txr<NUL>-f
```

then if the script is invoked with arguments `-a b c`, the command line will ultimately be transformed into:

```
/path/to/txr -f /path/to/scriptfile -i a b c
```

which allows **TXR** to process the `-i` option, leaving `a`, `b` and `c` as arguments for the script.

However, note that there is a subtle issue with the `-f` option that has been inserted via the Null Hack: namely, this insertion happens after **TXR** has opened the script file and read the hash-bang line from it. This means that when the inserted `-f` option is being processed, the script file is already open. A special behavior occurs. The `-f` option processing notices that the argument to `-f` is identical to the pathname of name of the script file that **TXR** has already opened for processing. The `-f` option and its argument are then skipped.

6.2.5 Hash Bang and Setuid

TXR supports setuid hash-bang scripting, even on platforms that do not support setuid and setgid attributes on hash-bang scripts. On such platforms, **TXR** has to be installed setuid/setgid. See the section SETUID/SETGID OPERATION. On some platforms, it may also be necessary to to use the `--reexec` option.

6.3 Whitespace

Outside of directives, whitespace is significant in **TXR** queries, and represents a pattern match for white-space in the input. An extent of text consisting of an undivided mixture of tabs and spaces is a whitespace token.

Whitespace tokens match a precisely identical piece of whitespace in the input, with one exception: a whitespace token consisting of precisely one space has a special meaning. It is equivalent to the regular expression `@/[ ]+/`: match an extent of one or more spaces (but not tabs!). Multiple consecutive spaces do not have this meaning.

Thus, the query line `"a b"` (one space between `a` and `b`) matches `"a b"` with any number of spaces between the two letters.

For matching a single space, the syntax `@\` can be used (backslash-escaped space).

It is more often necessary to match multiple spaces than to match exactly one space, so this rule simplifies many queries and inconveniences only a few.

In output clauses, string and character literals and quasiliterals, a space token denotes a space.

6.4 Text

Query material which is not escaped by the special character `@` is literal text, which matches input character for character. Text which occurs at the beginning of a line matches the beginning of a line. Text which starts in the middle of a line, other than following a variable, must match exactly at the current position, where the previous match left off. Moreover, if the text is the last element in the line, its match is anchored to the end of the line.

An empty query line matches an empty line in the input. Note that an empty input stream does not contain any lines, and therefore is not matched by an empty line. An empty line in the input is represented by a newline character which is either the first character of the file, or follows a previous newline-terminated line.

Input streams which end without terminating their last line with a newline are tolerated, and are treated as if they had the terminator.

Text which follows a variable has special semantics, described in the section Variables below.

A query may not leave a line of input partially matched. If any portion of a line of input is matched, it must be entirely matched, otherwise a matching failure results. However, a query may leave unmatched lines. Matching only four lines of a ten-line file is not a matching failure. The `eof` directive can be used to explicitly match the end of a file.

In the following example, the query matches the text, even though the text has an extra line.

code:
```
Four score and seven
years ago our
```

data:
```
Four score and seven
years ago our
forefathers
```

In the following example, the query **fails** to match the text, because the text has extra material on one line that is not matched:

code:
```
I can carry nearly eighty gigs
in my head
```

data:
```
I can carry nearly eighty gigs of data
in my head
```

Needless to say, if the text has insufficient material relative to the query, that is a failure also.

To match arbitrary material from the current position to the end of a line, the "match any sequence of characters, including empty" regular expression `@/.*/` can be used. Example:

code:
```
I can carry nearly eighty gigs@/.*/
```

data:
```
I can carry nearly eighty gigs of data
```

In this example, the query matches, since the regular expression matches the string "of data". (See the Regular Expressions section below.)

Another way to do this is:

code:
```
I can carry nearly eighty gigs@(skip)
```

6.5 Special Characters in Text

Control characters may be embedded directly in a query (with the exception of newline characters). An alternative to embedding is to use escape syntax. The following escapes are supported:

`@\`*`newline`*
: A backslash immediately followed by a newline introduces a physical line break without breaking up the logical line. Material following this sequence continues to be interpreted as a continuation of the previous line, so that indentation can be introduced to show the continuation without appearing in the data.

`@\`*`space`*
: A backslash followed by a space encodes a space. This is useful in line continuations when it is necessary for some or all of the leading spaces to be preserved. For instance the two line sequence

```
abcd@\
  @\  efg
```

is equivalent to the line

```
abcd  efg
```

The two spaces before the `@\` in the second line are consumed. The spaces after are preserved.

`@\a` Alert character (ASCII 7, BEL).

`@\b` Backspace (ASCII 8, BS).

`@\t` Horizontal tab (ASCII 9, HT).

`@\n` Line feed (ASCII 10, LF). Serves as abstract newline on POSIX systems.

`@\v` Vertical tab (ASCII 11, VT).

`@\f` Form feed (ASCII 12, FF). This character clears the screen on many kinds of terminals, or ejects a page of text from a line printer.

`@\r` Carriage return (ASCII 13, CR).

`@\e` Escape (ASCII 27, ESC)

`@\x`*`hex-digits`*
: A `@\x` immediately followed by a sequence of hex digits is interpreted as a hexadecimal numeric character code. For instance `@\x41` is the ASCII character A. If a semicolon character immediately follows the hex digits, it is consumed, and characters which follow are not considered part of the hex escape even if they are hex digits.

`@\`*`octal-digits`*
: A `@\` immediately followed by a sequence of octal digits (0 through 7) is interpreted as an octal character code. For instance `@\010` is character 8, same as `@\b`. If a semicolon character immediately follows the octal digits, it is consumed, and subsequent characters are not treated as part of the octal escape, even if they are octal digits.

Note that if a newline is embedded into a query line with `@\n`, this does not split the line into two; it's embedded into the line and thus cannot match anything. However, `@\n` may be useful in the `@(cat)` directive and in `@(output)`.

6.6 Character Handling and International Characters

TXR represents text internally using wide characters, which are used to represent Unicode code points. Script source code, as well as all data sources, are assumed to be in the UTF-8 encoding. In **TXR** and **TXR Lisp** source, extended characters can be used directly in comments, literal text, string literals, quasiliterals and regular expressions. Extended characters can also be expressed indirectly using hexadecimal or octal escapes. On some platforms, wide characters may be restricted to 16 bits, so that **TXR** can only work with characters in the BMP (Basic Multilingual Plane) subset of Unicode.

TXR does not use the localization features of the system library; its handling of extended characters is not affected by environment variables like `LANG` and `L_CTYPE`. The program reads and writes only the UTF-8 encoding.

TXR deals with UTF-8 separately in its parser and in its I/O streams implementation.

TXR's text streams perform UTF-8 conversion internally, such that **TXR** applications use Unicode code points.

In text streams, invalid UTF-8 bytes are treated as follows. When an invalid byte is encountered in the middle of a multibyte character, or if the input ends in the middle of a multibyte character, or if an invalid character is decoded, such as an overlong from, or code in the range U+DC00 through U+DCFF, the UTF-8 decoder returns to the starting byte of the ill-formed multibyte character, and extracts just one byte, mapping that byte to the Unicode character range U+DC00 through U+DCFF, producing that code point as the decoded result. The decoder is then reset to its initial state and begins decoding at the following byte, where the same algorithm is repeated.

Furthermore, because **TXR** internally uses a null-terminated character representation of strings which easily interoperates with C language interfaces, when a null character is read from a stream, **TXR** converts it to the code U+DC00. On output, this code converts back to a null byte, as explained in the previous paragraph. By means of this representational trick, **TXR** can handle textual data containing null bytes.

In contrast to the above, the **TXR** parser scans raw UTF-8 bytes from a binary stream, rather than using a text stream. The parser performing its own recognition of UTF-8 sequences in certain language constructs, using a UTF-8 decoder only when processing certain kinds of tokens.

Comments are read without regard for encoding, so invalid encoding bytes in comments are not detected. A comment is simply a sequence of bytes terminated by a newline.

Invalid UTF-8 encountered while scanning identifiers and character names in character literal (hash-backslash) syntax is diagnosed as a syntax error.

UTF-8 in string literals is treated in the same way as UTF-8 in text streams. Invalid UTF-8 bytes are mapped into code points in the U+DC000 through U+DCFF range, and incorporated as such into the resulting string object which the literal denotes. The same remarks apply to regular-expression literals.

6.7 Regular Expression Directives

In place of a piece of text (see section Text above), a regular-expression directive may be used, which has the following syntax:

```
@/RE/
```

where the RE part enclosed in slashes represents regular-expression syntax (described in the section Regular Expressions below).

Long regular expressions can be broken into multiple lines using a backslash-newline sequence. Whitespace before the sequence or after the sequence is not significant, so the following two are equivalent:

```
@/reg \
  ular/

@/regular/
```

There may not be whitespace between the backslash and newline.

Whereas literal text simply represents itself, regular expression denotes a (potentially infinite) set of texts. The regular-expression directive matches the longest piece of text (possibly empty) which belongs to the set denoted by the regular expression. The match is anchored to the current position; thus if the directive is the first element of a line, the match is anchored to the start of a line. If the regular-expression directive is the last element of a line, it is anchored to the end of the line also: the regular expression must match the text from the current position to the end of the line.

Even if the regular expression matches the empty string, the match will fail if the input is empty, or has run out of data. For instance suppose the third line of the query is the regular expression `@/.*/`, but the input is a file which has only two lines. This will fail: the data has no line for the regular expression to match. A line containing no characters is not the same thing as the absence of a line, even though both abstractions imply an absence of characters.

Like text which follows a variable, a regular-expression directive which follows a variable has special semantics, described in the section Variables below.

6.8 Variables

Much of the query syntax consists of arbitrary text, which matches file data character for character. Embedded within the query may be variables and directives which are introduced by a `@` character. Two consecutive `@@` characters encode a literal `@`.

A variable-matching or substitution directive is written in one of several ways:

```
@sident
@{bident}
@*sident
@*{bident}
@{bident /regex/}
@{bident (fun [arg ...])}
@{bident number}
@{bident bident}
```

The forms with an `*` indicate a long match, see Longest Match below. The forms with the embedded regexp `/regex/` or function or *number* have special semantics; see Positive Match below.

The identifier `t` cannot be used as a name; it is a reserved symbol which denotes the value true. An attempt to use the variable `@t` will result in an exception. The symbol `nil` can be used where a variable name is required syntactically, but it has special semantics, described in a section below.

A *sident* is a "simple identifier" form which is not delimited by braces.

A *sident* consists of any combination of one or more letters, numbers, and underscores. It may not look like a number, so that for instance `123` is not a valid *sident*, but `12A` is valid. Case is sensitive, so that `FOO` is different from `foo`, which is different from `Foo`.

The braces around an identifier can be used when material which follows would otherwise be interpreted as being part of the identifier. When a name is enclosed in braces it is a *bident*.

The following additional characters may be used as part of a *bident* which are not allowed in a *sident*:

```
! $ % & * + - < = > ? \ ~
```

Moreover, most Unicode characters beyond U+007F may appear in a *bident*, with certain exceptions. A character may not be used if it is any of the Unicode space characters, a member of the high or low surrogate region, a member of any Unicode private-use area, or is either of the two characters U+FFFE and U+FFFF. These situations produce a syntax error. Invalid UTF-8 in an identifier is also a syntax error.

The rule still holds that a name cannot look like a number so `+123` is not a valid *bident* but these are valid: `a->b`, `*xyz*`, `foo-bar`.

The syntax `@FOO_bar` introduces the name `FOO_bar`, whereas `@{FOO}_bar` means the variable named `"FOO"` followed by the text `"_bar"`. There may be whitespace between the `@` and the name, or opening brace. Whitespace is also allowed in the interior of the braces. It is not significant.

If a variable has no prior binding, then it specifies a match. The match is determined from some current position in the data: the character which immediately follows all that has been matched previously. If a variable occurs at the start of a line, it matches some text at the start of the line. If it occurs at the end of a line, it matches everything from the current position to the end of the line.

6.9 Negative Match

If a variable is one of the plain forms

```
@sident
@{bident}
@*sident
```

```
@*{bident}
```

then this is a "negative match". The extent of the matched text (the text bound to the variable) is determined by looking at what follows the variable, and ranges from the current position to some position where the following material finds a match. This is why this is called a "negative match": the spanned text which ends up bound to the variable is that in which the match for the trailing material did not occur.

A variable may be followed by a piece of text, a regular-expression directive, a function call, a directive, another variable, or nothing (i.e. occurs at the end of a line). These cases are described in detail below.

6.9.1 Variable Followed by Nothing

If the variable is followed by nothing, the negative match extends from the current position in the data, to the end of the line. Example:

code: `a b c @FOO`

data: `a b c defghijk`

result: `FOO="defghijk"`

6.9.2 Variable Followed by Text

For the purposes of determining the negative match, text is defined as a sequence of literal text and regular expressions, not divided by a directive. So for instance in this example:

```
@a:@/foo/bcd e@(maybe)f@(end)
```

the variable `a` is considered to be followed by `":@/foo/bcd e"`.

If a variable is followed by text, then the extent of the negative match is determined by searching for the first occurrence of that text within the line, starting at the current position.

The variable matches everything between the current position and the matching position (not including the matching position). Any whitespace which follows the variable (and is not enclosed inside braces that surround the variable name) is part of the text. For example:

code: `a b @FOO e f`

data: `a b c d e f`

result: `FOO="c d"`

In the above example, the pattern text `"a b "` matches the data `"a b "`. So when the `@FOO` variable is processed, the data being matched is the remaining `"c d e f"`. The text which follows `@FOO` is `" e f"`. This is found within the data `"c d e f"` at position 3 (counting from 0). So positions 0–2 `("c d")` constitute the matching text which is bound to FOO.

6.9.3 Variable Followed by a Function Call or Directive

If the variable is followed by a function call, or a directive, the extent is determined by scanning the text for the first position where a match occurs for the entire remainder of the line. (For a description of functions, see Functions.)

For example:

```
@foo@(bind a "abc")xyz
```

Here, `@foo` will match the text from the current position to where `"xyz"` occurs, even though there is a `@(bind)` directive. Furthermore, if more material is added after the `"xyz"`, it is part of the search. Note

the difference between the following two:

```
@foo@/abc/@(func)
@foo@(func)@/abc/
```

In the first example, `@foo` matches the text from the current position until the match for the regular expression `"abc"`. `@(func)` is not considered when processing `@foo`. In the second example, `@foo` matches the text from the current position until the position which matches the function call, followed by a match for the regular expression. The entire sequence `@(func)@/abc/` is considered.

6.9.4 Consecutive Variables

If an unbound variable specifies a fixed-width match or a regular expression, then the issue of consecutive variables does not arise. Such a variable consumes text regardless of any context which follows it.

However, what if an unbound variable with no modifier is followed by another variable? The behavior depends on the nature of the other variable.

If the other variable is also unbound, and also has no modifier, this is a semantic error which will cause the query to fail. A diagnostic message will be issued, unless operating in quiet mode via `-q`. The reason is that there is no way to bind two consecutive variables to an extent of text; this is an ambiguous situation, since there is no matching criterion for dividing the text between two variables. (In theory, a repetition of the same variable, like `@FOO@FOO`, could find a solution by dividing the match extent in half, which would work only in the case when it contains an even number of characters. This behavior seems to have dubious value.)

An unbound variable may be followed by one which is bound. The bound variable is effectively replaced by the text which it denotes, and the logic proceeds accordingly.

It is possible for a variable to be bound to a regular expression. If `x` is an unbound variable and `y` is bound to a regular expression `RE`, then `@x@y` means `@x@/RE/`. A variable `v` can be bound to a regular expression using, for example, `@(bind v #/RE/)`.

The `@*` syntax for longest match is available. Example:

code: `@FOO:@BAR@FOO`

data: `xyz:defxyz`

result: `FOO=xyz, BAR=def`

Here, `FOO` is matched with `"xyz"`, based on the delimiting around the colon. The colon in the pattern then matches the colon in the data, so that `BAR` is considered for matching against `"defxyz"`. `BAR` is followed by `FOO`, which is already bound to `"xyz"`. Thus `"xyz"` is located in the `"defxyz"` data following `"def"`, and so BAR is bound to `"def"`.

If an unbound variable is followed by a variable which is bound to a list, or nested list, then each character string in the list is tried in turn to produce a match. The first match is taken.

An unbound variable may be followed by another unbound variable which specifies a regular expression or function call match. This is a special case called a "double variable match". What happens is that the text is searched using the regular expression or function. If the search fails, then neither variable is bound: it is a matching failure. If the search succeeds, then the first variable is bound to the text which is skipped by the search. The second variable is bound to the text matched by the regular expression or function. Example:

code: `@foo@{bar /abc/}`

data: `xyz@#abc`

result: `foo="xyz@#", BAR="abc"`

6.9.5 Consecutive Variables via Directive

Two variables can be de facto consecutive in a manner shown in the following example:

```
@var1@(all)@var2@(end)
```

This is treated just like the variable followed by directive. No semantic error is identified, even if both variables are unbound. Here, `@var2` matches everything at the current position, and so `@var1` ends up bound to the empty string.

Example 1: `b` matches at position 0 and `a` binds the empty string:

code: `@a@(all)@b@(end)`

data: `abc`

result: `a=""`
`b="abc"`

Example 2: `*a` specifies longest match (see Longest Match below), and so it takes everything:

code: `@*a@(all)@b@(end)`

data: `abc`

result: `a="abc"`
`b=""`

6.9.6 Longest Match

The closest-match behavior for the negative match can be overridden to longest match behavior. A special syntax is provided for this: an asterisk between the `@` and the variable, e.g.:

code: `a @*{FOO}cd`

data: `a b cdcdcdcd`

result: `FOO="b cdcdcd"`

code: `a @{FOO}cd`

data: `a b cdcdcd`

result: `FOO="b "`

In the former example, the match extends to the rightmost occurrence of `"cd"`, and so `FOO` receives `"b cdcdcd"`. In the latter example, the `*` syntax isn't used, and so a leftmost match takes place. The extent covers only the `"b "`, stopping at the first `"cd"` occurrence.

6.10 Positive Match

There are syntactic variants of variable syntax which have an embedded expression enclosed with the variable in braces:

```
@{bident /regex/}
@{bident (fun [args ...])}
@{bident number}
@{bident bident}
```

These specify a variable binding that is driven by a positive match derived from a regular expression, function or character count, rather than from trailing material (which is regarded as a "negative" match, since the variable is bound to material which is **skipped** in order to match the trailing material).

The positive match syntax is processed without considering any following syntax, and therefore may be followed by an unbound variable.

In the `@{bident /regex/}` form, the match extends over all characters from the current position which match the regular expression *regex*. (See the Regular Expressions section below.) If the variable already has a value, the text extracted by the regular expression must exactly match the variable.

In the `@{bident (fun [args ...])}` form, the match extends over lines or characters which are matched by the call to the function, if the call succeeds. Thus `@{x (y z w)}` is just like `@(y z w)`, except that the region of text skipped over by `@(y z w)` is also bound to the variable `x`. Except in one special case, the matching takes place horizontally within the current line, and the spanned range of text is treated as a string. The exception is that if the `@{bident (fun [args ...])}` appears as the only element of a line, and *fun* has a binding as a vertical function, then the function is invoked in the same manner as it would be by the `@(fun [args ...])` syntax. Then the variable indicated by *bident* is bound to the list of lines matched by the function call. Pattern functions are described in the Functions section below. The function is invoked even if the variable already has a value. The text matched by the function must match the variable.

In the `@{bident number}` form, the match processes a field of text which consists of the specified number of characters, which must be a nonnegative number. If the data line doesn't have that many characters starting at the current position, the match fails. A match for zero characters produces an empty string. The text which is actually bound to the variable is all text within the specified field, but excluding leading and trailing whitespace. If the field contains only spaces, then an empty string is extracted. This fixed-field extraction takes place whether or not the variable already has a binding. If it already has a binding, then it must match the extracted, trimmed text.

The `@{bident bident}` syntax allows the *number* or *regex* modifier to come from a variable. The variable must be bound and contain a nonnegative integer or regular expression. For example, `@{x y}` behaves like `@{x 3}` if `y` is bound to the integer 3. It is an error if `y` is unbound.

6.11 Special Symbols `nil` and `t`

Just like in the Common Lisp language, the names `nil` and `t` are special.

`nil` symbol stands for the empty list object, an object which marks the end of a list, and Boolean false. It is synonymous with the syntax `()` which may be used interchangeably with `nil` in most constructs.

In **TXR Lisp**, `nil` and `t` cannot be used as variables. When evaluated, they evaluate to themselves.

In the **TXR** pattern language, `nil` can be used in the variable binding syntax, but does not create a binding; it has a special meaning. It allows the variable-matching syntax to be used to skip material, in ways similar to the `skip` directive.

The `nil` symbol is also used as a `block` name, both in the **TXR** pattern language and in **TXR Lisp**. A block named `nil` is considered to be anonymous.

6.12 Keyword Symbols

Names beginning with the `:` (colon) character are keyword symbols. These also stand for themselves and may not be used as variables. Keywords are useful for labeling information and situations.

6.13 Regular Expressions

Regular expressions are a language for specifying sets of character strings. Through the use of pattern-matching elements, a regular expression is able to denote an infinite set of texts. **TXR** contains an original

implementation of regular expressions, which supports the following syntax:

`.` The period is a "wildcard" that matches any character.

`[]` Character class: matches a single character, from the set specified by special syntax written between the square brackets. This supports basic regexp character class syntax. POSIX notation like `[:digit:]` is not supported. The regex tokens `\s`, `\d` and `\w` are permitted in character classes, but not their complementing counterparts. These tokens simply contribute their characters to the class. The class `[a-zA-Z]` means match an uppercase or lowercase letter; the class `[0-9a-f]` means match a digit or a lowercase letter; the class `[^0-9]` means match a non-digit, and so forth. There are no locale-specific behaviors in **TXR** regular expressions; `[A-Z]` denotes an ASCII/Unicode range of characters. The class `[\d.]` means match a digit or the period character. A `]` or `-` can be used within a character class, but must be escaped with a backslash. A `^` in the first position denotes a complemented class, unless it is escaped by backslash. In any other position, it denotes itself. Two backslashes code for one backslash. So for instance `[\[\-]` means match a `[` or `-` character, `[^^]` means match any character other than `^`, and `[\^\\]` means match either a `^` or a backslash. Regex operators such as `*`, `+` and `&` appearing in a character class represent ordinary characters. The characters `-`, `]` and `^` occurring outside of a character class are ordinary. Unescaped `/` characters can appear within a character class. The empty character class `[]` matches no character at all, and its complement `[^]` matches any character, and is treated as a synonym for the `.` (period) wildcard operator.

`\s`, `\w` and `\d`
: These regex tokens each match a single character. The `\s` regex token matches a wide variety of ASCII whitespace characters and Unicode spaces. The `\w` token matches alphabetic word characters; it is equivalent to the character class `[A-Za-z_]`. The `\d` token matches a digit, and is equivalent to `[0-9]`.

`\S`, `\W` and `\D`
: These regex tokens are the complemented counterparts of `\s`, `\w` and `\d`. The `\S` token matches all those characters which `\s` does not match, `\W` matches all characters that `\w` does not match and `\D` matches nondigits.

`empty` An empty expression is a regular expression. It represents the set of strings consisting of the empty string; i.e. it matches just the empty string. The empty regex can appear alone as a full regular expression (for instance the **TXR** syntax `@//` with nothing between the slashes) and can also be passed as a subexpression to operators, though this may require the use of parentheses to make the empty regex explicit. For example, the expression `a|` means: match either `a`, or nothing. The forms `*` and `(*)` are syntax errors; though not useful, the correct way to match the empty expression zero or more times is the syntax `()*`.

`nomatch`
: The nomatch regular expression represents the empty set: it matches no strings at all, not even the empty string. There is no dedicated syntax to directly express nomatch in the regex language. However, the empty character class `[]` is equivalent to nomatch, and may be considered to be a notation for it. Other representations of nomatch are possible: for instance, the regex `~.*` which is the complement of the regex that denotes the set of all possible strings, and thus denotes the empty set. A nomatch has uses; for instance, it can be used to temporarily "comment out" regular expressions. The regex `([]abc|xyz)` is equivalent to `(xyz)`, since the `[]abc` branch cannot match anything. Using `[]` to "block" a subexpression allows you to leave it in place, then enable it later by removing the "block".

`(R)` If `R` is a regular expression, then so is `(R)`. The contents of parentheses denote one regular expression unit, so that for instance in `(RE)*`, the `*` operator applies to the entire parenthesized group. The syntax `()` is valid and equivalent to the empty regular expression.

`R?` Optionally match the preceding regular expression `R`.

`R*` Match the expression `R` zero or more times. This operator is sometimes called the "Kleene star", or "Kleene closure". The Kleene closure favors the longest match. Roughly speaking, if there are

two or more ways in which `R1*R2` can match, then that match occurs in which `R1*` matches the longest possible text.

`R+` Match the preceding expression `R` one or more times. Like `R*`, this favors the longest possible match: `R+` is equivalent to `RR*`.

`R1%R2` Match `R1` zero or more times, then match `R2`. If this match can occur in more than one way, then it occurs such that `R1` is matched the fewest number of times, which is opposite from the behavior of `R1*R2`. Repetitions of `R1` terminate at the earliest point in the text where a nonempty match for `R2` occurs. Because it favors shorter matches, `%` is termed a non-greedy operator. If `R2` is the empty expression, or equivalent to it, then `R1%R2` reduces to `R1*`. So for instance `(R%)` is equivalent to `(R*)`, since the missing right operand is interpreted as the empty regex. Note that whereas the expression `(R1*R2)` is equivalent to `(R1*)R2`, the expression `(R1%R2)` is **not** equivalent to `(R1%)R2`. Also note that `A(XY%Z)B` is equivalent to `AX(Y%Z)B`. This is because the precedence of `%` is higher than that of catenation on its left side; this rule prevents the given syntax from expressing the `XY` catenation. The expression may be understood as: `A(X(Y%Z))B` where the inner parentheses clarify how the syntax surrounding the `%` operator is being parsed, and the outer parentheses are superfluous. The correct way to assert catenation of `XY` as the left operand of `%` is `A(XY)%ZB`. To specify `XY` as the left operand, and limit the right operand to just `Z`, the correct syntax is `A((XY)%Z)B`. By contrast, the expression `A(X%YZ)B` is not equivalent to `A(X%Y)ZB` because the precedence of `%` is lower than that of catenation on its right side. The operator is effectively "bi-precedential".

`~R` Match the opposite of the following expression `R`; that is, match exactly those texts that `R` does not match. This operator is called complement, or logical not.

`R1R2` Two consecutive regular expressions denote catenation: the left expression must match, and then the right.

`R1|R2` Match either the expression `R1` or `R2`. This operator is known by a number of names: union, logical or, disjunction, branch, or alternative.

`R1&R2` Match both the expression `R1` and `R2` simultaneously; i.e. the matching text must be one of the texts which are in the intersection of the set of texts matched by `R1` and the set matched by `R2`. This operator is called intersection, logical and, or conjunction.

Any character which is not a regular-expression operator, a backslash escape, or the slash delimiter, denotes a one-position match of that character itself.

Any of the special characters, including the delimiting `/`, and the backslash, can be escaped with a backslash to suppress its meaning and denote the character itself.

Furthermore, all of the same escapes that are described in the section Special Characters in Text above are supported — the difference is that in regular expressions, the `@` character is not required, so for example a tab is coded as `\t` rather than `@\t`. Octal and hex character escapes can be optionally terminated by a semicolon, which is useful if the following characters are octal or hex digits not intended to be part of the escape.

Only the above escapes are supported. Unlike in some other regular-expression implementations, if a backlash appears before a character which isn't a regex special character or one of the supported escape sequences, it is an error. This wasn't true of historic versions of **TXR**. See the COMPATIBILITY section.

Precedence table, highest to lowest:

Operators	Class	Associativity
`(R) []`	primary	
`R? R+ R* R%...`	postfix	left-to-right

`R1R2`	catenation	left-to-right
`~R ...%R`	unary	right-to-left
`R1&R2`	intersection	left-to-right
`R1|R2`	union	left-to-right

The `%` operator is like a postfix operator with respect to its left operand, but like a unary operator with respect to its right operand. Thus `a~b%c~d` is `a(~(b%(c(~d))))`, demonstrating right-to-left associativity, where all of `b%` may be regarded as a unary operator being applied to `c~d`. Similarly, `a?*+%b` means `(((a?)*)+)%b`, where the trailing `%b` behaves like a postfix operator.

In **TXR**, regular expression matches do not span multiple lines. The regex language has no feature for multiline matching. However, the `@(freeform)` directive allows the remaining portion of the input to be treated as one string in which line terminators appear as explicit characters. Regular expressions may freely match through this sequence.

It's possible for a regular expression to match an empty string. For instance, if the next input character is `z`, facing the regular expression `/a?/`, there is a zero-character match: the regular expression's state machine can reach an acceptance state without consuming any characters. Examples:

code: `@A@/a?/@/.*/`

data: `zzzzz`

result: `A=""`

code: `@{A /a?/}@B`

data: `zzzzz`

result: `A="", B="zzzz"`

code: `@*A@/a?/`

data: `zzzzz`

result: `A="zzzzz"`

In the first example, variable `@A` is followed by a regular expression which can match an empty string. The expression faces the letter `z` at position 0 in the data line. A zero-character match occurs there, therefore the variable `A` takes on the empty string. The `@/.*/` regular expression then consumes the line.

Similarly, in the second example, the `/a?/` regular expression faces a `z`, and thus yields an empty string which is bound to `A`. Variable `@B` consumes the entire line.

The third example requests the longest match for the variable binding. Thus, a search takes place for the rightmost position where the regular expression matches. The regular expression matches anywhere, including the empty string after the last character, which is the rightmost place. Thus variable `A` fetches the entire line.

For additional information about the advanced regular-expression operators, see NOTES ON EXOTIC REGULAR EXPRESSIONS below.

6.14 Compound Expressions

If the `@` escape character is followed by an open parenthesis or square bracket, this is taken to be the start of a **TXR Lisp** compound expression.

The **TXR** language has the unusual property that its syntactic elements, so-called *directives*, are Lisp compound expressions. These expressions not only enclose syntax, but expressions which begin with certain symbols de facto behave as tokens in a phrase structure grammar. For instance, the expression `@(collect)` begins a block which must be terminated by the expression `@(end)`, otherwise there is a syntax

error. The `collect` expression can contain arguments which modify the behavior of the construct, for instance `@(collect :gap 0 :vars (a b))`. In some ways, this situation might be compared to HTML, in which an element such as `<a>` must be terminated by `</a>` and can have attributes such as `<a href="...">`.

Compound expressions contain subexpressions which are other compound expressions or literal objects of various kinds. Among these are: symbols, numbers, string literals, character literals, quasiliterals and regular expressions. These are described in the following sections. Additional kinds of literal objects exist, which are discussed in the TXR LISP section of the manual.

Some examples of compound expressions are:

```
(banana)

(a b c (d e f))

(  a (b (c d) (e  ) ))

("apple" #\b #\space 3)

(a #/[a-z]*/ b)

(_ `@file.txt`)
```

Symbols occurring in a compound expression follow a slightly more permissive lexical syntax than the *bident* in the syntax `@{`*bident*`}` introduced earlier. The `/` (slash) character may be part of an identifier, or even constitute an entire identifier. In fact a symbol inside a directive is a *lident*. This is described in the Symbol Tokens section under TXR LISP. A symbol must not be a number; tokens that look like numbers are treated as numbers and not symbols.

6.15 Character Literals

Character literals are introduced by the `#\` (hash-backslash) syntax, which is either followed by a character name, the letter `x` followed by hex digits, the letter `o` followed by octal digits, or a single character. Valid character names are:

```
nul                 linefeed            return
alarm               newline             esc
backspace           vtab                space
tab                 page                pnul
```

For instance `#\esc` denotes the escape character.

This convention for character literals is similar to that of the Scheme language. Note that `#\linefeed` and `#\newline` are the same character. The `#\pnul` character is specific to **TXR** and denotes the `U+DC00` code in Unicode; the name stands for "pseudo-null", which is related to its special function. For more information about this, see the section "Character Handling and International Characters".

6.16 String Literals

String literals are delimited by double quotes. A double quote within a string literal is encoded using `\"` and a backslash is encoded as `\\`. Backslash escapes like `\n` and `\t` are recognized, as are hexadecimal escapes like `\xFF` or `\xabc` and octal escapes like `\123`. Ambiguity between an escape and subsequent text can be resolved by adding a semicolon delimiter after the escape: `"\xabc;d"` is a string consisting of the character `U+0ABC` followed by `"d"`. The semicolon delimiter disappears. To write a literal semicolon

immediately after a hex or octal escape, write two semicolons, the first of which will be interpreted as a delimiter. Thus, `"\x21;;"` represents `"!;"`.

Note that the source code syntax of **TXR** string literals is specified in UTF-8, which is decoded into an internal string representation consisting of code points. The numeric escape sequences are an abstract syntax for specifying code points, not for specifying bytes to be inserted into the UTF-8 representation, even if they lie in the 8-bit range. Bytes cannot be directly specified, other than literally. However, when a **TXR** string object is encoded to UTF-8, every code point lying in the range U+DC00 through U+DCFF is converted to a single byte by taking the low-order eight bits of its value. By manipulating code points in this special range, **TXR** programs can reproduce arbitrary byte sequences in text streams. Also note that the `\u` escape sequence for specifying code points found in some languages is unnecessary and absent, since the existing hexadecimal and octal escapes satisfy this requirement. More detailed information is given in the earlier section Character Handling and International Characters.

If the line ends in the middle of a literal, it is an error, unless the last character is a backslash. This backslash is a special escape which does not denote a character; rather, it indicates that the string literal continues on the next line. The backslash is deleted, along with whitespace which immediately precedes it, as well as leading whitespace in the following line. The escape sequence `"\ "` (backslash space) can be used to encode a significant space.

Example:

```
"foo   \
 bar"

"foo   \
\ bar"

"foo\  \
 bar"
```

The first string literal is the string `"foobar"`. The second two are `"foo bar"`.

6.17 Word List Literals

A word list literal (WLL) provides a convenient way to write a list of strings when such a list can be given as whitespace-delimited words.

There are two flavors of the WLL: the regular WLL which begins with `#"` (hash, double quote) and the splicing list literal which begins with `#*"` (hash, star, double quote).

Both types are terminated by a double quote, which may be escaped as `\"` in order to include it as a character. All the escaping conventions used in string literals can be used in word literals.

Unlike in string literals, whitespace (tabs and spaces) is not significant in word literals: it separates words. A whitespace character may be escaped with a backslash in order to include it as a literal character.

Just like in string literals, an unescaped newline character is not allowed. A newline preceded by a backslash is permitted. Such an escaped backslash, together with any leading and trailing unescaped whitespace, is removed and replaced with a single space.

Example:

```
#"abc def ghi"   --> notates ("abc" "def" "ghi")
```

```
#"abc   def \
    ghi"         --> notates ("abc" "def" "ghi")

#"abc\ def ghi" --> notates ("abc def" "ghi")

#"abc\ def\ \
 \ ghi"         --> notates ("abc def " " ghi")
```

A splicing word literal differs from a word literal in that it does not produce a list of string literals, but rather it produces a sequence of string literals that is merged into the surrounding syntax. Thus, the following two notations are equivalent:

```
(1 2 3 #*"abc def" 4 5 #"abc def")

(1 2 3 "abc" "def" 4 5 ("abc" "def"))
```

The regular WLL produced a single list object, but the splicing WLL expanded into multiple string literal objects.

6.18 String Quasiliterals

Quasiliterals are similar to string literals, except that they may contain variable references denoted by the usual `@` syntax. The quasiliteral represents a string formed by substituting the values of those variables into the literal template. If `a` is bound to `"apple"` and `b` to `"banana"`, the quasiliteral `` `one @a and two @{b}s` `` represents the string `"one apple and two bananas"`. A backquote escaped by a backslash represents itself. Unlike in directive syntax, two consecutive `@` characters do not code for a literal `@`, but cause a syntax error. The reason for this is that compounding of the `@` syntax is meaningful. Instead, there is a `\@` escape for encoding a literal `@` character. Quasiliterals support the full output variable syntax. Expressions within variable substitutions follow the evaluation rules of **TXR Lisp**. This hasn't always been the case: see the COMPATIBILITY section.

Quasiliterals can be split into multiple lines in the same way as ordinary string literals.

6.19 Quasiword List Literals

The quasiword list literals (QLLs) are to quasiliterals what WLLs are to ordinary literals. (See the above section Word List Literals.)

A QLL combines the convenience of the WLL with the power of quasistrings.

Just as in the case of WLLs, there are two flavors of the QLL: the regular QLL which begins with `#` ` (hash, backquote) and the splicing QLL which begins with `#*` ` (hash, star, backquote).

Both types are terminated by a backquote, which may be escaped as `\` ` in order to include it as a character. All the escaping conventions used in quasiliterals can be used in QLLs.

Unlike in quasiliterals, whitespace (tabs and spaces) is not significant in QLLs: it separates words. A whitespace character may be escaped with a backslash in order to include it as a literal character.

A newline is not permitted unless escaped. An escaped newline works exactly the same way as it does in WLLs.

Note that the delimiting into words is done before the variable substitution. If the variable `a` contains spaces, then `#` `@a` ` nevertheless expands into a list of one item: the string derived from `a`.

Examples:

```
#`abc @a ghi`  --> notates (`abc` `@a` `ghi`)

#`abc   @d@e@f \
ghi`            --> notates (`abc` `@d@e@f` `ghi`)

#`@a\ @b @c` --> notates (`@a @b` `@c`)
```

A splicing QLL differs from an ordinary QLL in that it does not produce a list of quasiliterals, but rather it produces a sequence of quasiliterals that is merged into the surrounding syntax.

6.20 Numbers

TXR supports integers and floating-point numbers.

An integer literal is made up of digits `0` through `9`, optionally preceded by a + or − sign. The character `,` (comma) may appear between digits, as a visual separator of no semantic significance. The digit sequence must start and end with a digit. Runs of consecutive commas are permitted. Commas outside of the digit sequence are interpreted as the Lisp unquote syntax.

Compatibility node: support for separator commas appeared in **TXR** 283. Older **TXR** versions will interpret commas in the middle of numeric constants as instances of the unquote syntax.

Examples:

```
123
-34
+0
-0
+23448352730498379238472938472323 4
-1,000,000,001
1,2,3,,4  ;; equivalent to 1234
```

Examples that are not integer tokens:

```
,123   ;; equivalent to (sys:unquote 123)
123,a  ;; equivalent to 123, followed by (sys:unquote a)
-,1    ;; symbol -  followed by (sys:unquote 1)
```

An integer constant can also be specified in hexadecimal using the prefix `#x` followed by an optional sign, followed by hexadecimal digits: `0` through `9` and the uppercase or lowercase letters `A` through `F`:

```
#xFF    ;; 255
#x-ABC  ;; -2748
```

These digits may contain separator commas, just as in the case of the decimal integer:

```
#xFFFF,FFFF,FFFF
```

Similarly, octal numbers are supported with the prefix `#o` followed by octal digits:

```
#o777         ;; 511
#o123,456     ;; 42797
```

and binary numbers can be written with a `#b` prefix:

```
#b1110        ;; 14
#b1111,1111   ;; 255
```

A comma between the radix prefix and digits is a syntax error:

```
#x,DEF5,549C  ;; Syntax error
#b,1001,1101  ;; Likewise
```

Note that the `#b` prefix is also used for buffer literals.

A floating-point literal is marked by the inclusion of a decimal point, the scientific E notation, or both. It is an optional sign, followed by a mantissa consisting of digits, a decimal point, more digits, and then an optional E notation consisting of the letter `e` or `E`, an optional + or − sign, and then digits indicating the exponent value. In the mantissa, the digits are not optional. At least one digit must either precede the decimal point or follow it. That is to say, a decimal point by itself is not a floating-point constant.

The digits of the mantissa may include separator commas, in the same manner as decimal integer literals, in both the integer and fractional part. The digits of the exponent may not include separator commas.

Examples:

```
.123
123.
1E-3
20E40
.9E1
9.E19
-.5
+3E+3
1.E5
1,123,456.935,342E+013
```

Examples which are not floating-point constant tokens:

```
.      ;; dot token, not a number
123E   ;; the symbol 123E
1.0E-  ;; syntax error: invalid floating point constant
1.0E   ;; syntax error: invalid floating point constant
1.E    ;; syntax error: invalid floating point literal
.e     ;; syntax error: dot token followed by symbol
,1.0   ;; equivalent to (sys:unquote 1.0)
```

In **TXR** there is a special "dotdot" token consisting of two consecutive periods. An integer constant followed immediately by dotdot is recognized as such; it is not treated as a floating constant followed by a dot. That is to say, `123..` does not mean `123. .` (floating point `123.0` value followed by dot token). It means `123 ..` (integer `123` followed by `..` token).

Dialect Note: unlike in Common Lisp, `123.` is not an integer, but the floating-point number `123.0`.

Integers within a certain small range centered on zero have `fixnum` type. Values in the `fixnum` range fit into a Lisp value directly, not requiring heap allocation. A value which is implemented as a reference to a heap-allocated object is called *boxed*, whereas a self-contained value not referencing any storage elsewhere

is called *unboxed*. Thus values in the `fixnum` are unboxed; those outside of the range have `bignum` type instead, and are boxed. The variables `fixnum-min` and `fixnum-max` indicate the range.

Floating-point values are all unboxed if **TXR** is built with "NaN boxing" enabled, otherwise they are all boxed. The Lisp expression `(eq (read "0.0") (read "0.0"))` returns `t` under NaN boxing, indicating that the two instances of 0.0 are the same object. In the absence of NaN boxing, the two `read` calls produce distinct, boxed representations of 0.0, which compare unequal under `eq`. (The expression `(eq 0.0 0.0)` may not be relied upon if it is compiled, since compilation may deduplicate identical boxed literals, leading to a false positive.)

6.21 Comments

Comments of the form `@;` were introduced earlier. Inside compound expressions, another convention for comments exists: Lisp comments, which are introduced by the `;` (semicolon) character and span to the end of the line.

Example:

```
@(foo  ; this is a comment
  bar  ; this is another comment
  )
```

This is equivalent to `@(foo bar)`.

7 DIRECTIVES

7.1 Overview

When a **TXR Lisp** compound expression occurs in **TXR** preceded by a `@`, it is a *directive*.

Directives which are based on certain symbols are, additionally, involved in a phrase-structure syntax which uses Lisp expressions as if they were tokens.

For instance, the directive

```
@(collect)
```

not only denotes a compound expression with the `collect` symbol in its head position, but it also introduces a syntactic phrase which requires a matching `@(end)` directive. In other words, `@(collect)` is not only an expression, but serves as a kind of token in a higher-level, phrase-structure grammar.

Effectively, `collect` is a reserved symbol in the **TXR** language. A **TXR** program cannot use this symbol as the name of a pattern function due to its role in the syntax. The symbol has no reserved role in **TXR Lisp**.

Usually if this type of directive occurs alone in a line, not preceded or followed by other material, it is involved in a "vertical" (or line-oriented) syntax.

If such a directive is embedded in a line (has preceding or trailing material) then it is in a horizontal syntactic and semantic context (character-oriented).

There is an exception: the definition of a horizontal function looks like this:

```
@(define name (arg))body material@(end)
```

Yet, this is considered one vertical item, which means that it does not match a line of data. (This is necessary because all horizontal syntax matches something within a line of data, which is undesirable for definitions.)

Many directives exhibit both horizontal and vertical syntax, with different but closely related semantics. Some are vertical only, some are horizontal only.

A summary of the available directives follows:

`@(eof)`
: Explicitly match the end of file. Fails if unmatched data remains in the input stream. Can capture or match the termination status of a pipe.

`@(eol)`
: Explicitly match the end of line. Fails if the current position is not the end of a line. Also fails if no data remains (there is no current line).

`@(next)`
: Continue matching in another file or data source.

`@(block)`
: Groups together a sequence of directives into a logical name block, which can be explicitly terminated from within by using the `@(accept)` and `@(fail)` directives. Blocks are described in the section Blocks below.

`@(skip)`
: Treat the remaining query as a subquery unit, and search the lines (or characters) of the input file until that subquery matches somewhere. A `skip` is also an anonymous block.

`@(trailer)`
: Treat the remaining query or subquery as a match for a trailing context. That is to say, if the remainder matches, the data position is not advanced.

`@(freeform)`
: Treat the remainder of the input as one big string, and apply the following query line to that string. The newline characters (or custom separators) appear explicitly in that string.

`@(fuzz)`
: The `fuzz` directive, inspired by the patch utility, specifies a partial match for some lines.

`@(line)` and `@(chr)`
: These directives match a variable or expression against the current line number or character position.

`@(name)`
: Match a variable against the name of the current data source.

`@(data)`
: Match a variable against the remaining data (a lazy list of strings).

`@(some)`
: Multiple clauses are each applied to the same input. Succeeds if at least one of the clauses matches the input. The bindings established by earlier successful clauses are visible to the later clauses.

`@(all)`
: Multiple clauses are applied to the same input. Succeeds if and only if each one of the clauses matches. The clauses are applied in sequence, and evaluation stops on the first failure. The bindings established by earlier successful clauses are visible to the later clauses.

`@(none)`
: Multiple clauses are applied to the same input. Succeeds if and only if none of them match. The clauses are applied in sequence, and evaluation stops on the first success. No bindings are ever produced by this construct.

`@(maybe)`
: Multiple clauses are applied to the same input. No failure occurs if none of them match. The bindings established by earlier successful clauses are visible to the later clauses.

`@(cases)`
: Multiple clauses are applied to the same input. Evaluation stops on the first successful clause.

`@(require)`
: The `require` directive is similar to the `do` directive in that it evaluates one or more **TXR Lisp** expressions. If the result of the rightmost expression is `nil`, then `require` triggers a match failure. See the TXR LISP section far below.

`@(if)`, `@(elif)`, and `@(else)`
: The `if` directive with optional `elif` and `else` clauses allows one of multiple bodies of pattern-matching directives to be conditionally selected by testing the values of Lisp expressions. It is also available inside `@(output)` for conditionally selecting output clauses.

`@(choose)`
: Multiple clauses are applied to the same input. The one whose effect persists is the one which maximizes or minimizes the length of a particular variable.

`@(empty)`
: The `@(empty)` directive matches the empty string. It is useful in certain situations, such as expressing an empty match in a directive that doesn't accept an empty clause. The `@(empty)` syntax has another meaning in `@(output)` clauses, in conjunction with `@(repeat)`.

`@(define` *`name`* `(`*`args`* `...))`
: Introduces a function. Functions are described in the Functions section below.

`@(call` *`expr arg*`*`)`
: Performs function indirection. Evaluates *`expr`*, which must produce a symbol that names a pattern function. Then that pattern function is invoked.

`@(gather)`
: Searches text for matches for multiple clauses which may occur in arbitrary order. For convenience, lines of the first clause are treated as separate clauses.

`@(collect)`
: Search the data for multiple matches of a clause. Collect the bindings in the clause into lists, which are output as array variables. The `@(collect)` directive is line-oriented. It works with a multi-line pattern and scans line by line. A similar directive called `@(coll)` works within one line.

  A collect is an anonymous block.

`@(and)`
: Separator of clauses for `@(some)`, `@(all)`, `@(none)`, `@(maybe)` and `@(cases)`. Equivalent to `@(or)`. The choice is stylistic.

`@(or)` Separator of clauses for `@(some)`, `@(all)`, `@(none)`, `@(maybe)` and `@(cases)`. Equivalent to `@(and)`. The choice is stylistic.

`@(end)`
: Required terminator for `@(some)`, `@(all)`, `@(none)`, `@(maybe)`, `@(cases)`, `@(if)`, `@(collect)`, `@(coll)`, `@(output)`, `@(repeat)`, `@(rep)`, `@(try)`, `@(block)` and `@(define)`.

`@(fail)`
: Terminate the processing of a block, as if it were a failed match. Blocks are described in the section Blocks below.

`@(accept)`
: Terminate the processing of a block, as if it were a successful match. What bindings emerge may depend on the kind of block: `collect` has special semantics. Blocks are described in the section Blocks below.

`@(try)`
: Indicates the start of a try block, which is related to exception handling, described in the Exceptions section below.

`@(catch)` and `@(finally)`
: Special clauses within `@(try)`. See Exceptions below.

`@(defex)` and `@(throw)`
: Define custom exception types; throw an exception. See Exceptions below.

`@(assert)`
: The `assert` directive requires the following material to match, otherwise it throws an exception. It is useful for catching mistakes or omissions in parts of a query that are surefire matches.

`@(flatten)`
: Normalizes a set of specified variables to one-dimensional lists. Those variables which have a scalar value are reduced to lists of that value. Those which are lists of lists (to an arbitrary level of nesting) are converted to flat lists of their leaf values.

`@(merge)`
: Binds a new variable which is the result of merging two or more other variables. Merging has somewhat complicated semantics.

`@(cat)`
: Decimates a list (any number of dimensions) to a string, by catenating its constituent strings, with an optional separator string between all of the values.

`@(bind)`
: Binds one or more variables against a value using a structural pattern match. A limited form of unification takes place which can cause a match to fail.

`@(set)`
: Destructively assigns one or more existing variables using a structural pattern, using syntax similar to bind. Assignment to unbound variables triggers an error.

`@(rebind)`
: Evaluates an expression in the current binding environment, and then creates new bindings for the variables in the structural pattern. Useful for temporarily overriding variable values in a scope.

`@(forget)`
: Removes variable bindings.

`@(local)`
: Synonym of `@(forget)`.

`@(output)`
: A directive which encloses an output clause in the query. An output section does not match text, but produces text. The directives above are not understood in an output clause.

`@(repeat)`
: A directive understood within an `@(output)` section, for repeating multiline text, with successive substitutions pulled from lists. The directive `@(rep)` produces iteration over lists horizontally within one line. These directives have a different meaning in matching clauses, providing a shorthand notation for `@(collect :vars nil)` and `@(coll :vars nil)`, respectively.

`@(deffilter)`
: The `deffilter` directive is used for defining named filters, which are useful for filtering variable substitutions in output blocks. Filters are useful when data must be translated between different representations that have different special characters or other syntax, requiring escaping or similar treatment. Note that it is also possible to use a function as a filter. See Function Filters below.

  Named filters are stored in the hash table held in the Lisp special variable `*filters*`.

`@(filter)`
: The `filter` directive passes one or more variables through a given filter or chain or filters, updating them with the filtered values.

`@(load)` and `@(include)`
: The `load` and `include` directives allow **TXR** programs to be modularized. They bring in code from a file, in two different ways.

`@(do)` The `do` directive is used to evaluate **TXR Lisp** expressions, discarding their result values. See the TXR LISP section far below.

`@(mdo)`
The `mdo` (macro `do`) directive evaluates **TXR Lisp** expressions immediately, during the parsing of the **TXR** syntax in which it occurs.

`@(in-package)`
The `in-package` directive is used to switch to a different symbol package. It mirrors the **TXR Lisp** macro of the same name.

7.2 Subexpression Evaluation

Some directives contain subexpressions which are evaluated. Two distinct styles of evaluations occur in **TXR**: bind expressions and Lisp expressions. Which semantics applies to an expression depends on the syntactic context in which it occurs: which position in which directive.

The evaluation of **TXR Lisp** expressions is described in the TXR LISP section of the manual.

Bind expressions are so named because they occur in the `@(bind)` directive. **TXR** pattern function invocations also treat argument expressions as bind expressions.

The `@(rebind)`, `@(set)`, `@(merge)`, and `@(deffilter)` directives also use bind expression evaluation. Bind expression evaluation also occurs in the argument position of the `:tlist` keyword in the `@(next)` directive.

Unlike Lisp expressions, bind expressions do not support operators. If a bind expression is a nested list structure, it is a template denoting that structure. Any symbol in any position of that structure is interpreted as a variable. When the bind expression is evaluated, those corresponding positions in the template are replaced by the values of the variables.

Anywhere where a variable can appear in a bind expression's nested list structure, a Lisp expression can appear preceded by the `@` character. That Lisp expression is evaluated and its value is substituted into the bind expression's template.

Moreover, a Lisp expression preceded by `@` can be used as an entire bind expression. The value of that Lisp expression is then taken as the bind expression value.

Any object in a bind expression which is not a nested list structure containing Lisp expressions or variables denotes itself literally.

Examples:

In the following examples, the variables `a` and `b` are assumed to have the string values `"foo"` and `"bar"`, respectively.

The `->` notation indicates the value of each expression.

```
a               ->  "foo"
(a b)           ->  ("foo" "bar")
((a) ((b) b))   ->  (("foo") (("bar") "bar"))
(list a b)      ->  error: unbound variable list
@(list a b)     ->  ("foo" "bar") ;; Lisp expression
(a @[b 1..:])   ->  ("foo" "ar")  ;; Lisp eval of [b 1..:]
```

```
(a @(+ 2 2))   ->  ("foo" 4)     ;; Lisp eval of (+ 2 2)
#(a b)         ->  (a b)         ;; Vector literal, not list.
[a b]          ->  error: unbound variable dwim
```

The last example above `[a b]` is a notation equivalent to `(dwim a b)` and so follows similarly to the example involving `list`.

7.3 Input Scanning and Data Manipulation

7.3.1 The `next` directive

The `next` directive indicates that the remaining directives in the current block are to be applied against a new input source.

It can only occur by itself as the only element in a query line, and takes various arguments, according to these possibilities:

```
@(next)
@(next source [:nothrow] [:noclose])
@(next :args)
@(next :env)
@(next :list lisp-expr)
@(next :tlist bind-expr)
@(next :string lisp-expr)
@(next :var var)
@(next nil)
```

The lone `@(next)` without arguments specifies that subsequent directives will match inside the next file in the argument list which was passed to **TXR** on the command line.

If *`source`* is given, it must be a **TXR Lisp** expression which denotes an input source. Its value may be a string or an input stream. For instance, if variable `A` contains the text `"data"`, then `@(next A)` means switch to the file called `"data"`, and `@(next `@A.txt`)` means to switch to the file `"data.txt"`. The directive `@(next (open-command `git log`))` switches to the input stream connected to the output of the `git log` command.

If the input source cannot be opened for whatever reason, **TXR** throws an exception (see Exceptions below). An unhandled exception will terminate the program. Often, such a drastic measure is inconvenient; if `@(next)` is invoked with the `:nothrow` keyword, then if the input source cannot be opened, the situation is treated as a simple match failure. The `:nothrow` keyword also ensures that when the stream is later closed, which occurs when the lazy list reads all of the available data, the implicit call to the `close-stream` function specifies `nil` as the argument value to that function's *`throw-on-error-p`* parameter. This `:nothrow` mechanism does not suppress all exceptions related to the processing of that stream; unusual conditions encountered during the reading of data from the stream may throw exceptions.

When the subsequent directives which follow `@(next)` are processed, the directive terminates, and any stream which had been opened for *`source`* is closed. If the `:noclose` keyword is present, then this is prevented; the stream remains open. Note: keeping the stream open may be necessary if the `@(data)` directive is used to capture the input list into a variable whose value is used after the `@(next)` directive terminates, because the input list is lazy, and may depend on the stream continuing to be open.

The variant `@(next :args)` means that the remaining command-line arguments are to be treated as a data source. For this purpose, each argument is considered to be a line of text. The argument list does include that argument which specifies the file that is currently being processed or was most recently processed. As the arguments are matched, they are consumed. This means that if a `@(next)` directive

without arguments is executed in the scope of `@(next :args)`, it opens the file named by the first unconsumed argument.

To process arguments, and then continue with the original file and argument list, wrap the argument processing in a `@(block)`. When the block terminates, the input source and argument list are restored to what they were before the block.

The variant `@(next :env)` means that the list of process environment variables is treated as a source of data. It looks like a text file stream consisting of lines of the form `"name=value"`. If this feature is not available on a given platform, an exception is thrown.

The syntax `@(next :list` *`lisp-expr`*`)` treats **TXR Lisp** expression *`lisp-expr`* as a source of text. The value of *`lisp-expr`* is flattened to a simple list in a way similar to the `@(flatten)` directive. The resulting list is treated as if it were the lines of a text file: each element of the list must be a string, which represents a line. If the strings happen contain embedded newline characters, they are a visible constituent of the line, and do not act as line separators.

The syntax `@(next :tlist` *`bind-expr`*`)` is similar to `@(next :list ...)` except that *`bind-expr`* is not a **TXR Lisp** expression, but a **TXR** bind expression.

The syntax `@(next :var` *`var`*`)` requires *`var`* to be a previously bound variable. The value of the variable is retrieved and treated like a list, in the same manner as under `@(next :list ...)`. Note that `@(next :var x)` is not always the same as `@(next :tlist x)`, because `:var x` strictly requires `x` to be a **TXR** variable, whereas the `x` in `:tlist x` is an expression which can potentially refer to Lisp variable.

The syntax `@(next :string` *`lisp-expr`*`)` treats expression *`lisp-expr`* as a source of text. The value of the expression must be a string. Newlines in the string are interpreted as line terminators.

A string which is not terminated by a newline is tolerated, so that:

```
@(next :string "abc")
@a
```

binds `a` to `"abc"`. Likewise, this is also the case with input files and other streams whose last line is not terminated by a newline.

However, watch out for empty strings, which are analogous to a correctly formed empty file which contains no lines:

```
@(next :string "")
@a
```

This will not bind `a` to `""`; it is a matching failure. The behavior of `:list` is different. The query

```
@(next :list "")
@a
```

binds `a` to `""`. The reason is that under `:list` the string `""` is flattened to the list `("")` which is not an empty input stream, but a stream consisting of one empty line.

The `@(next nil)` variant indicates that the following subquery is applied to empty data, and the list of data sources from the command line is considered empty. This directive is useful in front of **TXR** code which doesn't process data sources from the command line, but takes command-line arguments. The

`@(next nil)` incantation absolutely prevents **TXR** from trying to open the first command-line argument as a data source.

Note that the `@(next)` directive only redirects the source of input over the scope of subquery in which the that directive appears. For example, the following query looks for the line starting with `"xyz"` at the top of the file `"foo.txt"`, within a `some` directive. After the `@(end)` which terminates the `@(some)`, the `"abc"` is matched in the previous input stream which was in effect before the `@(next)` directive:

```
@(some)
@(next "foo.txt")
xyz@suffix
@(end)
abc
```

However, if the `@(some)` subquery successfully matched `"xyz@suffix"` within the file `foo.text`, there is now a binding for the `suffix` variable, which is visible to the remainder of the entire query. The variable bindings survive beyond the clause, but the data stream does not.

7.3.2 The `skip` directive

The `skip` directive considers the remainder of the query as a search pattern. The remainder is no longer required to strictly match at the current line in the current input stream. Rather, the current stream is searched, starting with the current line, for the first line where the entire remainder of the query will successfully match. If no such line is found, the `skip` directive fails. If a matching position is found, the remainder of the query is processed from that point.

The remainder of the query can itself contain `skip` directives. Each such directive performs a recursive subsearch.

Skip comes in vertical and horizontal flavors. For instance, skip and match the last line:

```
@(skip)
@last
@(eof)
```

Skip and match the last character of the line:

```
@(skip)@{last 1}@(eol)
```

The `skip` directive has two optional arguments, which are evaluated as **TXR Lisp** expressions. If the first argument evaluates to an integer, its value limits the range of lines scanned for a match. Judicious use of this feature can improve the performance of queries.

Example: scan until `"size: @SIZE"` matches, which must happen within the next 15 lines:

```
@(skip 15)
size: @SIZE
```

Without the range limitation, `skip` will keep searching until it consumes the entire input source. In a horizontal `skip`, the range-limiting numeric argument is expressed in characters, so that

```
abc@(skip 5)def
```

means: there must be a match for `"abc"` at the start of the line, and then within the next five characters, there must be a match for `"def"`.

Sometimes a skip is nested within a `collect`, or following another skip. For instance, consider:

```
@(collect)
begin @BEG_SYMBOL
@(skip)
end @BEG_SYMBOL
@(end)
```

The above `collect` iterates over the entire input. But, potentially, so does the embedded `skip`. Suppose that `"begin x"` is matched, but the data has no matching `"end x"`. The skip will search in vain all the way to the end of the data, and then the collect will try another iteration back at the beginning, just one line down from the original starting point. If it is a reasonable expectation that an `end x` occurs 15 lines of a `"begin x"`, this can be specified instead:

```
@(collect)
begin @BEG_SYMBOL
@(skip 15)
end @BEG_SYMBOL
@(end)
```

If the symbol `nil` is used in place of a number, it means to scan an unlimited range of lines; thus, `@(skip nil)` is equivalent to `@(skip)`.

If the symbol `:greedy` is used, it changes the semantics of the skip to longest match semantics. For instance, match the last three space-separated tokens of the line:

```
@(skip :greedy) @a @b @c
```

Without `:greedy`, the variable `@c` may match multiple tokens, and end up with spaces in it, because nothing follows `@c` and so it matches from any position which follows a space to the end of the line. Also note the space in front of `@a`. Without this space, `@a` will get an empty string.

A line-oriented example of greedy skip: match the last line without using `@(eof)`:

```
@(skip :greedy)
@last_line
```

There may be a second numeric argument. This specifies a minimum number of lines to skip before looking for a match. For instance, skip 15 lines and then search indefinitely for `begin ...`:

```
@(skip nil 15)
begin @BEG_SYMBOL
```

The two arguments may be used together. For instance, the following matches if and only if the 15th line of input starts with `begin` :

```
@(skip 1 15)
begin @BEG_SYMBOL
```

Essentially, `@(skip 1 n)` means "hard skip by *n* lines". `@(skip 1 0)` is the same as `@(skip 1)`, which is a noop, because it means: "the remainder of the query must match starting on the next line", or, more briefly, "skip exactly zero lines", which is the behavior if the `skip` directive is omitted altogether.

Here is one trick for grabbing the fourth line from the bottom of the input:

```
@(skip)
@fourth_from_bottom
@(skip 1 3)
@(eof)
```

Or using greedy skip:

```
@(skip :greedy)
@fourth_from_bottom
@(skip 1 3)
```

Non-greedy skip with the `@(eof)` directive has a slight advantage because the greedy skip will keep scanning even though it has found the correct match, then backtrack to the last good match once it runs out of data. The regular skip with explicit `@(eof)` will stop when the `@(eof)` matches.

7.3.3 Reducing Backtracking with Blocks

The `skip` directive can consume considerable CPU time when multiple skips are nested. Consider:

```
@(skip)
A
@(skip)
B
@(skip)
C
```

This is actually nesting: the second and third skips occur within the body of the first one, and thus this creates nested iteration. **TXR** is searching for the combination of skips which match the pattern of lines `A`, `B` and `C` with backtracking behavior. The outermost skip marches through the data until it finds `A` followed by a pattern match for the second skip. The second skip iterates to find `B` followed by the third skip, and the third skip iterates to find `C`. If `A` and `B` are only one line each, then this is reasonably fast. But suppose there are many lines matching `A` and `B`, giving rise to a large number of combinations of skips which match `A` and `B`, and yet do not find a match for `C`, triggering backtracking. The nested stepping which tries the combinations of `A` and `B` can give rise to a considerable running time.

One way to deal with the problem is to unravel the nesting with the help of blocks. For example:

```
@(block)
@  (skip)
A
@(end)
@(block)
@  (skip)
B
@(end)
@(skip)
C
```

Now the scope of each skip is just the remainder of the block in which it occurs. The first skip finds `A`, and then the block ends. Control passes to the next block, and backtracking will not take place to a block which completed (unless all these blocks are enclosed in some larger construct which backtracks, causing the blocks to be re-executed.

This rewrite is not equivalent, and cannot be used for instance in backreferencing situations such as:

```
@;
@; Find three lines anywhere in the input which are identical.
@;
@(skip)
@line
@(skip)
@line
@(skip)
@line
```

This example depends on the nested search-within-search semantics.

7.3.4 The `trailer` directive

The `trailer` directive introduces a trailing portion of a query or subquery which matches input material normally, but in the event of a successful match, does not advance the current position. This can be used, for instance, to cause `@(collect)` to match partially overlapping regions.

Trailer can be used in vertical context:

```
@(trailer)
directives
...
```

or horizontal:

```
@(trailer) directives ...
```

A vertical `trailer` prevents the vertical input position from advancing as it is matched by *directives*, whereas a horizontal `trailer` prevents the horizontal position from advancing. In other words, `trailer` performs matching without consuming the input, providing a lookahead mechanism.

Example:

```
@(collect)
@line
@(trailer)
@(skip)
@line
@(end)
```

This script collects each line which has a duplicate somewhere later in the input. Without the `@(trailer)` directive, this does not work properly for inputs like:

```
111
222
111
222
```

Without `@(trailer)`, the first duplicate pair constitutes a match which spans over the `222`. After that pair is found, the matching continues after the second `111`.

With the `@(trailer)` directive in place, the collect body, on each iteration, only consumes the lines matched prior to `@(trailer)`.

7.3.5 The `freeform` directive

The `freeform` directive provides a useful alternative to **TXR**'s line-oriented matching discipline. The `freeform` directive treats all remaining input from the current input source as one big line. The query line which immediately follows freeform is applied to that line.

The syntax variations are:

```
@(freeform)
... query line ..

@(freeform number)
... query line ..

@(freeform string)
... query line ..

@(freeform number string)
... query line ..
```

where *`number`* and *`string`* denote **TXR Lisp** expressions which evaluate to an integer or string value, respectively.

If *`number`* and *`string`* are both present, they may be given in either order.

If the *`number`* argument is given, its value limits the range of lines which are combined together. For instance `@(freeform 5)` means to only consider the next five lines to be one big line. Without this argument, `freeform` is "bottomless". It can match the entire file, which creates the risk of allocating a large amount of memory.

If the *`string`* argument is given, it specifies a custom line terminator. The default terminator is `"\n"`. The terminator does not have to be one character long.

Freeform does not convert the entire remainder of the input into one big line all at once, but does so in a dynamic, lazy fashion, which takes place as the data is accessed. So at any time, only some prefix of the data exists as a flat line in which newlines are replaced by the terminator string, and the remainder of the data still remains as a list of lines.

After the subquery is applied to the virtual line, the unmatched remainder of that line is broken up into multiple lines again, by looking for and removing all occurrences of the terminator string within the flattened portion.

Care must be taken if the terminator is other than the default `"\n"`. All occurrences of the terminator string are treated as line terminators in the flattened portion of the data, so extra line breaks may be introduced. Likewise, in the yet unflattened portion, no breaking takes place, even if the text contains occurrences of the terminator string. The extent of data which is flattened, and the amount of it which remains, depends entirely on the query line underneath `@(flatten)`.

In the following example, lines of data are flattened using $ as the line terminator.

code:
```
@(freeform "$")
@a$@b:
@c
@d
```

data:
```
1
2:3
4
```

output (`-B`):
```
a="1"
b="2"
c="3"
d="4"
```

The data is turned into the virtual line `1$2:3$4$`. The `@a$@b:` subquery matches the `1$2:` portion, binding `a` to `"1"`, and `b` to `"2"`. The remaining portion `3$4$` is then split into separate lines again according to the line terminator `$i`:

```
3
4
```

Thus the remainder of the query

```
@c
@d
```

faces these lines, binding `c` to `3` and `d` to `4`. Note that since the data does not contain dollar signs, there is no ambiguity; the meaning may be understood in terms of the entire data being flattened and split again.

In the following example, `freeform` is used to solve a tokenizing problem. The Unix password file has fields separated by colons. Some fields may be empty. Using freeform, we can join the password file using `":"` as a terminator. By restricting freeform to one line, we can obtain each line of the password file with a terminating `":"`, allowing for a simple tokenization, because now the fields are colon-terminated rather than colon-separated.

Example:

```
@(next "/etc/passwd")
@(collect)
@(freeform 1 ":")
@(coll)@{token /[^:]*/}:@(end)
@(end)
```

7.3.6 The `fuzz` directive

The `fuzz` directive allows for an imperfect match spanning a set number of lines. It takes two arguments, both of which are **TXR Lisp** expressions that should evaluate to integers:

```
@(fuzz m n)
  ...
```

This expresses that over the next *n* query lines, the matching strictness is relaxed a little bit. Only *m* out of those *n* lines have to match. Afterward, the rest of the query follows normal, strict processing.

In the degenerate situation where there are fewer than *n* query lines following the `fuzz` directive, then *m* of them must succeed anyway. (If there are fewer than *m*, then this is impossible.)

7.3.7 The `line` and `chr` directives

The `line` and `chr` directives perform binding between the current input line number or character position within a line, against an expression or variable:

```
@(line 42)
@(line x)
abc@(chr 3)def@(chr y)
```

The directive `@(line 42)` means "match the current input line number against the integer 42". If the current line is 42, then the directive matches, otherwise it fails. `line` is a vertical directive which doesn't consume a line of input. Thus, the following matches at the beginning of an input stream, and `x` ends up bound to the first line of input:

```
@(line 1)
@(line 1)
@(line 1)
@x
```

The directive `@(line x)` binds variable `x` to the current input line number, if `x` is an unbound variable. If `x` is already bound, then the value of `x` must match the current line number, otherwise the directive fails.

The `chr` directive is similar to `line` except that it's a horizontal directive, and matches the character position rather than the line position. Character positions are measured from zero, rather than one. `chr` does not consume a character. Hence the two occurrences of `chr` in the following example both match, and `x` takes the entire line of input:

```
@(chr 0)@(chr 0)@x
```

The argument of `line` or `chr` may be an `@`-delimited Lisp expression. This is useful for matching computed lines or character positions:

```
@(line @(+ a (* b c)))
```

7.3.8 The `name` directive

The `name` directive performs a binding between the name of the current data source and a variable or bind expression:

```
@(name na)
@(name "data.txt")
```

If `na` is an unbound variable, it is bound and takes on the name of the data source, such as a file name. If `na` is bound, then it has to match the name of the data source, otherwise the directive fails.

The directive `@(name "data.txt")` fails unless the current data source has that name.

7.3.9 The `data` directive

The `data` directive performs a binding between the unmatched data at the current position, and and a variable or bind expression. The unmatched data takes the form of a list of strings:

```
@(data d)
```

The binding is performed on object equality. If `d` is already bound, a matching failure occurs unless `d`

contains the current unmatched data.

Matching the current data has various uses.

For instance, two branches of pattern matching can, at some point, bind the current data into different variables. When those paths join, the variables can be bound together to create the assertion that the current data had been the same at those points:

```
@(all)
@  (skip)
foo
@  (skip)
bar
@  (data x)
@(or)
@  (skip)
xyzzy
@  (skip)
bar
@  (data y)
@(end)
@(require (eq x y))
```

Here, two branches of the `@(all)` match some material which ends in the line `bar`. However, it is possible that this is a different line. The `data` directives are used to create an assertion that the data regions matched by the two branches are identical. That is to say, the unmatched data `x` captured after the first `bar` and the unmatched data `y` captured after the second `bar` must be the same object in order for `@(require (eq x y))` to succeed, which implies that the same `bar` was matched in both branches of the `@(all)`.

Another use of `data` is simply to gain access to the trailing remainder of the unmatched input in order to print it, or do some special processing on it.

The `tprint` Lisp function is useful for printing the unmatched data as newline-terminated lines:

```
@(data remainder)
@(do (tprint remainder))
```

7.3.10 The `eof` directive

The `eof` directive, if not given any argument, matches successfully when no more input is available from the current input source.

In the following example, the *`line`* variable captures the text `"One-line file"` and then since that is the last line of input, the `eof` directive matches:

code:
```
@line
@(eof)
```

data:
```
One-line file
```

If the data consisted of two or more lines, `eof` would fail.

The `eof` directive may be given a single argument, which is a pattern that matches the termination status of the input source. This is useful when the input source is a process pipe. For the purposes of `eof`, sources

which are not process pipes have the symbol `t` as their termination status.

In the following example, which assumes the availability of a POSIX shell command interpreter in the host system, the variable *a* captures the string `"a"` and the *status* variable captures the integer value `5`, which is the termination status of the command:

```
@(next (open-command "echo a; exit 5"))
@a
@(eof status)
```

7.3.11 The `some`, `all`, `none`, `maybe`, `cases` and `choose` directives

These directives, called the parallel directives, combine multiple subqueries, which are applied at the same input position, rather than to consecutive input.

They come in vertical (line mode) and horizontal (character mode) flavors.

In horizontal mode, the current position is understood to be a character position in the line being processed. The clauses advance this character position by moving it to the right. In vertical mode, the current position is understood to be a line of text within the stream. A clause advances the position by some whole number of lines.

The syntax of these parallel directives follows this example:

```
@(some)
subquery1
.
.
.
@(and)
subquery2
.
.
.
@(and)
subquery3
.
.
.
@(end)
```

And in horizontal mode:

```
@(some)subquery1...@(and)subquery2...@(and)subquery3...@(end)
```

Long horizontal lines can be broken up with line continuations, allowing the above example to be written like this, which is considered a single logical line:

```
@(some)@\
   subquery1...@\
@(and)@\
   subquery2...@\
@(and)@\
   subquery3...@\
@(end)
```

The `@(some)`, `@(all)`, `@(none)`, `@(maybe)`, `@(cases)` or `@(choose)` must be followed by at least one subquery clause, and be terminated by `@(end)`. If there are two or more subqueries, these additional clauses are indicated by `@(and)` or `@(or)`, which are interchangeable. The separator and terminator directives also must appear as the only element in a query line.

The `choose` directive requires keyword arguments. See below.

The syntax supports arbitrary nesting. For example:

```
QUERY:              SYNTAX TREE:

@(all)              all -+
@  (skip)                +- skip -+
@  (some)                |        +- some -+
it                       |        |        +- TEXT
@  (and)                 |        |        +- and
@    (none)              |        |        +- none -+
was                      |        |        |        +- TEXT
@    (end)               |        |        |        +- end
@  (end)                 |        |        +- end
a dark                   |        +- TEXT
@(end)                   *- end
```

nesting can be indicated using whitespace between `@` and the directive expression. Thus, the above is an `@(all)` query containing a `@(skip)` clause which applies to a `@(some)` that is followed by the text line `"a dark"`. The `@(some)` clause combines the text line `"it"`, and a `@(none)` clause which contains just one clause consisting of the line `"was"`.

The semantics of the parallel directives is:

`@(all)`

Each of the clauses is matched at the current position. If any of the clauses fails to match, the directive fails (and thus does not produce any variable bindings). Clauses following the failed directive are not evaluated. Bindings extracted by a successful clause are visible to the clauses which follow, and if the directive succeeds, all of the combined bindings emerge.

`@(some [ :resolve (`*var* `...) ])`

Each of the clauses is matched at the current position. If any of the clauses succeed, the directive succeeds, retaining the bindings accumulated by the successfully matching clauses. Evaluation does not stop on the first successful clause. Bindings extracted by a successful clause are visible to the clauses which follow.

The `:resolve` parameter is for situations when the `@(some)` directive has multiple clauses that need to bind some common variables to different values: for instance, output parameters in functions. Resolve takes a list of variable name symbols as an argument. This is called the resolve set. If the clauses of `@(some)` bind variables in the resolve set, those bindings are not visible to later clauses. However, those bindings do emerge out of the `@(some)` directive as a whole. This creates a conflict: what if two or more clauses introduce different bindings for a variable in the resolve set? This is why it is called the resolve set: conflicts for variables in the resolve set are automatically resolved in favor of later directives.

Example:

```
@(some :resolve (x))
```

```
@  (bind a "a")
@  (bind x "x1")
@(or)
@  (bind b "b")
@  (bind x "x2")
@(end)
```

Here, the two clauses both introduce a binding for `x`. Without the `:resolve` parameter, this would mean that the second clause fails, because `x` comes in with the value `"x1"`, which does not bind with `"x2"`. But because `x` is placed into the resolve set, the second clause does not see the `"x1"` binding. Both clauses establish their bindings independently creating a conflict over `x`. The conflict is resolved in favor of the second clause, and so the bindings which emerge from the directive are:

```
a="a"
b="b"
x="x2"
```

`@(none)`
: Each of the clauses is matched at the current position. The directive succeeds only if all of the clauses fail. If any clause succeeds, the directive fails, and subsequent clauses are not evaluated. Thus, this directive never produces variable bindings, only matching success or failure.

`@(maybe)`
: Each of the clauses is matched at the current position. The directive always succeeds, even if all of the clauses fail. Whatever bindings are found in any of the clauses are retained. Bindings extracted by any successful clause are visible to the clauses which follow.

`@(cases)`
: Each of the clauses is matched at the current position. The clauses are matched, in order, at the current position. If any clause matches, the matching stops and the bindings collected from that clause are retained. Any remaining clauses after that one are not processed. If no clause matches, the directive fails, and produces no bindings.

`@(choose [ :longest` *var* `|` `:shortest` *var* `])`
: Each of the clauses is matched at the current position in order. In this construct, bindings established by an earlier clause are not visible to later clauses. Although any or all of the clauses can potentially match, the clause which succeeds is the one which maximizes or minimizes the length of the text bound to the specified variable. The other clauses have no effect.

For all of the parallel directives other than `@(none)` and `@(choose)`, the query advances the input position by the greatest number of lines that match in any of the successfully matching subclauses that are evaluated. The `@(none)` directive does not advance the input position.

For instance if there are two subclauses, and one of them matches three lines, but the other one matches five lines, then the overall clause is considered to have made a five line match at its position. If more directives follow, they begin matching five lines down from that position.

7.3.12 The `require` directive

The syntax of `@(require)` is:

```
@(require lisp-expression)
```

The `require` directive evaluates a **TXR Lisp** expression. (See TXR LISP far below.) If the expression yields a true value, then it succeeds, and matching continues with the directives which follow. Otherwise the directive fails.

In the context of the `require` directive, the expression should not be introduced by the `@` symbol; it is expected to be a Lisp expression.

Example:

```
@; require that 4 is greater than 3
@; This succeeds; therefore, @a is processed
@(require (> (+ 2 2) 3))
@a
```

7.3.13 The `if` directive

The `if` directive allows for conditional selection of pattern-matching clauses, based on the Boolean results of Lisp expressions.

A variant of the `if` directive is also available for use inside an `output` clauses, where it similarly allows for the conditional selection of output clauses.

The syntax of the `if` directive can be exemplified as follows:

```
@(if lisp-expr)
.
.
.
@(elif lisp-expr)
.
.
.
@(elif lisp-expr)
.
.
.
@(else)
.
.
.
@(end)
```

The `@(elif)` and `@(else)` clauses are all optional. If `@(else)` is present, it must be last, before `@(end)`, after any `@(elif)` clauses. Any of the clauses may be empty.

Example:

```
@(if (> (length str) 42))
foo: @a @b
@(else)
{@c}
@(end)
```

In this example, if the length of the variable `str` is greater than `42`, then matching continues with `"foo: @a b"`, otherwise it proceeds with `{@c}`.

More precisely, how the `if` directive works is as follows. The Lisp expressions are evaluated in order, starting with the `if` expression, then the `elif` expressions if any are present. If any Lisp expression yields a true result (any value other than `nil`) then evaluation of Lisp expressions stops. The corresponding clause of that Lisp expression is selected and pattern matching continues with that clause. The result of that clause (its success or failure, and any newly bound variables) is then taken as the result of the `if` directive. If none of the Lisp expressions yield true, and an `else` clause is present, then that clause is processed and its result determines the result of the `if` directive. If none of the Lisp expressions yield true, and there is no `else` clause, then the `if` directive is deemed to have trivially succeeded, allowing matching to continue with whatever directive follows it.

7.3.14 The Lisp `if` versus TXR `if`

The `@(output)` directive supports the embedding of Lisp expressions, whose values are interpolated into the output. In particular, Lisp `if` expressions are useful. For instance `@(if expr "A" "B")` reproduces `A` if `expr` yields a true value, otherwise `B`. Yet the `@(if)` directive is also supported in `@(output)`. How the apparent conflict between the two is resolved is that the two take different numbers of arguments. An `@(if)` which has no arguments at all is a syntax error. One that has one argument is the head of the `if` directive syntax which must be terminated by `@(end)` and which takes the optional `@(elif)` and `@(else)` clauses. An `@(if)` which has two or more arguments is parsed as a self-contained Lisp expression.

7.3.15 The `gather` directive

Sometimes text is structured as items that can appear in an arbitrary order. When multiple matches need to be extracted, there is a combinatorial explosion of possible orders, making it impractical to write pattern matches for all the possible orders.

The `gather` directive is for these situations. It specifies multiple clauses which all have to match somewhere in the data, but in any order.

For further convenience, the lines of the first clause of the `gather` directive are implicitly treated as separate clauses.

The syntax follows this pattern:

```
@(gather)
one-line-query1
one-line-query2
.
.
.
one-line-queryN
@(and)
multi
line
query1
.
.
.
@(and)
multi
line
query2
.
.
```

```
.
@(end)
```

The multiline clauses are optional. The `gather` directive takes keyword parameters, see below.

7.3.16 The `until` / `last` clause in `gather`

Similarly to `collect`, `gather` has an optional `until`/`last` clause:

```
@(gather)
...
@(until)
...
@(end)
```

How `gather` works is that the text is searched for matches for the single-line and multiline queries. The clauses are applied in the order in which they appear. Whenever one of the clauses matches, any bindings it produces are retained and it is removed from further consideration. Multiple clauses can match at the same text position. The position advances by the longest match from among the clauses which matched. If no clauses match, the position advances by one line. The search stops when all clauses are eliminated, and then the cumulative bindings are produced. If the data runs out, but unmatched clauses remain, the directive fails.

Example: extract several environment variables, which do not appear in a particular order:

```
@(next :env)
@(gather)
USER=@USER
HOME=@HOME
SHELL=@SHELL
@(end)
```

If the `until` or `last` clause is present and a match occurs, then the matches from the other clauses are discarded and the `gather` terminates. The difference between `until`/`last` is that any bindings bindings established in `last` are retained, and the input position is advanced past the matching material. The `until`/`last` clause has visibility to bindings established in the previous clauses in that same iteration, even though those bindings end up thrown away.

For consistency, the `:mandatory` keyword is supported in the `until`/`last` clause of `gather`. The semantics of using `:mandatory` in this situation is tricky. In particular, if it is in effect, and the `gather` terminates successfully by collecting all required matches, it will trigger a failure. On the other hand, if the `until` or `last` clause activates before all required matches are gathered, a failure also occurs, whether or not the clause is `:mandatory`.

Meaningful use of `:mandatory` requires that the gather be open-ended; it must allow some (or all) variables not to be required. The presence of the option means that for `gather` to succeed, all required variables must be gathered first, but then termination must be achieved via the `until`/`last` clause before all `gather` clauses are satisfied.

7.3.17 Keyword parameters in `gather`

The `gather` directive accepts the keyword parameter `:vars`. The argument to `:vars` is a list of required and optional variables. A required variable is specified as a symbol. An optional variable is specified as a two element list which pairs a symbol with a Lisp expression. That Lisp expression is evaluated and specifies the default value for the variable.

Example:

```
@(gather :vars (a b c (d "foo")))
...
@(end)
```

Here, `a`, `b` and `c` are required variables, and `d` is optional, with the default value given by the Lisp expression `"foo"`.

The presence of `:vars` changes the behavior in three ways.

Firstly, even if all the clauses in the `gather` match successfully and are eliminated, the directive will fail if the required variables do not have bindings. It doesn't matter whether the bindings are existing, or whether they are established by `gather`.

Secondly, if some of the clauses of `gather` did not match, but all of the required variables have bindings, then the directive succeeds. Without the presence of `:vars`, it would fail in this situation.

Thirdly, if `gather` succeeds (all required variables have bindings), then all of the optional variables which do not have bindings are given bindings to their default values.

The expressions which give the default values are evaluated whenever the `gather` directive is evaluated, whether or not their values are used.

7.3.18 The `collect` directive

The syntax of the `collect` directive is:

```
@(collect)
... lines of subquery
@(end)
```

or with an `until` or `last` clause:

```
@(collect)
... lines of subquery: main clause
@(until)
... lines of subquery: until clause
@(end)

@(collect)
... lines of subquery: main clause
@(last)
... lines of subquery: last clause
@(end)
```

The `repeat` symbol may be specified instead of `collect`, which changes the meaning:

```
@(repeat)
... lines of subquery
@(end)
```

The `@(repeat)` syntax is equivalent to `@(collect :vars nil)` and doesn't take the `:vars` clause. It accepts other `collect` parameters.

The subquery is matched repeatedly, starting at the current line. If it fails to match, it is tried starting at the subsequent line. If it matches successfully, it is tried at the line following the entire extent of matched data, if there is one. Thus, the collected regions do not overlap. (Overlapping behavior can be obtained: see the `@(trailer)` directive.)

Unless certain keywords are specified, or unless the collection is explicitly failed with `@(fail)`, it always succeeds, even if it collects nothing, and even if the `until`/`last` clause never finds a match.

If no `until`/`last` clause is specified, and the `collect` is not limited using parameters, the collection is unbounded: it consumes the entire data file.

7.3.19 The `until` / `last` clause in `collect`

If an `until`/`last` clause is specified, the collection stops when that clause matches at the current position.

If an `until` clause terminates `collect`, no bindings are collected at that position, even if the main clause matches at that position also. Moreover, the position is not advanced. The remainder of the query begins matching at that position.

If a `last` clause terminates `collect`, the behavior is different. Any bindings captured by the main clause are thrown away, just like with the `until` clause. However, the bindings in the `last` clause itself survive, and the position is advanced to skip over that material.

Example:

code:
```
@(collect)
@a
@(until)
42
@b
@(end)
@c
```

data:
```
1
2
3
42
5
6
```

result:
```
a[0]="1"
a[1]="2"
a[2]="3"
c="42"
```

The line `42` is not collected, even though it matches `@a`. Furthermore, the `@(until)` does not advance the position, so variable `c` takes `42`.

If the `@(until)` is changed to `@(last)` the output will be different:

result:
```
a[0]="1"
a[1]="2"
a[2]="3"
b="5"
c="6"
```

The `42` is not collected into a list, just like before. But now the binding captured by `@b` emerges. Furthermore, the position advances so variable now takes `6`.

The binding variables within the clause of a `collect` are treated specially. The multiple matches for each variable are collected into lists, which then appear as array variables in the final output.

Example:

code:
```
@(collect)
@a:@b:@c
@(end)
```

data:
```
John:Doe:101
Mary:Jane:202
Bob:Coder:313
```

result:
```
a[0]="John"
a[1]="Mary"
a[2]="Bob"
b[0]="Doe"
b[1]="Jane"
b[2]="Coder"
c[0]="101"
c[1]="202"
c[2]="313"
```

The query matches the data in three places, so each variable becomes a list of three elements, reported as an array.

Variables with list bindings may be referenced in a query. They denote a multiple match. The `-D` command-line option can establish a one-dimensional list binding.

The clauses of `collect` may be nested. Variable matches collated into lists in an inner `collect` are again collated into nested lists in the outer `collect`. Thus an unbound variable wrapped in N nestings of `@(collect)` will be an N-dimensional list. A one-dimensional list is a list of strings; a two-dimensional list is a list of lists of strings, etc.

It is important to note that the variables which are bound within the main clause of a `collect`, that is, the variables which are subject to collection, appear, within the `collect`, as normal one-value bindings. The collation into lists happens outside of the `collect`. So for instance in the query:

```
@(collect)
@x=@x
@(end)
```

The left `@x` establishes a binding for some material preceding an equal sign. The right `@x` refers to that binding. The value of `@x` is different in each iteration, and these values are collected. What finally comes out of the `collect` clause is a single variable called `x` which holds a list containing each value that was ever instantiated under that name within the `collect` clause.

Also note that the `until` clause has visibility over the bindings established in the main clause. This is true even in the terminating case when the `until` clause matches, and the bindings of the main clause are discarded.

7.3.20 Keyword parameters in `collect`

By default, `collect` searches the rest of the input indefinitely, or until the `until`/`last` clause matches. It skips arbitrary amounts of nonmatching material before the first match, and between matches.

Within the `@(collect)` syntax, it is possible to specify keyword parameters for additional control of the

behavior. A keyword parameter consist of a keyword symbol followed by an argument, enclosed within the `@(collect)` syntax. The following are the supported keywords.

`:maxgap` *n*

The `:maxgap` keyword takes a numeric argument *n*, which is a Lisp expression. It causes `collect` to terminate if it fails to find a match after skipping *n* lines from the starting position, or more than *n* lines since any successful match. For example,

```
@(collect :maxgap 5)
```

specifies that the gap between the current position and the first match for the body of the `collect`, or between consecutive matches can be no longer than five lines. A `:maxgap` value of `0` means that the collected regions must be adjacent and must match right from the starting position. For instance:

```
@(collect :maxgap 0)
M @a
@(end)
```

means: from here, collect consecutive lines of the form `"M ..."`. This will not search for the first such line, nor will it skip lines which do not match this form.

`:mingap` *n*

The `:mingap` keyword complements `:maxgap`, though not exactly. Its argument *n*, a Lisp expression, specifies a minimum number of lines which must separate consecutive matches. However, it has no effect on the distance from the starting position to the first match.

`:gap` *n*

The `:gap` keyword effectively specifies `:mingap` and `:maxgap` at the same time, and can only be used if these other two are not used. Thus:

```
@(collect :gap 1)
@a
@(end)
```

means: collect every other line starting with the current line.

`:times` *n*

This shorthand means the same thing as if

`:mintimes` *n* `:maxtimes` *n*

were specified. This means that exactly *n* matches must occur. If fewer occur, then `collect` fails. The `collect` stops once it achieves `n` matches.

`:mintimes` *n*

The argument *n* of the `:mintimes` keyword is a Lisp expression which specifies that at least *n* matches must occur, or else `collect` fails.

`:mintimes` *n*

The Lisp argument expression *n* of the `:mintimes` keyword specifies that at most *n* matches are collected.

`:lines` *n*

The argument *n* of the `:lines` keyword parameter is a Lisp expression which specifies the upper bound on how many lines should be scanned by `collect`, measuring from the starting position. The extent of the `collect` body is not counted. Example:

```
@(collect :lines 2)
foo: @a
bar: @b
baz: @c
@(end)
```

The above `collect` will look for a match only twice: at the current position, and one line down.

`:vars ({`*variable* | (*variable default-value*)`}*)`

The `:vars` keyword specifies a restriction on what variables will emanate from the `collect`. Its argument is a list of variable names. An empty list may be specified using empty parentheses or, equivalently, the symbol `nil`. The *default-value* element of the syntax is a Lisp expression. The behavior of the `:vars` keyword is specified in the following section, "Specifying variables in `collect`".

`:lists (`*variable*`*)`

The `:lists` keyword indicates a list of variables. After the `collect` terminates, each *variable* in the list which does not have a binding is bound to the empty list symbol `nil`. Unlike `:vars` the `:lists` mechanism doesn't assert that only the listed variables may emanate from the `collect`. It also doesn't assert that each iteration of the `collect` must bind each of those variables.

`:counter {`*variable* | (*variable starting-value*)`}`

The `:counter` keyword's argument is a variable name symbol, or a compound expression consisting of a variable name symbol and the **TXR Lisp** expression *starting-value*. If this keyword argument is specified, then a binding for *variable* is established prior to each repetition of the `collect` body, to an integer value representing the repetition count. By default, repetition counts begin at zero. If *starting-value* is specified, it must evaluate to a number. This number is then added to each repetition count, and *variable* takes on the resulting displaced value.

If there is an existing binding for *variable* prior to the processing of the `collect`, then the variable is shadowed.

The binding is collected in the same way as other bindings that are established in the `collect` body.

The repetition count only increments after a successful match.

The `variable` is visible to the `collect`'s `until`/`last` clause. If that clause is being processed after a successful match of the body, then *variable* holds an integer value. If the body fails to match, then the `until`/`last` clause sees a binding for `variable` with a value of `nil`.

7.3.21 Specifying variables in `collect`

Normally, any variable for which a new binding occurs in a `collect` block is collected. A `collect` clause may be "sloppy": it can neglect to collect some variables on some iterations, or bind some variables which are intended to behave like local temporaries, but end up collated into lists. Another issue is that the `collect` clause might not match anything at all, and then none of the variables are bound.

The `:vars` keyword allows the query writer to add discipline the `collect` body.

The argument to `:vars` is a list of variable specs. A variable spec is either a symbol, denoting a required variable, or a `(`*`symbol default-value`*`)` pair, where *`default-value`* is a Lisp expression whose value specifies a default value for the variable, which is optional.

When a `:vars` list is specified, it means that only the given variables can emerge from the successful `collect`. Any newly introduced bindings for other variables do not propagate. More precisely, whenever the `collect` body matches successfully, the following three rules apply:

1. If `:vars` specifies required variables, the `collect` body must bind all of them, or else must not bind any variable at all, whether listed in `:vars` or not, otherwise an exception of type `query-error` is thrown.
2. If `:vars` specifies required variables, and also specifies default variables, and the `collect` body binds no variable at all, then the default variables are not bound to their default values.
3. If `:vars` specifies optional variables, and all required variables are bound by the `collect` body, then all those optional variables that are not bound by the `collect` body are bound to their default values. Under this rule, if `:vars` specifies no required variables, that is deemed to be logically equivalent to all required variables being bound.

In the event that `collect` does not match anything, the variables specified in `:vars`, whether required or optional, are all bound to empty lists. These bindings are established after the processing of the `until/last` clause, if present.

Example:

```
@(collect :vars (a b (c "foo")))
@a @c
@(end)
```

Here, if the body `"@a @c"` matches, an error will be thrown because one of the mandatory variables is `b`, and the body neglects to produce a binding for `b`.

Example:

```
@(collect :vars (a (c "foo")))
@a @b
@(end)
```

Here, if `"@a @b"` matches, only `a` will be collected, but not `b`, because `b` is not in the variable list. Furthermore, because there is no binding for `c` in the body, a binding is created with the value `"foo"`, exactly as if `c` matched such a piece of text.

In the following example, the assumption is that `THIS NEVER MATCHES` is not found anywhere in the input but the line `THIS DOES MATCH` is found and has a successor which is bound to `a`. Because the body did not match, the `:vars a` and `b` should be bound to empty lists. But `a` is bound by the last clause to some text, so this takes precedence. Only `b` is bound to an empty list.

```
@(collect :vars (a b))
THIS NEVER MATCHES
@(last)
THIS DOES MATCH
@a
@(end)
```

The following means: do not allow any variables to propagate out of any iteration of the `collect` and therefore collect nothing:

```
@(collect :vars nil)
...
@(end)
```

Instead of writing `@(collect :vars nil)`, it is possible to write `@(repeat)`. `@(repeat)` takes all `collect` keywords, except for `:vars`. There is a `@(repeat)` directive used in `@(output)` clauses; that is a different directive.

7.3.22 Mandatory `until` and `last`

The `until`/`last` clause supports the option keyword `:mandatory`, exemplified by the following:

```
@(collect)
...
@(last :mandatory)
...
@(end)
```

This means that the `collect` **must** be terminated by a match for the `until`/`last` clause, or else by an explicit `@(accept)`.

Specifically, the `collect` cannot terminate due to simply running out of data, or exceeding a limit on the number of matches that may be collected. In those situations, if an `until` or `last` clause is present with `:mandatory`, the `collect` is deemed to have failed.

7.3.23 The `coll` directive

The `coll` directive is the horizontal version of `collect`. Whereas `collect` works with multiline clauses on line-oriented material, `coll` works within a single line. With `coll`, it is possible to recognize repeating regularities within a line and collect lists.

Regular-expression-based Positive Match variables work well with `coll`.

Example: collect a comma-separated list, terminated by a space.

code: `@(coll)@{A /[^, ]+/}@(until) @(end)@B`

data: `foo,bar,xyzzy blorch`

result:
```
A[0]="foo"
A[1]="bar"
A[2]="xyzzy"
B=blorch
```

Here, the variable `A` is bound to tokens which match the regular expression `/[^, ]+/`: nonempty sequence of characters other than commas or spaces.

Like `collect`, `coll` searches for matches. If no match occurs at the current character position, it tries at the next character position. Whenever a match occurs, it continues at the character position which follows the last character of the match, if such a position exists.

If not bounded by an until clause, it will exhaust the entire line. If the until clause matches, then the collection stops at that position, and any bindings from that iteration are discarded. Like collect, coll also supports an `until`/`last` clause, which propagates variable bindings and advances the position. The

`:mandatory` keyword is supported.

`coll` clauses nest, and variables bound within a coll are available to clauses within the rest of the `coll` clause, including the `until`/`last` clause, and appear as single values. The final list aggregation is only visible after the `coll` clause.

The behavior of `coll` leads to difficulties when a delimited variable are used to match material which is delimiter separated rather than terminated. For instance, entries in a comma-separated files usually do not appear as `"a,b,c,"` but rather `"a,b,c"`.

So for instance, the following result is not satisfactory:

code: `@(coll)@a @(end)`

data: `1 2 3 4 5`

result:
```
a[0]="1"
a[1]="2"
a[2]="3"
a[3]="4"
```

The `5` is missing because it isn't followed by a space, which the text-delimited variable match `"@a "` looks for. After matching "4 ", coll continues to look for matches, and doesn't find any. It is tempting to try to fix it like this:

code: `@(coll)@a@/ ?/@(end)`

data: `1 2 3 4 5`

result:
```
a[0]=""
a[1]=""
a[2]=""
a[3]=""
a[4]=""
a[5]=""
a[6]=""
a[7]=""
a[8]=""
```

The problem now is that the regular expression `/ ?/` (match either a space or nothing), matches at any position. So when it is used as a variable delimiter, it matches at the current position, which binds the empty string to the variable, the extent of the match being zero. In this situation, the `coll` directive proceeds character by character. The solution is to use positive matching: specify the regular expression which matches the item, rather than a trying to match whatever follows. The `collect` directive will recognize all items which match the regular expression:

code: `@(coll)@{a /[^ ]+/}@(end)`

data: `1 2 3 4 5`

result:
```
a[0]="1"
a[1]="2"
a[2]="3"
a[3]="4"
a[4]="5"
```

The `until` clause can specify a pattern which, when recognized, terminates the collection. So for instance, suppose that the list of items may or may not be terminated by a semicolon. We must exclude the semicolon from being a valid character inside an item, and add an until clause which recognizes a semicolon:

code: `@(coll)@{a /[^ ;]+/}@(until);@(end);`

data: `1 2 3 4 5;`

result:
```
a[0]="1"
a[1]="2"
a[2]="3"
a[3]="4"
a[4]="5"
```

Whether followed by the semicolon or not, the items are collected properly.

Note that the `@(end)` is followed by a semicolon. That's because when the `@(until)` clause meets a match, the matching material is not consumed.

This repetition can be avoided by using `@(last)` instead of `@(until)` since `@(last)` consumes the terminating material.

Instead of the above regular-expression-based approach, this extraction problem can also be solved with `cases`:

code: `@(coll)@(cases)@a @(or)@a@(end)@(end)`

data: `1 2 3 4 5`

result:
```
a[0]="1"
a[1]="2"
a[2]="3"
a[3]="4"
a[4]="5"
```

7.3.24 Keyword parameters in `coll`

The `@(coll)` directive takes most of the same parameters as `@(collect)`. See the section Keyword parameters in `collect` above. So for instance `@(coll :gap 0)` means that the collects must be consecutive, and `@(coll :maxtimes 2)` means that at most two matches will be collected. The `:lines` keyword does not exist, but there is an analogous `:chars` keyword.

The `@(coll)` directive takes the `:vars` keyword.

The shorthand `@(rep)` may be used instead of `@(coll :vars nil)`. `@(rep)` takes all keywords, except `:vars`.

7.3.25 The `flatten` directive

The `flatten` directive can be used to convert variables to one-dimensional lists. Variables which have a scalar value are converted to lists containing that value. Variables which are multidimensional lists are flattened to one-dimensional lists.

Example (without `@(flatten)`):

code:
```
@b
@(collect)
@(collect)
@a
@(end)
@(end)
```

data:
```
0
1
2
3
```

```
4
5
```

result:
```
b="0"
a_0[0]="1"
a_1[0]="2"
a_2[0]="3"
a_3[0]="4"
a_4[0]="5"
```

Example (with `@(flatten)`):

code:
```
@b
@(collect)
@(collect)
@a
@(end)
@(end)
@(flatten a b)
```

data:
```
0
1
2
3
4
5
```

result:
```
b="0"
a[0]="1"
a[1]="2"
a[2]="3"
a[3]="4"
a[4]="5"
```

7.3.26 The `merge` directive

The syntax of `merge` follows the pattern:

`@(merge` *`destination`* `[`*`sources`* `...])`

`destination` is a variable, which receives a new binding. *`sources`* are bind expressions.

The `merge` directive provides a way of combining collected data from multiple nested lists in a way which normalizes different nesting levels among the sources. This directive is useful for combining the results from collects at different levels of nesting into a single nested list such that parallel elements are at equal depth.

A new binding is created for the *`destination`* variable, which holds the result of the operation.

The `merge` directive performs its special function if invoked with at least three arguments: a destination and two sources.

The one-argument case `@(merge x)` binds a new variable `x` and initializes it with the empty list and is thus equivalent to `@(bind x)`. Likewise, the two-argument case `@(merge x y)` is equivalent to `@(bind x y)`, establishing a binding for `x` which is initialized with the value of `y`.

To understand what merge does when two sources are given, as in `@(merge C A B)`, we first have to define a property called depth. The depth of an atom such as a string is defined as `1`. The depth of an empty list is `0`. The depth of a nonempty list is one plus the depth of its deepest element. So for instance

`"foo"` has depth 1, `("foo")` has depth 2, and `("foo" ("bar"))` has depth three.

We can now define a binary (two argument) merge(A, B) function as follows. First, merge(A, B) normalizes the values A and B to produce a pair of values which have equal depth, as defined above. If either value is an atom it is first converted to a one-element list containing that atom. After this step, both values are lists; and the only way an argument has depth zero is if it is an empty list. Next, if either value has a smaller depth than the other, it is wrapped in a list as many times as needed to give it equal depth. For instance if A is `("a")` and B is `((("b" "c") ("d" "e")))` then A is converted to `((("a")))`. Finally, the list values are appended together to produce the merged result. In the case of the preceding two example values, the result is: `((("a"))) (("b" "c") ("d" "e")))`. The result is stored into a the newly bound destination variable `C`.

If more than two source arguments are given, these are merged by a left-associative reduction, which is to say that a three argument `merge(X, Y, Z)` is defined as `merge(merge(X, Y), Z)`. The leftmost two values are merged, and then this result is merged with the third value, and so on.

7.3.27 The `cat` directive

The `cat` directive converts a list variable into a single piece of text. The syntax is:

```
@(cat var [sep])
```

The *sep* argument is a Lisp expression whose value specifies a separating piece of text. If it is omitted, then a single space is used as the separator.

Example:

code:
```
@(coll)@{a /[^ ]+/}@(end)
@(cat a ":")
```

data:
```
1 2 3 4 5
```

result:
```
a="1:2:3:4:5"
```

7.3.28 The `bind` directive

The syntax of the `bind` directive is:

```
@(bind pattern bind-expression {keyword value}*)
```

The `bind` directive is a kind of pattern match, which matches one or more variables given in *pattern* against a value produced by the *bind-expression* on the right.

Variable names occurring in the *pattern* expression may refer to bound or unbound variables.

All variable references occurring in *bind-expression* must have a value.

Binding occurs as follows. The tree structure of *pattern* and the value of *bind-expression* are considered to be parallel structures.

Any variables in *pattern* which are unbound receive a new binding, which is initialized with the structurally corresponding piece of the object produced by *bind-expression*.

Any variables in *pattern* which are already bound must match the corresponding part of the value of *bind-expression*, or else the `bind` directive fails. Variables which are already bound are not altered, retaining their current values even if the matching is inexact.

The simplest `bind` is of one variable against itself, for instance binding `A` against `A`:

```
@(bind A A)
```

This will throw an exception if `A` is not bound. If `A` is bound, it succeeds, since `A` matches itself.

The next simplest `bind` binds one variable to another:

```
@(bind A B)
```

Here, if `A` is unbound, it takes on the same value as `B`. If `A` is bound, it has to match `B`, or the `bind` fails. Matching means that either

- `A` and `B` are the same text
- `A` is text, `B` is a list, and `A` occurs within `B`.
- vice versa: `B` is text, `A` is a list, and `B` occurs within `A`.
- `A` and `B` are lists and are either identical, or one is found as a substructure within the other.

The right-hand side does not have to be a variable. It may be some other object, like a string, quasiliteral, regexp, or list of strings, etc. For instance,

```
@(bind A "ab\tc")
```

will bind the string `"ab\tc"` to the variable `A` if `A` is unbound. If `A` is bound, this will fail unless `A` already contains an identical string. However, the right-hand side of a `bind` cannot be an unbound variable, nor a complex expression that contains unbound variables.

The left-hand side of `bind` can be a nested list pattern containing variables. The last item of a list at any nesting level can be preceded by a `.` (dot), which means that the variable matches the rest of the list from that position.

Example 1:

> Suppose that the list A contains `("now" "now" "brown" "cow").` Then the directive `@(bind (H N . C) A)`, assuming that `H`, `N` and `C` are unbound variables, will bind `H` to `"how"`, code N to `"now"`, and `C` to the remainder of the list `("brown" "cow").`
>
> Example: suppose that the list `A` is nested to two dimensions and contains `(("how" "now") ("brown" "cow")).` Then `@(bind ((H N) (B C)) A)` binds `H` to `"how"`, `N` to `"now"`, `B` to `"brown"` and `C` to `"cow"`.
>
> The dot notation may be used at any nesting level. it must be followed by an item. The forms `(.)` and `(X .)` are invalid, but `(. X)` is valid and equivalent to `X`.
>
> The number of items in a left pattern match must match the number of items in the corresponding right side object. So the pattern `()` only matches an empty list. The notations `()` and `nil` mean exactly the same thing.
>
> The symbols `nil`, `t` and keyword symbols may be used on either side. They represent themselves. For example `@(bind :foo :bar)` fails, but `@(bind :foo :foo)` succeeds since the two sides denote the same keyword symbol object.

Example 2:

In this example, suppose `A` contains `"foo"` and `B` contains bar. Then `@(bind (X (Y Z)) (A (B "hey")))` binds `X` to `"foo"`, `Y` to `"bar"` and `Z` to `"hey"`. This is because the *bind-expression* produces the object `("foo" ("bar" "hey"))` which is then structurally matched against the pattern `(X (Y Z))`, and the variables receive the corresponding pieces.

7.3.29 Keywords in the `bind` directive

The `bind` directive accepts these keywords:

`:lfilt`

The argument to `:lfilt` is a filter specification. When the left side pattern contains a binding which is therefore matched against its counterpart from the right side expression, the left side is filtered through the filter specified by `:lfilt` for the purposes of the comparison. For example:

```
@(bind "a" "A" :lfilt :upcase)
```

produces a match, since the left side is the same as the right after filtering through the :upcase filter.

`:rfilt`

The argument to `:rfilt` is a filter specification. The specified filter is applied to the right-hand-side material prior to matching it against the left side. The filter is not applied if the left side is a variable with no binding. It is only applied to determine a match. Binding takes place the unmodified right-hand-side object.

For example, the following produces a match:

```
@(bind "A" "a" :rfilt :upcase)
```

`:filter`

This keyword is a shorthand to specify both filters to the same value. For instance `:filter :upcase` is equivalent to `:lfilt :upcase :rfilt :upcase`.

For a description of filters, see Output Filtering below.

Compound filters like `(:fromhtml :upcase)` are supported with all these keywords. The filters apply across arbitrary patterns and nested data.

Example:

```
@(bind (a b c) ("A" "B" "C"))
@(bind (a b c) (("z" "a") "b" "c") :rfilt :upcase)
```

Here, the first bind establishes the values for `a`, `b` and `c`, and the second bind succeeds, because the value of a matches the second element of the list `("z" "a")` if it is upcased, and likewise `b` matches `"b"` and `c` matches `"c"` if these are upcased.

7.3.30 Lisp forms in the `bind` directive

TXR Lisp forms, introduced by `@` may be used in the *bind-expression* argument of `bind`, or as the entire form. This is consistent with the rules for bind expressions.

TXR Lisp forms can be used in the *pattern* expression also.

Example:

```
@(bind a @(+ 2 2))
@(bind @(+ 2 2) @(* 2 2))
```

Here, a is bound to the integer 4. The second bind then succeeds because the forms (+ 2 2) and (* 2 2) produce equal values.

7.3.31 The set directive

The syntax of the set directive is:

```
@(set pattern bind-expression)
```

The set directive syntactically resembles bind, but is not a pattern match. It overwrites the previous values of variables with new values from the right-hand side. Each variable that is assigned must have an existing binding: set will not induce binding.

Examples follow.

Store the value of A back into A, an operation with no effect:

```
@(set A A)
```

Exchange the values of A and B:

```
@(set (A B) (B A))
```

Store a string into A:

```
@(set A "text")
```

Store a list into A:

```
@(set A ("line1" "line2"))
```

Destructuring assignment. A ends up with "A", B ends up with ("B1" "B2") and C binds to ("C1" "C2").

```
@(bind D ("A" ("B1" "B2") "C1" "C2"))
@(bind (A B C) (() () ()))
@(set (A B . C) D)
```

Note that set does not support a **TXR Lisp** expression on the left side, so the following are invalid syntax:

```
@(set @(+ 1 1) @(* 2 2))
@(set @b @(list "a"))
```

The second one is erroneous even though there is a variable on the left. Because it is preceded by the @ escape, it is a Lisp variable, and not a pattern variable.

The set directive also doesn't support Lisp expressions in the *pattern*, which must consist only of variables.

7.3.32 The `rebind` directive

The syntax of the `rebind` directive is:

```
@(rebind pattern bind-expression)
```

The `rebind` directive resembles `bind`. It combines the semantics of `local` and `bind` into a single directive. The *`bind-expression`* is evaluated in the current environment, and its value remembered. Then a new environment is produced in which all the variables specified in *`pattern`* are absent. Then, the pattern is newly bound in that environment against the previously produced value, as if using `bind`.

The old environment with the previous variables is not modified; it continues to exist. This is in contrast with the `set` directive, which mutates existing bindings.

`rebind` makes it easy to create temporary bindings based on existing bindings.

```
@(define pattern-function (arg))
@;; inside a pattern function:
@(rebind recursion-level @(+ recursion-level 1))
@;; ...
@(end)
```

When the function terminates, the previous value of recursion-level is restored. The effect is less verbose and more efficient than the following equivalent

```
@(define pattern-function (arg))
@;; inside a pattern function:
@(local temp)
@(set temp recursion-level)
@(local recursion-level)
@(set recursion-level @(+ temp 1))
@;; ...
@(end)
```

Like `bind`, `rebind` supports nested patterns, such as

```
@(rebind (a (b c)) (1 (2 3))
```

but it does not support any keyword arguments. The filtering features of `bind` do not make sense in `rebind` because the variables are always reintroduced into an environment in which they don't exist, whereas filtering applies in situations when bound variables are matched against values.

The `rebind` directive also doesn't support Lisp expressions in the *`pattern`*, which must consist only of variables.

7.3.33 The `forget` directive

The `forget` has two spellings: `@(forget)` and `@(local)`.

The arguments are one or more symbols, for example:

```
@(forget a)
@(local a b c)
```

this can be written

```
@(local a)
@(local a b c)
```

Directives which follow the `forget` or `local` directive no longer see any bindings for the symbols mentioned in that directive, and can establish new bindings.

It is not an error if the bindings do not exist.

It is strongly recommended to use the `@(local)` spelling in functions, because the forgetting action simulates local variables: for the given symbols, the machine forgets any earlier variables from outside of the function, and consequently, any new bindings for those variables belong to the function. (Furthermore, functions suppress the propagation of variables that are not in their parameter list, so these locals will be automatically forgotten when the function terminates.)

7.3.34 The `do` directive

The syntax of `@(do)` is:

```
@(do lisp-expression*)
```

The `do` directive evaluates zero or more **TXR Lisp** expressions. (See TXR LISP far below.) The value of the expression is ignored, and matching continues with the directives which follow the `do` directive, if any.

In the context of the `do` directive, the expression should not be introduced by the `@` symbol; it is expected to be a Lisp expression.

Example:

```
@; match text into variables a and b, then insert into hash table h
@(bind h @(hash))
@a:@b
@(do (set [h a] b))
```

7.3.35 The `mdo` directive

The syntax of `@(mdo)` is:

```
@(mdo lisp-expression*)
```

Like the `do` directive, `mdo` (macro-time `do`) evaluates zero or more **TXR Lisp** expressions. Unlike `do`, `mdo` performs this evaluation immediately upon being parsed. Then it disappears from the syntax.

The effect of `@(mdo e0 e1 e2 ...)` is exactly like `@(do (macro-time e0 e1 e2 ...))` except that `do` doesn't disappear from the syntax.

Another difference is that `do` can be used as a horizontal or vertical directive, whereas `mdo` is only vertical.

7.3.36 The `in-package` directive

The `in-package` directive shares the same syntax and semantics as the **TXR Lisp** macro of the same name:

```
(in-package name)
```

The `in-package` directive is evaluated immediately upon being parsed, leaving no trace in the syntax

tree of the surrounding **TXR** query.

It causes the `*package*` special variable to take on the package denoted by *name*.

The directive that *name* is either a string or symbol. An error exception is thrown if this isn't the case. Otherwise it searches for the package. If the package is not found, an error exception is thrown.

7.4 Blocks

7.4.1 Overview

Blocks are sections of a query which are either denoted by a name, or are anonymous. They may nest: blocks can occur within blocks and other constructs.

Blocks are useful for terminating parts of a pattern-matching search prematurely, and escaping to a higher level. This makes blocks not only useful for simplifying the semantics of certain pattern matches, but also an optimization tool.

Judicious use of blocks and escapes can reduce or eliminate the amount of backtracking that **TXR** performs.

7.4.2 The `block` directive

The `@(block name)` directive introduces a named block, except when *name* is the symbol `nil`. The `@(block)` directive introduces an unnamed block, equivalent to `@(block nil)`.

The `@(skip)` and `@(collect)` directives introduce implicit anonymous blocks, as do function bodies.

Blocks must be terminated by `@(end)` and can be vertical:

```
@(block [name])
...
@(end)
```

or horizontal:

```
@(block [name])...@(end)
```

7.4.3 Block Scope

The names of blocks are in a distinct namespace from the variable binding space. So `@(block foo)` is unrelated to the variable `@foo`.

A block extends from the `@(block ...)` directive which introduces it, until the matching `@(end)`, and may be empty. For instance:

```
@(some)
abc
@(block foo)
xyz
@(end)
@(end)
```

Here, the block foo occurs in a `@(some)` clause, and so it extends to the `@(end)` which terminates the block. After that `@(end)`, the name foo is not associated with a block (is not "in scope"). The second

`@(end)` terminates the `@(some)` block.

The implicit anonymous block introduced by `@(skip)` has the same scope as the `@(skip)`: it extends over all of the material which follows the skip, to the end of the containing subquery.

7.4.4 Block Nesting

Blocks may nest, and nested blocks may have the same names as blocks in which they are nested. For instance:

```
@(block)
@(block)
...
@(end)
@(end)
```

is a nesting of two anonymous blocks, and

```
@(block foo)
@(block foo)
@(end)
@(end)
```

is a nesting of two named blocks which happen to have the same name. When a nested block has the same name as an outer block, it creates a block scope in which the outer block is "shadowed"; that is to say, directives which refer to that block name within the nested block refer to the inner block, and not to the outer one.

7.4.5 Block Semantics

A block normally does nothing. The query material in the block is evaluated normally. However, a block serves as a termination point for `@(fail)` and `@(accept)` directives which are in scope of that block and refer to it.

The precise meaning of these directives is:

`@(fail` *`name`*`)`
: Immediately terminate the enclosing query block called *`name`*, as if that block failed to match anything. If more than one block by that name encloses the directive, the innermost block is terminated. No bindings emerge from a failed block.

`@(fail)`
: Immediately terminate the innermost enclosing anonymous block, as if that block failed to match.

  The `@(fail)` directive has a vertical and horizontal form.

  If the implicit block introduced by `@(skip)` is terminated in this manner, this has the effect of causing `skip` itself to fail. In other words, the behavior is as if `@(skip)`'s search did not find a match for the trailing material, except that it takes place prematurely (before the end of the available data source is reached).

  If the implicit block associated with a `@(collect)` is terminated this way, then the entire `collect` fails. This is a special behavior, because a `collect` normally does not fail, even if it matches nothing and collects nothing!

To prematurely terminate a `collect` by means of its anonymous block, without failing it, use `@(accept)`.

`@(accept` *name*`)`
: Immediately terminate the enclosing query block called *name*, as if that block successfully matched. If more than one block by that name encloses the directive, the innermost block is terminated.

`@(accept)`
: Immediately terminate the innermost enclosing anonymous block, as if that block successfully matched.

`@(accept)` communicates the current bindings and input position to the terminated block. These bindings and current position may be altered by special interactions between certain directives and `@(accept)`, described in the following section. Communicating the current bindings and input position means that the block which is terminated by `@(accept)` exhibits the bindings which were collected just prior to the execution of that `@(accept)` and the input position which was in effect at that time.

`@(accept)` has a vertical and horizontal form. In the horizontal form, it communicates a horizontal input position. A horizontal input position thus communicated will only take effect if the block being terminated had been suspended on the same line of input.

If the implicit block introduced by `@(skip)` is terminated by `@(accept)`, this has the effect of causing the skip itself to succeed, as if all of the trailing material had successfully matched.

If the implicit block associated with a `@(collect)` is terminated by `@(accept)`, then the collection stops. All bindings collected in the current iteration of the collect are discarded. Bindings collected in previous iterations are retained, and collated into lists in accordance with the semantics of collect.

Example: alternative way to achieve `@(until)` termination:

```
@(collect)
@  (maybe)
---
@  (accept)
@  (end)
@LINE
@(end)
```

This query will collect entire lines into a list called `LINE`. However, if the line `---` is matched (by the embedded `@(maybe)`), the collection is terminated. Only the lines up to, and not including the `---` line, are collected. The effect is identical to:

```
@(collect)
@LINE
@(until)
---
@(end)
```

The difference (not relevant in these examples) is that the until clause has visibility into the bindings set up by the main clause.

However, the following example has a different meaning:

```
@(collect)
@LINE
@  (maybe)
---
@  (accept)
@  (end)
@(end)
```

Now, lines are collected until the end of the data source, or until a line is found which is followed by a `---` line. If such a line is found, the collection stops, and that line is not included in the collection! The `@(accept)` terminates the process of the collect body, and so the action of collecting the last `@LINE` binding into the list is not performed.

Example: communication of bindings and input position:

code:
```
@(some)
@(block foo)
@first
@(accept foo)
@ignored
@(end)
@second
```

data:
```
1
2
3
```

result:
```
first="1"
second="2"
```

At the point where the `accept` occurs, the foo block has matched the first line, bound the text `"1"` to the variable `@first`. The block is then terminated. Not only does the `@first` binding emerge from this terminated block, but what also emerges is that the block advanced the data past the first line to the second line. Next, the `@(some)` directive ends, and propagates the bindings and position. Thus the `@second` which follows then matches the second line and takes the text `"2"`.

Example: abandonment of `@(some)` clause by `@(accept)`:

In the following query, the foo block occurs inside a maybe clause. Inside the foo block there is a `@(some)` clause. Its first subclause matches variable `@first` and then terminates block foo. Since block foo is outside of the `@(some)` directive, this has the effect of terminating the `@(some)` clause:

code:
```
@(maybe)
@(block foo)
@  (some)
@first
@  (accept foo)
@  (or)
@one
@two
@three
@four
@  (end)
@(end)
@second
```

data:
```
1
2
3
4
5
```

result:
```
first="1"
second="2"
```

The second clause of the `@(some)` directive, namely:

```
@one
@two
@three
@four
```

is never processed. The reason is that subclauses are processed in top to bottom order, but the processing was aborted within the first clause the `@(accept foo)`. The `@(some)` construct never gets the opportunity to match four lines.

If the `@(accept foo)` line is removed from the above query, the output is different:

code:
```
@(maybe)
@(block foo)
@  (some)
@first
@#          <--  @(accept foo) removed from here!!!
@  (or)
@one
@two
@three
@four
@  (end)
@(end)
@second
```

data:
```
1
2
3
4
5
```

result:
```
first="1"
one="1"
two="2"
three="3"
four="4"
second="5"
```

Now, all clauses of the `@(some)` directive have the opportunity to match. The second clause grabs four lines, which is the longest match. And so, the next line of input available for matching is `5`, which goes to the `@second` variable.

7.4.6 Interaction Between the `trailer` and `accept` Directives

If one of the clauses which follow a `@(trailer)` requests a successful termination to an outer block via `@(accept)`, then `@(trailer)` intercepts the escape and adjusts the data extent to the position that it was given.

Example:

code:
```
@(block)
@(trailer)
@line1
@line2
@(accept)
@(end)
@line3
```

data:
```
1
2
3
```

result:
```
line1="1"
line2="2"
line3="1"
```

The variable `line3` is bound to `"1"` because although `@(accept)` yields a data position which has advanced to the third line, this is intercepted by `@(trailer)` and adjusted back to the first line. Neglecting to do this adjustment would violate the semantics of `trailer`.

7.4.7 Interaction Between the `next` and `accept` Directives

When the clauses under a `next` directive are terminated by an `accept`, such that control passes to a block which surrounds that `next`, the `accept` is intercepted by `next`.

The input position being communicated by the `accept` is replaced with the original input position in the original stream which is in effect prior to the `next` directive. The `accept` transfer is then resumed.

In other words, `accept` cannot be used to "leak" the new stream out of a `next` scope.

However, `next` has no effect on the bindings being communicated.

Example:

```
@(next "file-x")
@(block b)
@(next "file-y")
@line
@(accept b)
@(end)
```

Here, the variable `line` matches the first line of the file `"file-y"`, after which an `accept` transfer is initiated, targeting block `b`. This transfer communicates the `line` binding, as well as the position within `file-y`, pointing at the second line. However, the `accept` traverses the `next` directive, causing it to be abandoned. The special unwinding action within that directive detects this transfer and rewrites the input position to be the original one within the stream associated with `"file-x"`. Note that this special handling exists in order for the behavior to be consistent with what would happen if the `@(accept b)` were removed, and the block `b` terminated normally: because the inner `next` is nested within that block, **TXR** would backtrack to the previous input position within `"file-x"`.

7.4.8 Interaction Between Functions and the `accept` directive

If a pattern function is terminated due to `accept`, the function return mechanism intercepts the `accept`. The bindings being communicated by that `accept` are then subject to the special resolution with respect to the function parameters, exactly as if the bindings were being returned normally out of the function. The

resolved bindings then replace those being communicated by the `accept` and the `accept` transfer is resumed.

Example:

```
@(define fun (a))
@  (bind a "a")
@  (bind b "b")
@  (accept blk)
@(end)
@(block blk)
@(fun x)
this line is skipped by accept
@(end)
```

Here, the `accept` initiates a control transfer which communicates the `a` and `b` variable bindings which are visible in that scope. This transfer is intercepted by the function, and the treatment of the bindings follows to the same rules as a normal return (which, in the given function, would readily take place if the `accept` directive were removed). The `b` variable is suppressed, because `b` isn't a parameter of the function. Because `a` **is** a parameter, and the argument to that parameter is the unbound variable `x`, the effect is that `x` is bound to the value of `a`. When the accept transfer reaches block `blk` and terminates it, all that emerges is the `x` binding carrying `"a"`.

If the `accept` invocation is removed from `fun`, then the function returns normally, producing the `x` binding. In that case, the line `this line is skipped by accept` isn't skipped since the block isn't being terminated; that line must match something.

7.4.9 Interaction Between `finally` and the `accept` directive

If the exception handling `try` directive protected body is terminated by an `accept` transfer, and if that `try` has a `finally` block, then there is a special interaction between the `finally` block and the `accept` transfer.

The processing of the `finally` block detects that it has been triggered by an `accept` transfer. Consequently, it retrieves the current input position and bindings from that transfer, and uses that position and those bindings for the processing of the `finally` clauses.

If the `finally` clauses succeed, then the new input position and new bindings are installed into the `accept` control transfer and that transfer resumes.

If the `finally` clauses fail, then the `accept` transfer is converted to a `fail`, with exactly the same block as its destination.

7.4.10 Vertical-Horizontal Mismatch Between `block` and `accept`

The `block`, `accept` and `fail` directives comes in horizontal and vertical forms.

This creates the possibility that an `accept` in horizontal context targets a vertical `block` or vice versa, raising the question of how the input position is treated. The semantics of this is defined.

If a horizontal-context `accept` targets a vertical block, the current position at the target block will be the following line. That is to say, when the horizontal `accept` occurs, there is a current input line which may have unconsumed material past the current position. If the `accept` communicates its input position to a vertical context, that unconsumed material is skipped, as if it had been matched and the vertical position is advanced to the next line.

If a horizontal block catches a vertical accept, it rejects that `accept`'s position and stays at the current backtracking position for that block. Only the bindings from the `accept` are retained.

7.4.11 Horizontal-Horizontal Mismatch between `block` and `accept`

It is possible for a horizontal `accept` to terminate in a horizontal block which is processing a different line of input (or even a different input stream). This situation is treated the same way as vertical accept terminating in a horizontal block: the position communicated by `accept` is ignored, and only the bindings are taken.

7.5 Functions

7.5.1 Overview

TXR functions allow a query to be structured to avoid repetition. On a theoretical note, because **TXR** functions support recursion, functions enable **TXR** to match some kinds of patterns which exhibit self-embedding, or nesting, and thus cannot be matched by a regular language.

Functions in **TXR** are not exactly like functions in mathematics or functional languages, and are not like procedures in imperative programming languages. They are not exactly like macros either. What it means for a **TXR** function to take arguments and produce a result is different from the conventional notion of a function.

A **TXR** function may have one or more parameters. When such a function is invoked, an argument must be specified for each parameter. However, a special behavior is at play here. Namely, some or all of the argument expressions may be unbound variables. In that case, the corresponding parameters behave like unbound variables also. Thus **TXR** function calls can transmit the "unbound" state from argument to parameter.

It should be mentioned that functions have access to all bindings that are visible in the caller; functions may refer to variables which are not mentioned in their parameter list.

With regard to returning, **TXR** functions are also unconventional. If the function fails, then the function call is considered to have failed. The function call behaves like a kind of match; if the function fails, then the call is like a failed match.

When a function call succeeds, then the bindings emanating from that function are processed specially. Firstly, any bindings for variables which do not correspond to one of the function's parameters are thrown away. Functions may internally bind arbitrary variables in order to get their job done, but only those variables which are named in the function argument list may propagate out of the function call. Thus, a function with no arguments can only indicate matching success or failure, but not produce any bindings. Secondly, variables do not propagate out of the function directly, but undergo a renaming. For each parameter which went into the function as an unbound variable (because its corresponding argument was an unbound variable), if that parameter now has a value, that value is bound onto the corresponding argument.

Example:

```
@(define collect-words (list))
@(coll)@{list /[^ \t]+/}@(end)
@(end)
```

The above function `collect-words` contains a query which collects words from a line (sequences of characters other than space or tab), into the list variable called `list`. This variable is named in the parameter list of the function, therefore, its value, if it has one, is permitted to escape from the function call.

Suppose the input data is:

```
Fine summer day
```

and the function is called like this:

```
@(collect-words wordlist)
```

The result (with `txr -B`) is:

```
wordlist[0]=Fine
wordlist[1]=summer
wordlist[1]=day
```

How it works is that in the function call `@(collect-words wordlist)`, `wordlist` is an unbound variable. The parameter corresponding to that unbound variable is the parameter `list`. Therefore, that parameter is unbound over the body of the function. The function body collects the words of `"Fine summer day"` into the variable `list`, and then yields the that binding. Then the function call completes by noticing that the function parameter `list` now has a binding, and that the corresponding argument `wordlist` has no binding. The binding is thus transferred to the `wordlist` variable. After that, the bindings produced by the function are thrown away. The only enduring effects are:

- the function matched and consumed some input; and
- the function succeeded; and
- the `wordlist` variable now has a binding.

Another way to understand the parameter behavior is that function parameters behave like proxies which represent their arguments. If an argument is an established value, such as a character string or bound variable, the parameter is a proxy for that value and behaves just like that value. If an argument is an unbound variable, the function parameter acts as a proxy representing that unbound variable. The effect of binding the proxy is that the variable becomes bound, an effect which is settled when the function goes out of scope.

Within the function, both the original variable and the proxy are visible simultaneously, and are independent. What if a function binds both of them? Suppose a function has a parameter called `P`, which is called with an argument `A`, which is an unbound variable, and then, in the function, both `A` and `P` bound. This is permitted, and they can even be bound to different values. However, when the function terminates, the local binding of A simply disappears (because the symbol `A` is not among the parameters of the function). Only the value bound to `P` emerges, and is bound to `A`, which still appears unbound at that point. The `P` binding disappears also, and the net effect is that `A` is now bound. The "proxy" binding of `A` through the parameter `P` "wins" the conflict with the direct binding.

7.5.2 Definition Syntax

Function definition syntax comes in two flavors: vertical and horizontal. Horizontal definitions actually come in two forms, the distinction between which is hardly noticeable, and the need for which is made clear below.

A function definition begins with a `@(define ...)` directive. For vertical functions, this is the only element in a line.

The `define` symbol must be followed by a symbol, which is the name of the function being defined. After the symbol, there is a parenthesized optional argument list. If there is no such list, or if the list is specified as `()` or the symbol `nil` then the function has no parameters. Examples of valid `define` syntax are:

```
@(define foo)
@(define bar ())
@(define match (a b c))
```

If the `define` directive is followed by more material on the same line, then it defines a horizontal function:

```
@(define match-x)x@(end)
```

If the define is the sole element in a line, then it is a vertical function, and the function definition continues below:

```
@(define match-x)
x
@(end)
```

The difference between the two is that a horizontal function matches characters within a line, whereas a vertical function matches lines within a stream. The former `match-x` matches the character `x`, advancing to the next character position. The latter `match-x` matches a line consisting of the character `x`, advancing to the next line.

Material between `@(define)` and `@(end)` is the function body. The define directive may be followed directly by the `@(end)` directive, in which case the function has an empty body.

Functions may be nested within function bodies. Such local functions have dynamic scope. They are visible in the function body in which they are defined, and in any functions invoked from that body.

The body of a function is an anonymous block. (See Blocks above.)

7.5.3 Two Forms of The Horizontal Function

If a horizontal function is defined as the only element of a line, it may not be followed by additional material. The following construct is erroneous:

```
@(define horiz (x))@foo:@bar@(end)lalala
```

This kind of definition is actually considered to be in the vertical context, and like other directives that have special effects and that do not match anything, it does not consume a line of input. If the above syntax were allowed, it would mean that the line would not only define a function but also match `lalala`. This would, in turn, would mean that the `@(define)...@(end)` is actually in horizontal mode, and so it matches a span of zero characters within a line (which means that is would require a line of input to match: a surprising behavior for a nonmatching directive!)

A horizontal function can be defined in an actual horizontal context. This occurs if its is in a line where it is preceded by other material. For instance:

```
X@(define fun)...@(end)Y
```

This is a query line which must match the text `XY`. It also defines the function `fun`. The main use of this form is for nested horizontal functions:

```
@(define fun)@(define local_fun)...@(end)@(end)
```

7.5.4 Vertical-Horizontal Overloading

A function of the same name may be defined as both vertical and horizontal. Both functions are available at the same time. Which one is used by a call is resolved by context. See the section Vertical Versus Horizontal Calls below.

7.5.5 Call Syntax

A function is invoked by compound directive whose first symbol is the name of that function. Additional elements in the directive are the arguments. Arguments may be symbols, or other objects like string and character literals, quasiliterals ore regular expressions.

Example:

code:
```
@(define pair (a b))
@a @b
@(end)
@(pair first second)
@(pair "ice" cream)
```

data:
```
one two
ice milk
```

result:
```
first="one"
second="two"
cream="milk"
```

The first call to the function takes the line `"one two"`. The parameter `a` takes `"one"` and parameter `b` takes `"two"`. These are rebound to the arguments `first` and `second`. The second call to the function binds the a parameter to the word `"ice"`, and the `b` is unbound, because the corresponding argument `cream` is unbound. Thus inside the function, `a` is forced to match `ice`. Then a space is matched and `b` collects the text `"milk"`. When the function returns, the unbound `"cream"` variable gets this value.

If a symbol occurs multiple times in the argument list, it constrains both parameters to bind to the same value. That is to say, all parameters which, in the body of the function, bind a value, and which are all derived from the same argument symbol must bind to the same value. This is settled when the function terminates, not while it is matching. Example:

code:
```
@(define pair (a b))
@a @b
@(end)
@(pair same same)
```

data:
```
one two
```

result:
```
[query fails]
```

Here the query fails because `a` and `b` are effectively proxies for the same unbound variable `same` and are bound to different values, creating a conflict which constitutes a match failure.

7.5.6 Vertical Versus Horizontal Calls

A function call which is the only element of the query line in which it occurs is ambiguous. It can go either to a vertical function or to the horizontal one. If both are defined, then it goes to the vertical one.

Example:

code:
```
@(define which (x))@(bind x "horizontal")@(end)
@(define which (x))
@(bind x "vertical")
@(end)
```

```
@(which fun)
```

result: `fun="vertical"`

Not only does this call go to the vertical function, but it is in a vertical context.

If only a horizontal function is defined, then that is the one which is called, even if the call is the only element in the line. This takes place in a horizontal character-matching context, which requires a line of input which can be traversed:

Example:

code:
```
@(define which (x))@(bind x "horizontal")@(end)
@(which fun)
```

data: `ABC`

result: `[query fails]`

The query fails because since `@(which fun)` is in horizontal mode, it matches characters in a line. Since the function body consists only of `@(bind ...)` which doesn't match any characters, the function call requires an empty line to match. The line `ABC` is not empty, and so there is a matching failure. The following example corrects this:

Example:

code:
```
@(define which (x))@(bind x "horizontal")@(end)
@(which fun)
```

data: `[empty line]`

result: `fun="horizontal"`

A call made in a clearly horizontal context will prefer the horizontal function, and only fall back on the vertical one if the horizontal one doesn't exist. (In this fallback case, the vertical function is called with empty data; it is useful for calling vertical functions which process arguments and produce values.)

In the next example, the call is followed by trailing material, placing it in a horizontal context. Leading material will do the same thing:

Example:

code:
```
@(define which (x))@(bind x "horizontal")@(end)
@(define which (x))
@(bind x "vertical")
@(end)
@(which fun)B
```

data: `B`

result: `fun="horizontal"`

7.5.7 Local Variables

As described earlier, variables bound in a function body which are not parameters of the function are discarded when the function returns. However, that, by itself, doesn't make these variables local, because pattern functions have visibility to all variables in their calling environment. If a variable `x` exists already when a function is called, then an attempt to bind it inside a function may result in a failure. The `local` directive must be used in a pattern function to list which variables are local.

Example:

```
@(define path (path))@\
```

```
    @(local x y)@\
    @(cases)@\
      (@(path x))@(path y)@(bind path `(@x)@y`)@\
    @(or)@\
      @{x /[.,;'!?][^ \t\f\v]/}@(path y)@(bind path `@x@y`)@\
    @(or)@\
      @{x /[^ .,;'!?()\t\f\v]/}@(path y)@(bind path `@x@y`)@\
    @(or)@\
      @(bind path "")@\
    @(end)@\
  @(end)
```

This is a horizontal function which matches a path, which lands into four recursive cases. A path can be parenthesized path followed by a path; it can be a certain character followed by a path, or it can be empty

This function ensures that the variables it uses internally, `x` and `y`, do not have anything to do with any inherited bindings for `x` and `y`.

Note that the function is recursive, which cannot work without `x` and `y` being local, even if no such bindings exist prior to the top-level invocation of the function. The invocation `@(path x)` causes `x` to be bound, which is visible inside the invocation `@(path y)`, but that invocation needs to have its own binding of `x` for local use.

7.5.8 Nested Functions

Function definitions may appear in a function. Such definitions are visible in all functions which are invoked from the body (and not necessarily enclosed in the body). In other words, the scope is dynamic, not lexical. Inner definitions shadow outer definitions. This means that a caller can redirect the function calls that take place in a callee, by defining local functions which capture the references.

Example:

code:
```
@(define which)
@  (fun)
@(end)
@(define fun)
@  (output)
top-level fun!
@  (end)
@(end)
@(define callee)
@  (define fun)
@    (output)
local fun!
@    (end)
@  (end)
@  (which)
@(end)
@(callee)
@(which)
```

output:
```
local fun!
top-level fun!
```

Here, the function `which` is defined which calls `fun`. A top-level definition of `fun` is introduced which outputs `"top-level fun!"`. The function `callee` provides its own local definition of `fun` which

outputs `"local fun!"` before calling `which`. When `callee` is invoked, it calls `which`, whose `@(fun)` call is routed to callee's local definition. When `which` is called directly from the top level, its `fun` call goes to the top-level definition.

7.5.9 Indirect Calls

Function indirection may be performed using the `call` directive. If *`fun-expr`* is an Lisp expression which evaluates to a symbol, and that symbol names a function which takes no arguments, then

```
@(call fun-expr)
```

may be used to invoke the function. Additional expressions may be supplied which specify arguments.

Example 1:

```
@(define foo (arg))
 @(bind arg "abc")
 @(end)
 @(call 'foo b)
```

In this example, the effect is that `foo` is invoked, and `b` ends up bound to `"abc"`.

The `call` directive here uses the `'foo` expression to calculate the name of the function to be invoked. (See the `quote` operator).

This particular `call` expression can just be replaced by the direct invocation syntax `@(foo b)`.

The power of `call` lies in being able to specify the function as a value which comes from elsewhere in the program, as in the following example.

```
@(define foo (arg))
 @(bind arg "abc")
 @(end)
 @(bind f @'foo)
 @(call f b)
```

Here the `call` directive obtains the name of the function from the `f` variable.

Note that function names are resolved to functions in the environment that is apparent at the point in execution where the `call` takes place. The directive `@(call f args ...)` is precisely equivalent to `@(s args ...)` if, at the point of the call, `f` is a variable which holds the symbol `s` and symbol `s` is defined as a function. Otherwise it is erroneous.

7.6 Modularization

7.6.1 The `load` and `include` directives

The syntax of the `load` and `include` directives is:

```
@(load expr)
@(include expr)
```

Where *`expr`* is a Lisp expression that evaluates to a string giving the path of the file to load.

Firstly, the path given by *`expr`* is converted to an effective path, as follows.

If the value of the `*load-path*` variable has a current value which is not `nil` and the path given in

expr is pure relative according to the `pure-rel-path-p` function, then the effective path is interpreted taken relative to the directory portion of the path which is stored in `*load-path*`.

If `*load-path*` is `nil`, or the load path is not pure relative, then the path is taken as-is as the effective path.

Next, an attempt is made to open the file for processing, in almost exactly the same manner as by the **TXR Lisp** function `load`. The difference is that if the effective path is unsuffixed, then the `.txr` suffix is added to it, and that resulting path is tried first, and if it succeeds, then the file is treated as **TXR** Pattern Language syntax. If that fails, then the suffix `.tlo` is tried, and so forth, as described for the `load` function.

If these initial attempts to find the file fail, and the failure is due to the file not being found rather than some other problem such as a permission error, and *expr* isn't an absolute path according to `abs-path-p`, then additional attempts are made by searching for the file in the list of directories given in the `*load-search-dirs*` variable. Details are given in the description of the **TXR Lisp** `load` function.

Both the `load` and `include` directives bind the `*load-path*` variable to the path of the loaded file just before parsing syntax from it, The `*package*` variable is also given a new dynamic binding, whose value is the same as the existing binding. These bindings are removed when the load operation completes, restoring the prior values of these variables. The `*load-hooks*` variable is given a new dynamic binding, with a `nil` value.

If the file opened for processing is **TXR Lisp** source, or a compiled **TXR Lisp** file, then it is processed in the manner described for the `load` function.

Different requirements apply to the processing of the file under the `load` and `include` directives.

The `include` directive performs the processing of the file at parse time. If the file being processed is **TXR** Pattern Language, then it is parsed, and then its syntax replaces the `include` directive, as if it had originally appeared in its place. If a **TXR Lisp** source or a compiled **TXR Lisp** file is processed by `include` then the `include` directive is removed from the syntax.

The `load` directive performs the processing of the file at evaluation time. Evaluation time occurs after a **TXR** program is read from beginning to end and parsed. That is to say, when a **TXR** query is parsed, any embedded `@(load ...)` forms in it are parsed and constitute part of its syntax tree. They are executed when that query is executed, whenever its execution reaches those `load` directives. When the `load` directive processes **TXR** Pattern Language syntax, it parses the file in its entirety and then executes that file's directives against the current input position. Repeated executions of the same `load` directive result in repeated processing of the file.

Note: the `include` directive is useful for loading **TXR** files which contain Lisp macros which are needed by the parent program. The parent program cannot use `load` to bring in macros because macros are required during expansion, which takes place prior to evaluation time, whereas `load` doesn't execute until evaluation time.

See also: the `self-path`, `stdlib` and `*load-path*` variables in **TXR Lisp**.

7.7 Output

7.7.1 Introduction

A **TXR** query may perform custom output. Output is performed by `output` clauses, which may be embedded anywhere in the query, or placed at the end. Output occurs as a side effect of producing a part of a query which contains an `@(output)` directive, and is executed even if that part of the query ultimately fails to find a match. Thus output can be useful for debugging. An `output` clause specifies that its output

goes to a file, pipe, or (by default) standard output. If any output clause is executed whose destination is standard output, **TXR** makes a note of this, and later, just prior to termination, suppresses the usual printing of the variable bindings or the word false.

7.7.2 The `output` directive

The syntax of the `@(output)` directive is:

```
@(output [ destination ] { bool-keyword | keyword value }* )
.
. one or more output directives or lines
.
@(end)
```

If the directive has arguments, then the first one is evaluated. If it is an object other than a keyword symbol, then it specifies the optional *destination*. Any remaining arguments after the optional destination are the keyword list. If the destination is missing, then the entire argument list is a keyword list.

The *destination* argument, if present, is treated as a **TXR Lisp** expression and evaluated. The resulting value is taken as the output destination. The value may be a string which gives the pathname of a file to open for output. Otherwise, the destination must be a stream object.

The keyword list consists of a mixture of Boolean keywords which do not have an argument, or keywords with arguments.

The following Boolean keywords are supported:

`:nothrow`
: The `output` directive throws an exception if the output destination cannot be opened, unless the `:nothrow` keyword is present, in which case the situation is treated as a match failure.

  Note that since command pipes are processes that report errors asynchronously, a failing command will not throw an immediate exception that can be suppressed with `:nothrow`. This is for synchronous errors, like trying to open a destination file, but not having permissions, etc.

`:append`
: This keyword is meaningful for files, specifying append mode: the output is to be added to the end of the file rather than overwriting the file.

The following value keywords are supported:

`:filter`
: The argument can be a symbol, which specifies a filter to be applied to the variable substitutions occurring within the `output` clause. The argument can also be a list of filter symbols, which specifies that multiple filters are to be applied, in left-to-right order.

  See the later sections Output Filtering below, and The Deffilter Directive.

`:into` The argument of `:into` is a symbol which denotes a variable. The output will go into that variable. If the variable is unbound, it will be created. Otherwise, its contents are overwritten unless the `:append` keyword is used. If `:append` is used, then the new content will be appended to the previous content of the variable, after flattening the content to a list, as if by the `flatten` directive.

`:named`
: The argument of `:named` is a symbol which denotes a variable. The file or pipe stream which is opened for the output is stored in this variable, and is not closed at the end of the output block. This allows a subsequent output block to continue output on the same stream, which is possible using the next two keywords, `:continue` or `:finish`. A new binding is established for the variable, even if it already has an existing binding.

`:continue`
: A destination should not be specified if `:continue` is used. The argument of `:continue` is an expression, such as a variable name, that evaluates to a stream object. That stream object is used for the output block. At the end of the output block, the stream is flushed, but not closed. A usage example is given in the documentation for the Close Directive below.

`:finish`
: A destination should not be specified if `:finish` is used. The argument of `:finish` is an expression, such as a variable name, that evaluates to a stream object. That stream object is used for the output block. At the end of the output block, the stream is closed. An example is given in the documentation for the Close Directive below.

7.7.3 Output Text

Text in an output clause is not matched against anything, but is output verbatim to the destination file, device or command pipe.

7.7.4 Output Variables

Variables occurring in an output clause do not match anything; instead their contents are output.

A variable being output can be any object. If it is of a type other than a list or string, it will be converted to a string as if by the `tostring` function in **TXR Lisp**.

A list is converted to a string in a special way: the elements are individually converted to a string and then they are catenated together. The default separator string is a single space: an alternate separation can be specified as an argument in the brace substitution syntax. Empty lists turn into an empty string.

Lists may be output within `@(repeat)` or `@(rep)` clauses. Each nesting of these constructs removes one level of nesting from the list variables that it contains.

In an output clause, the `@{`*`name number`*`}` variable syntax generates fixed-width field, which contains the variable's text. The absolute value of the number specifies the field width. For instance `-20` and `20` both specify a field width of twenty. If the text is longer than the field, then it overflows the field. If the text is shorter than the field, then it is left-adjusted within that field, if the width is specified as a positive number, and right-adjusted if the width is specified as negative.

An output variable may specify a filter which overrides any filter established for the output clause. The syntax for this is `@{NAME :filter`*`filterspec`*`}`. The filter specification syntax is the same as in the output clause. See Output Filtering below.

7.7.5 Output Variables: Indexing

Additional syntax is supported in output variables that does not appear in pattern-matching variables.

A square bracket index notation may be used to extract elements or ranges from a variable, which works with strings, vectors and lists. Elements are indexed from zero. This notation is only available in brace-

enclosed syntax, and looks like this:

`@{name[expr]}`
: Extract the element at the position given by *expr*.

`@{name[expr1..expr2]}`
: Extract a range of elements from the position given by *expr1*, up to one position less than the position given by *expr2*.

  If the variable is a list, it is treated as a list substitution, exactly as if it were the value of an unsubscripted list variable. The elements of the list are converted to strings and catenated together with a separator string between them, the default one being a single space.

  An alternate character may be given as a string argument in the brace notation.

Example:

```
@(bind a ("a" "b" "c" "d"))
@(output)
@{a[1..3] "," 10}
@(end)
```

The above produces the text `"b,c"` in a field `10` spaces wide. The `[1..3]` argument extracts a range of `a`; the `","` argument specifies an alternate separator string, and `10` specifies the field width.

7.7.6 Output Substitutions

The brace syntax has another syntactic and semantic extension in `output` clauses. In place of the symbol, an expression may appear. The value of that expression is substituted.

Example:

```
@(bind a "foo")
@(output)
@{`@a:` -10}
```

Here, the quasiliteral expression `` `@a:` `` is evaluated, producing the string `"foo:"`. This string is printed right-adjusted in a `10` character field.

7.7.7 The `repeat` directive

The `repeat` directive generates repeated text from a "boilerplate", by taking successive elements from lists. The syntax of repeat is like this:

```
@(repeat)
.
.
main clause material, required
.
.
special clauses, optional
.
.
@(end)
```

`repeat` has four types of special clauses, any of which may be specified with empty contents, or omitted entirely. They are described below.

`repeat` takes arguments, also described below.

All of the material in the main clause and optional clauses is examined for the presence of variables. If none of the variables hold lists which contain at least one item, then no output is performed, (unless the repeat specifies an `@(empty)` clause, see below). Otherwise, among those variables which contain nonempty lists, repeat finds the length of the longest list. This length of this list determines the number of repetitions, R.

If the `repeat` contains only a main clause, then the lines of this clause is output R times. Over the first repetition, all of the variables which, outside of the repeat, contain lists are locally rebound to just their first item. Over the second repetition, all of the list variables are bound to their second item, and so forth. Any variables which hold shorter lists than the longest list eventually end up with empty values over some repetitions.

Example: if the list `A` holds `"1"`, `"2"` and `"3"`; the list `B` holds `"A"`, `"B"`; and the variable `C` holds `"X"`, then

```
@(repeat)
>> @C
>> @A @B
@(end)
```

will produce three repetitions (since there are two lists, the longest of which has three items). The output is:

```
>> X
>> 1 A
>> X
>> 2 B
>> X
>> 3
```

The last line has a trailing space, since it is produced by `"@A @B"`, where `B` has an empty value. Since `C` is not a list variable, it produces the same value in each repetition.

The special clauses are:

`@(single)`
: If the `repeat` produces exactly one repetition, then the contents of this clause are processed for that one and only repetition, instead of the main clause or any other clause which would otherwise be processed.

`@(first)`
: The body of this clause specifies an alternative body to be used for the first repetition, instead of the material from the main clause.

`@(last)`
: The body of this clause is used instead of the main clause for the last repetition.

`@(empty)`
: If the repeat produces no repetitions, then the body of this clause is output. If this clause is absent or empty, the repeat produces no output.

`@(mod n m)`
: The forms `n` and `m` are Lisp expressions that evaluate to integers. The value of `m` should be nonzero. The clause denoted this way is active if the repetition modulo `m` is equal to `n`. The first repetition is numbered zero. For instance the clause headed by `@(mod 0 2)` will be used on repetitions 0, 2, 4, 6, ... and `@(mod 1 2)` will be used on repetitions 1, 3, 5, 7, ...

`@(modlast n m)`
: The meaning of `n` and `m` is the same as in `@(mod n m)`, but one more condition is imposed. This clause is used if the repetition modulo `m` is equal to `n`, and if it is the last repetition.

The precedence among the clauses which take an iteration is: `single > first > modlast > last > mod > main`. That is, whenever two or more of these clauses can apply to a repetition, then the leftmost one in this precedence list will be selected. It is possible for all these clauses to be viable for processing the same repetition. If a `repeat` occurs which has only one repetition, then that repetition is simultaneously the first, only and last repetition. Moreover, it also matches `(mod 0 m)` and, because it is the last repetition, it matches `(modlast 0 m)`. In this situation, if there is a `@(single)` clause present, then the repetition shall be processed using that clause. Otherwise, if there is a `@(first)` clause present, that clause is activated. Failing that, `@(modlast)` is used if there is such a clause, featuring an `n` argument of zero. If there isn't, then the `@(last)` clause is considered, if present. Otherwise, the `@(mod)` clause is considered if present with an `n` argument of zero. Otherwise, none of these clauses are present or applicable, and the repetition is processed using the main clause.

The `@(empty)` clause does not appear in the above precedence list because it is mutually exclusive with respect to the others: it is processed only when there are no iterations, in which case even the main clause isn't active.

The `@(repeat)` clause supports arguments.

```
@(repeat
    [:counter {symbol | (symbol expr)}]
    [:vars ({symbol | (symbol expr)}*)])
```

The `:counter` argument designates a symbol which will behave as an integer variable over the scope of the clauses inside the repeat. The variable provides access to the repetition count, starting at zero, incrementing with each repetition. If the argument is given as `(`*`symbol expr`*`)` then *`expr`* is a Lisp expression whose value is taken as a displacement value which is added to each iteration of the counter. For instance `:counter (c 1)` specifies a counter `c` which counts from 1.

The `:vars` argument specifies a list of variable name symbols *`symbol`* or else pairs of the form `(`*`symbol init-form`*`)` consisting of a variable name and Lisp expression. Historically, the former syntax informed `repeat` about references to variables contained in Lisp code. This usage is no longer necessary as of **TXR** 243, since the `repeat` construct walks Lisp code, identifying all free variables. The latter syntax introduces a new pattern variable binding for *`symbol`* over the scope of the `repeat` construct. The *`init-form`* specifies a Lisp expression which is evaluated to produce the binding's value.

The `repeat` directive then processes the list of variables, selecting from it those which have a binding, either a previously existing binding or the one just introduced. For each selected variable, repeat will assume that the variable occurs in the repeat block and contains a list to be iterated.

The variable binding syntax supported by `:vars` of the form `(`*`symbol init-form`*`)` provides a

solution for situations when it is necessary to iterate over some list, but that list is the result of an expression, and not stored in any variable. A repeat block iterates only over lists emanating from variables; it does not iterate over lists pulled from arbitrary expressions.

Example: output all file names matching the `*.txr` pattern in the current directory:

```
@(output)
@(repeat :vars ((name (glob "*.txr"))))
@name
@(end)
@(end)
```

Prior to **TXR** 243, the simple variable-binding syntax supported by `:vars` of the form *`symbol`* was needed for situations in which **TXR Lisp** expressions which referenced variables were embedded in `@(repeat)` blocks. Variable references embedded in Lisp code were not identified in `@(repeat)`. For instance, the following produced no output, because no variables were found in the `repeat` body:

```
@(bind trigraph ("abc" "def" "ghi"))
@(output)
@(repeat)
@(reverse trigraph)
@(end)
@(end)
```

There is a reference to *`trigraph`* but it's inside the `(reverse trigraph)` Lisp expression that was not processed by `repeat`. The solution was to mention *`trigraph`* in the `:vars` construct:

```
@(bind trigraph ("abc" "def" "ghi"))
@(output)
@(repeat :vars (trigraph))
@(reverse trigraph)
@(end)
@(end)
```

Then the `repeat` block would iterate over *`trigraph`*, producing the output

```
cba
fed
igh
```

This workaround is no longer required as of **TXR** 243; the output is produced by the first example, without `:vars`.

7.7.8 Nested `repeat` directives

If a `repeat` clause encloses variables which hold multidimensional lists, those lists require additional nesting levels of `repeat` (or `rep`). It is an error to attempt to output a list variable which has not been decimated into primary elements via a `repeat` construct.

Suppose that a variable `X` is two-dimensional (contains a list of lists). `X` must be nested twice in a `repeat`. The outer `repeat` will traverse the lists contained in `X`. The inner `repeat` will traverse the elements of each of these lists.

A nested `repeat` may be embedded in any of the clauses of a `repeat`, not only in the main clause.

7.7.9 The `rep` directive

The `rep` directive is similar to `repeat`. Whereas `repeat` is line-oriented, `rep` generates material within a line. It has all the same clauses, but everything is specified within one line:

```
@(rep)... main material ... .... special clauses ...@(end)
```

More than one `@(rep)` can occur within a line, mixed with other material. A `@(rep)` can be nested within a `@(repeat)` or within another `@(rep)`.

Also, `@(rep)` accepts the same `:counter` and `:vars` arguments.

7.7.10 `repeat` and `rep` Examples

Example 1: show the list `L` in parentheses, with spaces between the elements, or the word `EMPTY` if the list is empty:

```
@(output)
@(rep)@L @(single)(@L)@(first)(@L @(last)@L)@(empty)EMPTY@(end)
@(end)
```

Here, the `@(empty)` clause specifies `EMPTY`. So if there are no repetitions, the text `EMPTY` is produced. If there is a single item in the list `L`, then `@(single)(@L)` produces that item between parentheses. Otherwise if there are two or more items, the first item is produced with a leading parenthesis followed by a space by `@(first)(@L` and the last item is produced with a closing parenthesis: `@(last)@L)`. All items in between are emitted with a trailing space by the main clause: `@(rep)@L`.

Example 2: show the list L like Example 1 above, but the empty list is `()`.

```
@(output)
(@(rep)@L @(last)@L@(end))
@(end)
```

This is simpler. The parentheses are part of the text which surrounds the `@(rep)` construct, produced unconditionally. If the list `L` is empty, then `@(rep)` produces no output, resulting in `()`. If the list `L` has one or more items, then they are produced with spaces each one, except the last which has no space. If the list has exactly one item, then the `@(last)` applies to it instead of the main clause: it is produced with no trailing space.

7.7.11 The `close` directive

The syntax of the `close` directive is:

```
@(close expr)
```

Where *expr* evaluates to a stream. The `close` directive can be used to explicitly close streams created using `@(output ... :named var)` syntax, as an alternative to `@(output :finish expr)`.

Examples:

Write two lines to `"foo.txt"` over two output blocks using a single stream:

```
@(output "foo.txt" :named foo)
Hello,
@(end)
```

```
@(output :continue foo)
world!
@(end)
@(close foo)
```

The same as above, using `:finish` rather than `:continue` so that the stream is closed at the end of the second block:

```
@(output "foo.txt" :named foo)
Hello,
@(end)
@(output :finish foo)
world!
@(end)
```

7.7.12 Output Filtering

Often it is necessary to transform the output to preserve its meaning under the convention of a given data format. For instance, if a piece of text contains the characters < or >, then if that text is being substituted into HTML, these should be replaced by `<` and `>`. This is what filtering is for. Filtering is applied to the contents of output variables, not to any template text. **TXR** implements named filters. Built-in filters are named by keywords, given below. User-defined filters are possible, however. See notes on the deffilter directive below.

Instead of a filter name, the syntax `(fun name)` can be used. This denotes that the function called *name* is to be used as a filter. This is described in the next section Function Filters below.

Built-in filters named by keywords:

`:tohtml`
: Filter text to HTML, representing special characters using HTML ampersand sequences. For instance > is replaced by `>`.

`:tohtml*`
: Filter text to HTML, representing special characters using HTML ampersand sequences. Unlike `:tohtml`, this filter doesn't treat the single and double quote characters. It is not suitable for preparing HTML fragments which end up inserted into HTML tag attributes.

`:fromhtml`
: Filter text with HTML codes into text in which the codes are replaced by the corresponding characters. For instance `>` is replaced by >.

`:upcase`
: Convert the 26 lowercase letters of the English alphabet to uppercase.

`:downcase`
: Convert the 26 uppercase letters of the English alphabet to lowercase.

`:frompercent`
: Decode percent-encoded text. Character triplets consisting of the `%` character followed by a pair of hexadecimal digits (case insensitive) are are converted to bytes having the value represented by the hexadecimal digits (most significant nybble first). Sequences of one or more such bytes are treated

as UTF-8 data and decoded to characters.

`:topercent`
: Convert to percent encoding according to RFC 3986. The text is first converted to UTF-8 bytes. The bytes are then converted back to text as follows. Bytes in the range 0 to 32, and 127 to 255 (note: including the ASCII DEL), bytes whose values correspond to ASCII characters which are listed by RFC 3986 as being in the "reserved set", and the byte value corresponding to the ASCII `%` character are encoded as a three-character sequence consisting of the `%` character followed by two hexadecimal digits derived from the byte value (most significant nybble first, upper case). All other bytes are converted directly to characters of the same value without any such encoding.

`:fromurl`
: Decode from URL encoding, which is like percent encoding, except that if the unencoded + character occurs, it is decoded to a space character. The `%20` sequence still decodes to space, and `%2B` to the + character.

`:tourl`
: Encode to URL encoding, which is like percent encoding except that a space maps to + rather than `%20`. The + character, being in the reserved set, encodes to `%2B`.

`:frombase64`
: Decode from the Base 64 encoding described in RFC 4648, section 5.

`:tobase64`
: Encode to the Base 64 encoding described in RFC 4648, section 5.

`:frombase64url`
: Decode from the Base64 encoding described in RFC 4648, section 6. This uses the URL and filename safe alphabet, in which the + (plus) and / (slash) characters used in regular Base 64 are respectively replaced with − (minus) and _ (underscore).

`:tobase64url`
: Encode to the Base 64 encoding described in RFC 4648, section 6. See `:frombase64url` above.

`:tonumber`
: Converts strings to numbers. Strings that contain a period, `e` or `E` are converted to floating point as if by the Lisp function `flo-str`. Otherwise they are converted to integer as if using `int-str` with a radix of 10. Non-numeric junk results in the object `nil`.

`:toint`
: Converts strings to integers as if using `int-str` with a radix of 10. Non-numeric junk results in the object `nil`.

`:tofloat`
: Converts strings to floating-point values as if using the function `flo-str`. Non-numeric junk results in the object `nil`.

`:hextoint`
: Converts strings to integers as if using `int-str` with a radix of 16. Non-numeric junk results in the object `nil`.

Examples:

To escape HTML characters in all variable substitutions occurring in an output clause, specify `:filter :tohtml` in the directive:

```
@(output :filter :tohtml)
...
@(end)
```

To filter an individual variable, add the syntax to the variable spec:

```
@(output)
@{x :filter :tohtml}
@(end)
```

Multiple filters can be applied at the same time. For instance:

```
@(output)
@{x :filter (:upcase :tohtml)}
@(end)
```

This will fold the contents of `x` to uppercase, and then encode any special characters into HTML. Beware of combinations that do not make sense. For instance, suppose the original text is HTML, containing codes like `"`. The compound filter `(:upcase :fromhtml)` will not work because `"` will turn to `"` which no longer be recognized by the `:fromhtml` filter, since the entity names in HTML codes are case-sensitive.

Capture some numeric variables and convert to numbers:

```
@date @time @temperature @pressure
@(filter :tofloat temperature pressure)
@;; temperature and pressure can now be used in calculations
```

7.7.13 Function Filters

A function can be used as a filter. For this to be possible, the function must conform to certain rules:

1. The function must take two special arguments, which may be followed by additional arguments.
2. When the function is called, the first argument will be bound to a string, and the second argument will be unbound. The function must produce a value by binding it to the second argument. If the filter is to be used as the final filter in a chain, it must produce a string.

For instance, the following is a valid filter function:

```
@(define foo_to_bar (in out))
@  (next :string in)
@  (cases)
foo
@    (bind out "bar")
@  (or)
@    (bind out in)
```

```
@   (end)
@(end)
```

This function binds the `out` parameter to `"bar"` if the in parameter is `"foo"`, otherwise it binds the `out` parameter to a copy of the `in` parameter. This is a simple filter.

To use the filter, use the syntax `(:fun foo_to_bar)` in place of a filter name. For instance in the `bind` directive:

```
@(bind "foo" "bar" :lfilt (:fun foo_to_bar))
```

The above should succeed since the left side is filtered from `"foo"` to `"bar"`, so that there is a match.

Function filters can be used in a chain:

```
@(output :filter (:downcase (:fun foo_to_bar) :upcase))
...
@(end)
```

Here is a split function which takes an extra argument which specifies the separator:

```
@(define split (in out sep))
@   (next :list in)
@   (coll)@(maybe)@token@sep@(or)@token@(end)@(end)
@   (bind out token)
@(end)
```

Furthermore, note that it produces a list rather than a string. This function separates the argument in into tokens according to the separator text carried in the variable `sep`.

Here is another function, `join`, which catenates a list:

```
@(define join (in out sep))
@   (output :into out)
@   (rep)@in@sep@(last)@in@(end)
@   (end)
@(end)
```

Now here is these two being used in a chain:

```
@(bind text "how,are,you")
@(output :filter (:fun split ",") (:fun join "-"))
@text
@(end)
```

Output:

```
how-are-you
```

When the filter invokes a function, it generates the first two arguments internally to pass in the input value and capture the output. The remaining arguments from the `(:fun ...)` construct are also passed to the function. Thus the string objects `","` and `"-"` are passed as the `sep` argument to `split` and `join`.

Note that `split` puts out a list, which `join` accepts. So the overall filter chain operates on a string: a

string goes into split, and a string comes out of join.

7.7.14 The `deffilter` directive

The `deffilter` directive allows a query to define a custom filter, which can then be used in `output` clauses to transform substituted data.

The syntax of `deffilter` is illustrated in this example:

code:
```
@(deffilter rot13
   ("a" "n")
   ("b" "o")
   ("c" "p")
   ("d" "q")
   ("e" "r")
   ("f" "s")
   ("g" "t")
   ("h" "u")
   ("i" "v")
   ("j" "w")
   ("k" "x")
   ("l" "y")
   ("m" "z")
   ("n" "a")
   ("o" "b")
   ("p" "c")
   ("q" "d")
   ("r" "e")
   ("s" "f")
   ("t" "g")
   ("u" "h")
   ("v" "i")
   ("w" "j")
   ("x" "k")
   ("y" "l")
   ("z" "m"))
@(collect)
@line
@(end)
@(output :filter rot13)
@(repeat)
@line
@(end)
@(end)
```

data:
```
hey there!
```

output:
```
url gurer!
```

The `deffilter` symbol must be followed by the name of the filter to be defined, followed by bind expressions which evaluate to lists of strings. Each list must be at least two elements long and specifies one or more texts which are mapped to a replacement text. For instance, the following specifies a telephone keypad mapping from uppercase letters to digits.

```
@(deffilter alpha_to_phone ("E" "0")
                           ("J" "N" "Q" "1")
                           ("R" "W" "X" "2")
```

```
                                ("D" "S" "Y" "3")
                                ("F" "T" "4")
                                ("A" "M" "5")
                                ("C" "I" "V" "6")
                                ("B" "K" "U" "7")
                                ("L" "O" "P" "8")
                                ("G" "H" "Z" "9"))

@(deffilter foo (`@a` `@b`) ("c" `->@d`))

@(bind x ("from" "to"))
@(bind y ("---" "+++"))
@(deffilter sub x y)
```

The last `deffilter` has the same effect as the `@(deffilter sub ("from" "to") ("---" "+++"))` directive.

Filtering works using a longest match algorithm. The input is scanned from left to right, and the longest piece of text is identified at every character position which matches a string on the left-hand side, and that text is replaced with its associated replacement text. The scanning then continues at the first character after the matched text.

If none of the strings matches at a given character position, then that character is passed through the filter untranslated, and the scan continues at the next character in the input.

Filtering is not in-place but rather instantiates a new text, and so replacement text is not re-scanned for more replacements.

If a filter definition accidentally contains two or more repetitions of the same left-hand string with different right-hand translations, the later ones take precedence. No warning is issued.

7.7.15 The `filter` directive

The syntax of the `filter` directive is:

```
@(filter FILTER { VAR }+ )
```

A filter is specified, followed by one or more variables whose values are filtered and stored back into each variable.

Example: convert `a`, `b`, and `c` to uppercase and HTML encode:

```
@(filter (:upcase :tohtml) a b c)
```

7.8 Exceptions

7.8.1 Introduction

The exceptions mechanism in **TXR** is another disciplined form of nonlocal transfer, in addition to the blocks mechanism (see Blocks above). Like blocks, exceptions provide a construct which serves as the target for a dynamic exit. Both blocks and exceptions can be used to bail out of deep nesting when some condition occurs. However, exceptions provide more complexity. Exceptions are useful for error handling, and **TXR** in fact maps certain error situations to exception control transfers. However, exceptions are not inherently an error-handling mechanism; they are a structured dynamic control transfer mechanism, one of

whose applications is error handling.

An exception control transfer (simply called an exception) is always identified by a symbol, which is its type. Types are organized in a subtype-supertype hierarchy. For instance, the `file-error` exception type is a subtype of the `error` type. This means that a file error is a kind of error. An exception handling block which catches exceptions of type `error` will catch exceptions of type `file-error`, but a block which catches `file-error` will not catch all exceptions of type `error`. A `query-error` is a kind of error, but not a kind of `file-error`. The symbol `t` is the supertype of every type: every exception type is considered to be a kind of `t`. (Mnemonic: `t` stands for type, as in any type).

Exceptions are handled using `@(catch)` clauses within a `@(try)` directive.

In addition to being useful for exception handling, the `@(try)` directive also provides unwind protection by means of a `@(finally)` clause, which specifies query material to be executed unconditionally when the `try` clause terminates, no matter how it terminates.

7.8.2 The `try` directive

The general syntax of the `try` directive is

```
@(try)
... main clause, required ...
... optional catch clauses ...
... optional finally clause
@(end)
```

A `catch` clause looks like:

```
@(catch TYPE [ PARAMETERS ])
.
.
.
```

and also this simple form:

```
@(catch)
.
.
.
```

which catches all exceptions, and is equivalent to `@(catch t)`.

A `finally` clause looks like:

```
@(finally)
...
.
.
```

The main clause may not be empty, but the catch and finally may be.

A try clause is surrounded by an implicit anonymous block (see Blocks section above). So for instance, the following is a no-op (an operation with no effect, other than successful execution):

```
@(try)
```

```
@(accept)
@(end)
```

The `@(accept)` causes a successful termination of the implicit anonymous block. Execution resumes with query lines or directives which follow, if any.

`try` clauses and blocks interact. For instance, an `accept` from within a try clause invokes a `finally`.

code:
```
@(block foo)
@  (try)
@    (accept foo)
@  (finally)
@      (output)
bye!
@      (end)
@  (end)
```

output:
```
bye!
```

How this works: the `try` block's main clause is `@(accept foo)`. This causes the enclosing block named `foo` to terminate, as a successful match. Since the `try` is nested within this block, it too must terminate in order for the block to terminate. But the try has a `finally` clause, which executes unconditionally, no matter how the try block terminates. The `finally` clause performs some output, which is seen.

Note that `finally` interacts with `accept` in subtle ways not revealed in this example; they are documented in the description of `accept` under the `block` directive documentation.

7.8.3 The `finally` clause

A `try` directive can terminate in one of three ways. The main clause may match successfully, and possibly yield some new variable bindings. The main clause may fail to match. Or the main clause may be terminated by a nonlocal control transfer, like an exception being thrown or a block return (like the block foo example in the previous section).

No matter how the `try` clause terminates, the `finally` clause is processed.

The `finally` clause is itself a query which binds variables, which leads to questions: what happens to such variables? What if the `finally` block fails as a query? As well as: what if a `finally` clause itself initiates a control transfer? Answers follow.

Firstly, a `finally` clause will contribute variable bindings only if the main clause terminates normally (either as a successful or failed match). If the main clause of the `try` block successfully matches, then the `finally` block continues matching at the next position in the data, and contributes bindings. If the main clause fails, then the `finally` block tries to match at the same position where the main clause failed.

The overall `try` directive succeeds as a match if either the main clause or the `finally` clause succeed. If both fail, then the `try` directive is a failed match.

Example:

code:
```
@(try)
@a
@(finally)
@b
@(end)
@c
```

data:
```
1
2
3
```

result:
```
a="1"
b="2"
c="3"
```

In this example, the main clause of the `try` captures line `"1"` of the data as variable `a`, then the finally clause captures `"2"` as `b`, and then the query continues with the `@c` line after try block, so that `c` captures `"3"`.

Example:

code:
```
@(try)
hello @a
@(finally)
@b
@(end)
@c
```

data:
```
1
2
```

result:
```
b="1"
c="2"
```

In this example, the main clause of the `try` fails to match, because the input is not prefixed with `"hello "`. However, the `finally` clause matches, binding `b` to `"1"`. This means that the try block is a successful match, and so processing continues with `@c` which captures `"2"`.

When `finally` clauses are processed during a nonlocal return, they have no externally visible effect if they do not bind variables. However, their execution makes itself known if they perform side effects, such as output.

A `finally` clause guards only the main clause and the `catch` clauses. It does not guard itself. Once the finally clause is executing, the `try` block is no longer guarded. This means if a nonlocal transfer, such as a block accept or exception, is initiated within the finally clause, it will not re-execute the `finally` clause. The `finally` clause is simply abandoned.

The disestablishment of blocks and `try` clauses is properly interleaved with the execution of `finally` clauses. This means that all surrounding exit points are visible in a `finally` clause, even if the `finally` clause is being invoked as part of a transfer to a distant exit point. The finally clause can make a control transfer to an exit point which is more near than the original one, thereby "hijacking" the control transfer. Also, the anonymous block established by the `try` directive is visible in the `finally` clause.

Example:

```
@(try)
@  (try)
@    (next "nonexistent-file")
@  (finally)
@    (accept)
@  (end)
@(catch file-error)
@  (output)
file error caught
@  (end)
```

```
@(end)
```

In this example, the `@(next)` directive throws an exception of type `file-error`, because the given file does not exist. The exit point for this exception is the `@(catch file-error)` clause in the outermost `try` block. The inner block is not eligible because it contains no catch clauses at all. However, the inner try block has a finally clause, and so during the processing of this exception which is headed for `@(catch file-error)`, the `finally` clause performs an anonymous `accept`. The exit point for that `accept` is the anonymous block surrounding the inner `try`. So the original transfer to the `catch` clause is thereby abandoned. The inner `try` terminates successfully due to the `accept`, and since it constitutes the main clause of the outer try, that also terminates successfully. The `"file error caught"` message is never printed.

7.8.4 `catch` clauses

`catch` clauses establish their associated `try` blocks as potential exit points for exception-induced control transfers (called "throws").

A `catch` clause specifies an optional list of symbols which represent the exception types which it catches. The `catch` clause will catch exceptions which are a subtype of any one of those exception types.

If a `try` block has more than one `catch` clause which can match a given exception, the first one will be invoked.

When a `catch` is invoked, it is understood that the main clause did not terminate normally, and so the main clause could not have produced any bindings.

`catch` clauses are processed prior to `finally`.

If a `catch` clause itself throws an exception, that exception cannot be caught by that same clause or its siblings in the same try block. The `catch` clauses of that block are no longer visible at that point. Nevertheless, the `catch` clauses are still protected by the finally block. If a catch clause throws, or otherwise terminates, the `finally` block is still processed.

If a `finally` block throws an exception, then it is simply aborted; the remaining directives in that block are not processed.

So the success or failure of the `try` block depends on the behavior of the `catch` clause or the `finally` clause, if there is one. If either of them succeed, then the try block is considered a successful match.

Example:

code:
```
@(try)
@  (next "nonexistent-file")
@  x
@  (catch file-error)
@a
@(finally)
@b
@(end)
@c
```

data:
```
1
2
3
```

result:
```
a="1"
b="2"
c="3"
```

Here, the `try` block's main clause is terminated abruptly by a `file-error` exception from the `@(next)` directive. This is handled by the `catch` clause, which binds variable `a` to the input line `"1"`. Then the `finally` clause executes, binding `b` to `"2"`. The `try` block then terminates successfully, and so `@c` takes `"3"`.

7.8.5 `catch` Clauses with Parameters

A `catch` clause may have parameters following the type name, like this:

```
@(catch pair (a b))
```

To write a catch-all with parameters, explicitly write the master supertype t:

```
@(catch t (arg ...))
```

Parameters are useful in conjunction with `throw`. The built-in `error` exceptions carry one argument, which is a string containing the error message. Using `throw`, arbitrary parameters can be passed from the throw site to the catch site.

7.8.6 The `throw` directive

The `throw` directive generates an exception. A type must be specified, followed by optional arguments, which are bind expressions. For example,

```
@(throw pair "a" `@file.txt`)
```

throws an exception of type `pair`, with two arguments, being `"a"` and the expansion of the quasiliteral `` `@file.txt` ``.

The selection of the target `catch` is performed purely using the type name; the parameters are not involved in the selection.

Binding takes place between the arguments given in `throw` and the target `catch`.

If any `catch` parameter, for which a `throw` argument is given, is a bound variable, it has to be identical to the argument, otherwise the catch fails. (Control still passes to the `catch`, but the catch is a failed match).

code:
```
@(bind a "apple")
@(try)
@(throw e "banana")
@(catch e (a))
@(end)
```

result:
```
[query fails]
```

If any argument is an unbound variable, the corresponding parameter in the `catch` is left alone: if it is an unbound variable, it remains unbound, and if it is bound, it stays as is.

code:
```
@(try)
@(throw e "honda" unbound)
@(catch e (car1 car2))
@car1 @car2
@(end)
```

data: `honda toyota`

result:
```
car1="honda"
car2="toyota"
```

If a `catch` has fewer parameters than there are throw arguments, the excess arguments are ignored:

code:
```
@(try)
@(throw e "banana" "apple" "pear")
@(catch e (fruit))
@(end)
```

result: `fruit="banana"`

If a `catch` has more parameters than there are throw arguments, the excess parameters are left alone. They may be bound or unbound variables.

code:
```
@(try)
@(throw e "honda")
@(catch e (car1 car2))
@car1 @car2
@(end)
```

data: `honda toyota`

result:
```
car1="honda"
car2="toyota"
```

A `throw` argument passing a value to a `catch` parameter which is unbound causes that parameter to be bound to that value.

`throw` arguments are evaluated in the context of the `throw`, and the bindings which are available there. Consideration of what parameters are bound is done in the context of the catch.

code:
```
@(bind c "c")
@(try)
@(forget c)
@(bind (a c) ("a" "lc"))
@(throw e a c)
@(catch e (b a))
@(end)
```

result:
```
c="c"
b="a"
a="lc"
```

In the above example, `c` has a top-level binding to the string `"c"`, but then becomes unbound via `forget` within the `try` construct, and rebound to the value `"lc"`. Since the `try` construct is terminated by a `throw`, these modifications of the binding environment are discarded. Hence, at the end of the query, variable `c` ends up bound to the original value `"c"`. The `throw` still takes place within the scope of the bindings set up by the `try` clause, so the values of `a` and `c` that are thrown are `"a"` and `"lc"`. However, at the `catch` site, variable `a` does not have a binding. At that point, the binding to `"a"` established in the `try` has disappeared already. Being unbound, the `catch` parameter `a` can take whatever value the corresponding throw argument provides, so it ends up with `"lc"`.

There is a horizontal form of `throw`. For instance:

```
abc@(throw e 1)
```

throws exception `e` if `abc` matches.

If `throw` is used to generate an exception derived from type `error` and that exception is not handled,

TXR will issue diagnostics on the `*stderr*` stream and terminate. If an exception derived from `warning` is not handled, **TXR** will generate diagnostics on the `*stderr*` stream, after which control returns to the `throw` directive, and proceeds with the next directive. If an exception not derived from `error` is thrown, control returns to the `throw` directive and proceeds with the next directive.

7.8.7 The `defex` directive

The `defex` directive allows the query writer to invent custom exception types, which are arranged in a type hierarchy (meaning that some exception types are considered subtypes of other types).

Subtyping means that if an exception type `B` is a subtype of `A`, then every exception of type `B` is also considered to be of type `A`. So a catch for type `A` will also catch exceptions of type `B`. Every type is a supertype of itself: an `A` is a kind of `A`. This implies that every type is a subtype of itself also. Furthermore, every type is a subtype of the type `t`, which has no supertype other than itself. Type `nil` is a subtype of every type, including itself. The subtyping relationship is transitive also. If `A` is a subtype of `B`, and `B` is a subtype of `C`, then `A` is a subtype of `C`.

`defex` may be invoked with no arguments, in which case it does nothing:

```
@(defex)
```

It may be invoked with one argument, which must be a symbol. This introduces a new exception type. Strictly speaking, such an introduction is not necessary; any symbol may be used as an exception type without being introduced by `@(defex)`:

```
@(defex a)
```

Therefore, this also does nothing, other than document the intent to use a as an exception.

If two or more argument symbols are given, the symbols are all introduced as types, engaged in a subtype-supertype relationship from left to right. That is to say, the first (leftmost) symbol is a subtype of the next one, which is a subtype of the next one and so on. The last symbol, if it had not been already defined as a subtype of some type, becomes a direct subtype of the master supertype `t`. Example:

```
@(defex d e)
@(defex a b c d)
```

The first directive defines `d` as a subtype of `e`, and `e` as a subtype of `t`. The second defines `a` as a subtype of `b`, `b` as a subtype of `c`, and `c` as a subtype of `d`, which is already defined as a subtype of `e`. Thus `a` is now a subtype of `e`. The above can be condensed to:

```
@(defex a b c d e)
```

Example:

code:
```
@(defex gorilla ape primate)
@(defex monkey primate)
@(defex human primate)
@(collect)
@(try)
@(skip)
@(cases)
gorilla @name
@(throw gorilla name)
@(or)
```

```
monkey @name
@(throw monkey name)
@(or)
human @name
@(throw human name)
@(end)@#cases
@(catch primate (name))
@kind @name
@(output)
we have a primate @name of kind @kind
@(end)@#output
@(end)@#try
@(end)@#collect
```

data:
```
gorilla joe
human bob
monkey alice
```

output:
```
we have a primate joe of kind gorilla
we have a primate bob of kind human
we have a primate alice of kind monkey
```

Exception types have a pervasive scope. Once a type relationship is introduced, it is visible everywhere. Moreover, the `defex` directive is destructive, meaning that the supertype of a type can be redefined. This is necessary so that something like the following works right:

```
@(defex gorilla ape)
@(defex ape primate)
```

These directives are evaluated in sequence. So after the first one, the `ape` type has the type `t` as its immediate supertype. But in the second directive, `ape` appears again, and is assigned the `primate` supertype, while retaining `gorilla` as a subtype. This situation could be diagnosed as an error, forcing the programmer to reorder the statements, but instead **TXR** obliges. However, there are limitations. It is an error to define a subtype-supertype relationship between two types if they are already connected by such a relationship, directly or transitively. So the following definitions are in error:

```
@(defex a b)
@(defex b c)
@(defex a c)@# error: a is already a subtype of c, through b

@(defex x y)
@(defex y x)@# error: circularity; y is already a supertype of x.
```

7.8.8 The `assert` directive

The `assert` directive requires the remaining query or subquery which follows it to match. If the remainder fails to match, the `assert` directive throws an exception. If the directive is simply

```
@(assert)
```

Then it throws an assertion of type assert, which is a subtype of error. The `assert` directive also takes arguments similar to the `throw` directive: an exception symbol and additional arguments which are bind expressions, and may be unbound variables. The following assert directive, if it triggers, will throw an exception of type `foo`, with arguments `1` and `"2"`:

```
@(assert foo 1 "2")
```

Example:

```
@(collect)
Important Header
----------------
@(assert)
Foo: @a, @b
@(end)
```

Without the assertion in places, if the `Foo: @a, @b` part does not match, then the entire interior of the `@(collect)` clause fails, and the collect continues searching for another match.

With the assertion in place, if the text `"Important Header"` and its underline match, then the remainder of the collect body must match, otherwise an exception is thrown. Now the program will not silently skip over any Important Header sections due to a problem in its matching logic. This is particularly useful when the matching is varied with numerous cases, and they must all be handled.

There is a horizontal `assert` directive also. For instance:

```
abc@(assert)d@x
```

asserts that if the prefix `"abc"` is matched, then it must be followed by a successful match for `"d@x"`, or else an exception is thrown.

If the exception is not handled, and is derived from `error` then **TXR** issues diagnostics on the `*stderr*` stream and terminates. If the exception is derived from `warning` and not handled, **TXR** issues a diagnostic on `*stderr*` after which control returns to the `assert` directive. Control silently returns to the `assert` directive if an exception of any other kind is not handled.

When control returns to `assert` due to an unhandled exception, it behaves like a failed match, similarly to the require directive.

8 TXR LISP

The **TXR** language contains an embedded Lisp dialect called **TXR Lisp**.

This language is exposed in **TXR** in a number of ways.

In any situation that calls for an expression, a Lisp expression can be used, if it is preceded by the `@` character. The Lisp expression is evaluated and its value becomes the value of that expression. Thus, **TXR** directives are embedded in literal text using `@`, and Lisp expressions are embedded in directives using `@` also.

Furthermore, certain directives evaluate Lisp expressions without requiring `@`. These are `@(do)`, `@(require)`, `@(assert)`, `@(if)` and `@(next)`.

TXR Lisp code can be placed into files. On the command line, **TXR** treats files with a `".tl"`, `".tlo"` or `".tlo.gz"` suffix as **TXR Lisp** source or compiled code, and the `@(load)` directive does also.

TXR also provides an interactive listener for Lisp evaluation.

Lastly, **TXR Lisp** expressions can be evaluated via the command line, using the `-e` and `-p` options.

Examples:

Bind variable `a` to the integer 4:

```
@(bind a @(+ 2 2))
```

Bind variable `b` to the standard input stream. Note that `@` is not required on a Lisp variable:

```
@(bind a *stdin*)
```

Define several Lisp functions inside `@(do)`:

```
@(do
  (defun add (x y) (+ x y))

  (defun occurs (item list)
    (cond ((null list) nil)
          ((atom list) (eql item list))
          (t (or (eq (first list) item)
                 (occurs item (rest list)))))))
```

Trigger a failure unless previously bound variable `answer` is greater than 42:

```
@(require (> (int-str answer) 42)
```

8.1 Overview

TXR Lisp is a small and simple dialect, like Scheme, but much more similar to Common Lisp than Scheme. It has separate value and function binding namespaces, like Common Lisp (and thus is a Lisp-2 type dialect), and represents Boolean **true** and **false** with the symbols `t` and `nil` (note the case sensitivity of identifiers denoting symbols!). Furthermore, the symbol `nil` is also the empty list, which terminates nonempty lists.

TXR Lisp has lexically scoped local variables and dynamic global variables, similarly to Common Lisp, including the convention that `defvar` marks symbols for dynamic binding in local scopes. Lexical closures are supported. **TXR Lisp** also supports global lexical variables via `defvarl`.

Functions are lexically scoped in **TXR Lisp**; they can be defined in the pervasive global environment using `defun` or in local scopes using `flet` and `labels`.

8.2 Additional Syntax

Much of the **TXR Lisp** syntax has been introduced in the previous sections of the manual, since directive forms are based on it. There is some additional syntax that is useful in **TXR Lisp** programming.

8.2.1 Symbol Tokens

The symbol tokens in **TXR Lisp**, called a *`lident`* (Lisp identifier) has a similar syntax to the *`bident`* (braced identifier) in the **TXR** pattern language. It may consist of all the same characters, as well as the `/` (slash) character which may not be used in a *`bident`*. Thus a *`lident`* may consist of these characters, in addition to letters, numbers and underscores:

```
! $ % & * + - < = > ? \ ~ /
```

and may not look like a number.

A *`lident`* may also include all of the Unicode characters which are permitted in a *`bident`*.

The one character which is allowed in a *`lident`* but not in a *`bident`* is `/` (forward slash).

A lone `/` is a valid *`lident`* and consequently a symbol token in **TXR Lisp**. The token `/abc/` is also a symbol, and, unlike in a braced expression, is not a regular expression. In **TXR Lisp** expressions, regular expressions are written with a leading `#`.

8.2.2 Package Prefixes

If a symbol name contains a colon, the *lident* characters, if any, before that colon constitute the package prefix.

For example, the syntax `foo:bar` denotes `bar` symbol in the `foo` package.

It is a syntax error to read a symbol whose package doesn't exist.

If the package exists, but the symbol name doesn't exist in that package, then the symbol is interned in that package.

If the package name is an empty string (the colon is preceded by nothing), the package is understood to be the `keyword` package. The symbol is interned in that package.

The syntax `:test` denotes the symbol `test` in the `keyword` package, the same as `keyword:test`.

Symbols in the keyword package are self-evaluating. This means that when a keyword symbol is evaluated as a form, the value of that form is the keyword symbol itself. Exactly two non-keyword symbols also have this special self-evaluating behavior: the symbols `t` and `nil` in the user package, whose fully qualified names are `usr:t` and `usr:nil`.

The syntax `@foo:bar` denotes the meta prefix `@` being applied to the `foo:bar` symbol, not to a symbol in the `@foo` package.

The syntax `#:bar` denotes an uninterned symbol named `bar`, described in the next section.

Dialect Note:

> In ANSI Common Lisp, the `foo:bar` syntax does not intern the symbol `bar` in the `foo` package; the symbol must exist and be an exported symbol, or else the syntax is erroneous. In ANSI Common Lisp, the syntax `foo::bar` does intern `foo` in the `bar` package. **TXR**'s package system has no double-colon syntax, and lacks the concept of exported symbols.

8.2.3 Uninterned Symbols

Uninterned symbols are written with the `#:` prefix, followed by zero or more *lident* characters. When an uninterned symbol is read, a new, unique symbol is constructed, with the specified name. Even if two uninterned symbols have the same name, they are different objects. The `make-sym` and `gensym` functions produce uninterned symbols.

"Uninterned" means "not entered into a package". Interning refers to a process which combines package lookup with symbol creation, which ensures that multiple occurrences of a symbol name in written syntax are all converted to the same object: the first occurrence creates the symbol and associates it with its name in a package. Subsequent occurrences do not create a new symbol, but retrieve the existing one.

8.2.4 Meta-Atoms and Meta-Expressions

An expression may be preceded by the `@` (at sign) character. If the expression is an `atom`, then this is a meta-atom, otherwise it is a meta-expression.

When the atom is a symbol, this is also called a meta-symbol and in situations when such a symbol behaves like a variable, it is also referred to as a meta-variable.

When the atom is an integer, the meta-atom expression is called a meta-number.

Meta-atom and meta-expression expressions have no evaluation semantics; evaluating them throws an exception. They play a syntactic role in the `op` operator, which makes use of meta-variables and meta-numbers, and in structural pattern matching, which uses meta-variables as pattern variables and whose operator vocabulary is based on meta-expressions.

Meta-expressions also appear in the quasiliteral notation.

In other situations, application code may assign meaning to meta syntax as the programmer sees fit.

Meta syntax is defined as a shorthand notation, as follows:

If `X` is the syntax of an atom, such as a symbol, string or vector, then `@X` is a shorthand for the expression `(sys:var X)`. Here, `sys:var` refers to the `var` symbol in the `system-package`.

If `X` is a compound expression, either `(...)` or `[...]`, then `@X` is a shorthand for the expression `(sys:expr X)`.

The behavior of `@` followed by the syntax of a floating-point constant introduced by a leading decimal point, not preceded by digits, is unspecified. Examples of this are `@.123` and `@.123E+5`.

The behavior of `@` followed by the syntax of a floating-point expression in E notation, which lacks a decimal point, is also unspecified. An example of this is `@12E5`.

It is a syntax error for `@` to be followed by what appears to be a floating-point constant consisting of a decimal point flanked by digits on both sides. For instance `@1.2` is rejected.

A meta-expression followed by a period, and the syntax of another object is otherwise interpreted as a referencing dot expression. For instance `@1.E3` denotes `(qref @1 E3)` which, in turn, denotes `(qref (sys:var 1) E3)`, even though the unprefixed character sequence `1.E3` is otherwise a floating-point constant.

8.2.5 Consing Dot

Unlike other major Lisp dialects, **TXR Lisp** allows a consing dot with no forms preceding it. This construct simply denotes the form which follows the dot. That is to say, the parser implements the following transformation:

```
(. expr) -> expr
```

This is convenient in writing function argument lists that only take variable arguments. Instead of the syntax:

```
(defun fun args ...)
```

the following syntax can be used:

```
(defun fun (. args) ...)
```

When a `lambda` form is printed, it is printed in the following style.

```
(lambda nil ...) -> (lambda () ...)
(lambda sym ...) -> (lambda (. sym) ...)
(lambda (sym) ...) -> (lambda (sym) ...)
```

In no other circumstances is `nil` printed as `()`, or an atom *sym* as `(. sym)`.

This notation is implemented for the square brackets, according to this transformation:

```
[. expr] -> (dwim . expr)
```

This is useful in Structural Pattern Matching, allowing a pattern like

```
[. @args]
```

to match a `dwim` expression and capture all of its arguments in a variable, without having to resort to the internal notation:

Compatibility Note: support for `[. expr]` was introduced in **TXR** 282. Older versions do not read the syntax, but do print `(dwim . @var)` as `[. @var]` which is then unreadable in those versions, breaking read-print consistency.

8.2.6 Referencing Dot

A dot token which is flanked by expressions on both sides, without any intervening whitespace, is the referencing dot, and not the consing dot. The referencing dot is a syntactic sugar which translated to the `qref` syntax ("quoted ref"). When evaluated as a form, this syntax denotes structure access; see Structures. However, it is possible to put this syntax to use for other purposes, in other contexts.

```
;; a.b may be almost any expressions
a.b           <-->  (qref a b)
a.b.c         <-->  (qref a b c)
a.(qref b c)  <-->  (qref a b c)
(qref a b).c  <-->  (qref (qref a b) c)
```

That is to say, this dot operator constructs a `qref` expression out of its left and right arguments. If the right argument of the dot is already a qref expression (whether produced by another instance of the dot operator, or expressed directly) it is merged. This requires the qref dot operator to be right-to-left associative, so that `a.b.c` works by first translating `b.c` to `(qref b c)`, and then adjoining `a` to produce `(qref a b c)`.

If the referencing dot is immediately followed by a question mark, it forms a single token, which produces the following syntactic variation, in which the following item is annotated as a list headed by the symbol `t`:

```
a.?b     <-->  (t a).b         <--> (qref (t a) b)
a.?b.?c  <-->  (t a).(t b).c   <--> (qref (t a) (t b) c)
a.?(b)   <-->  (t a).(b)       <--> (qref (t a) (b))
(a).?b   <-->  (t (a)).b       <--> (qref (t (a)) b)
```

This syntax denotes *null-safe* access to structure slots and methods. `a.?b` means that `a` may evaluate to `nil`, in which case the expression yields `nil`; otherwise, `a` must evaluate to a `struct` which has a slot `b`, and the expression denotes access to that slot. Similarly, `a.?(b 1)` means that if `a` evaluates to `nil`, the expression yields `nil`; otherwise, `a` is treated as a struct object whose method `b` is invoked with argument `1`, and the value returned by that method becomes the value of the expression.

Integer tokens cannot be involved in this syntax, because they form floating-point constants when

juxtaposed with a dot. Such ambiguous uses of floating-point tokens are diagnosed as syntax errors:

```
(a.4)   ;; error: cramped floating-point literal
(a .4)  ;; good: a followed by 0.4
```

8.2.7 Unbound Referencing Dot

Closely related to the referencing dot syntax is the unbound referencing dot. This is a dot which is flanked by an expression on the right, without any intervening whitespace, but is not preceded by an expression Rather, it is preceded by whitespace, or some punctuation such as `[`, `(` or `'`. This is a syntactic sugar which translates to `uref` syntax:

```
.a       <--> (uref a)
.a.b     <--> (uref a b)
.a.?b    <--> (uref (t a) b)
```

If the unbound referencing dot is itself combined with a question mark to form the `.?` token, then the translation to `uref` is as follows:

```
.?a      <--> (uref t a)
.?a.b    <--> (uref t a b)
.?a.?b   <--> (uref t a (t b))
```

When the unbound referencing dot is applied to a dotted expression, this can be understood as a conversion of `qref` to `uref`.

Indeed, this is exactly what happens if the unbound dot is applied to an explicit `qref` expression:

```
.(qref a b)   <--> (uref a b)
```

The unbound referencing dot takes its name from the semantics of the `uref` macro, which produces a function that implements late binding of an object to a method slot. Whereas the expression `obj.a.b` denotes accessing object `obj` to retrieve slot `a` and then accessing slot `b` of the object from that slot, the expression `.a.b.` represents a "disembodied" reference: it produces a function which takes an object as an argument and then performs the implied slot referencing on that argument. When the function is called, it is said to bind the referencing to the object. Hence that referencing is "unbound".

Whereas the expression `.a` produces a function whose argument must be an object, `.?a` produces a function whose argument may be `nil`. The function detects this case and returns `nil`.

8.2.8 Quote and Quasiquote

`'`*expr* The quote character in front of an expression is used for suppressing evaluation, which is useful for forms that evaluate to something other than themselves. For instance if `'(+ 2 2)` is evaluated, the value is the three-element list `(+ 2 2)`, whereas if `(+ 2 2)` is evaluated, the value is `4`. Similarly, the value of `'a` is the symbol `a` itself, whereas the value of `a` is the contents of the variable `a`.

`^`*qq-template*
The caret in front of an expression is a quasiquote. A quasiquote is like a quote, but with the possibility of substitution of material.

Under a quasiquote, form is considered to be a quasiquote template. The template is considered to be a literal structure, except that it may contain the notations `,`*expr* and

`,*expr` which denote non-constant parts.

A quasiquote gets translated into code which, when evaluated, constructs the structure implied by *qq-template*, taking into account the unquotes and splices.

A quasiquote also processes nested quasiquotes specially.

If *qq-template* does not contain any unquotes or splices (which match its level of nesting), or is simply an atom, then `^qq-template` is equivalent to `'qq-template`. in other words, it is like an ordinary quote. For instance `^(a b ^(c ,d))` is equivalent to `'(a b ^(c ,d))`. Although there is an unquote ,d it belongs to the inner quasiquote `^(c ,d)`, and the outer quasiquote does not have any unquotes of its own, making it equivalent to a quote.

Dialect Note: in Common Lisp and Scheme, `^form` is written `` `form ``, and quasiquotes are also informally known as backquotes. In **TXR**, the backquote character `` ` `` used for quasistring literals.

`,expr` The comma character is used within a *qq-template* to denote an unquote. Whereas the quasiquote suppresses evaluation, similarly to the quote, the comma introduces an exception: an element of a form which is evaluated. For example, list `^(a b c ,(+ 2 2) (+ 2 2))` is the list `(a b c 4 (+ 2 2))`. Everything in the quasiquote stands for itself, except for the `,(+ 2 2)` which is evaluated.

Note: if a variable is called `*x*`, then the syntax `,*x*` means `,* x*`: splice the value of `x*`. In this situation, whitespace between the comma and the variable name must be used: `, *x*`.

`,*expr`
The comma-star operator is used within quasiquote list to denote a splicing unquote. The form which follows `,*` must evaluate to a list. That list is spliced into the structure which the quasiquote denotes. For example: `'(a b c ,*(list (+ 3 3) (+ 4 4) d))` evaluates to `(a b c 6 8 d)`. The expression `(list (+ 3 3) (+ 4 4))` is evaluated to produce the list `(6 8)`, and this list is spliced into the quoted template.

`@,*expr`
This syntax is not a distinct quasiquoting operator, but rather the combination of an unquote occurring as a meta-expression, denoting the structure `(sys:expr ,expr)`. This structure is treated specially by the quasiquote expander. Code is generated for it such that if *expr* evaluates to an value *val* which is an `atom`, then the result will be the `(sys:var val)` structure. If *val* is a `cons` rather than an `atom`, then the result is the `(sys:expr val)` structure. In other words, when quasiquoting is used to insert a value under the `@` meta prefix, the expander generates code to analyze the type of the value, and produce to the form which is most likely intended.

Dialect Notes:

In other Lisp dialects, like Scheme and ANSI Common Lisp, the equivalent syntax is usually `,@` (comma at). The `@` character already has an assigned meaning in **TXR**, so `*` is used.

However, `*` is also a character that may appear in a symbol name, which creates a potential for ambiguity. The syntax `,*abc` denotes the application of the `,*` splicing operator to the symbolic expression `abc`; to apply the ordinary non-splicing unquote to the symbol `*abc`, whitespace must

be used: `, *abc`.

In **TXR**, the unquoting and splicing forms may freely appear outside of a quasiquote template. If they are evaluated as forms, however, they throw an exception:

```
,(+ 2 2) ;; error!

',(+ 2 2) --> ,(+ 2 2)
```

In other Lisp dialects, a comma not enclosed by backquote syntax is treated as a syntax error by the reader.

8.2.9 Quasiquoting non-List Objects

Quasiquoting is supported over hash table and vector literals (see Vectors and Hashes below). A hash table or vector literal can be quoted, like any object, for instance:

```
'#(1 2 3)
```

The `#(1 2 3)` literal is turned into a vector atom right in the **TXR** parser, and this atom is being quoted: this is `(quote atom)` syntactically, which evaluates to *atom*.

When a vector is quasi-quoted, this is a case of `^atom` which evaluates to *atom*.

A vector can be quasiquoted, for example:

```
^#(1 2 3)
```

Unquotes can occur within a quasiquoted vector:

```
(let ((a 42))
  ^#(1 ,a 3)) ; value is #(1 42 3)
```

In this situation, the `^#(...)` notation produces code which constructs a vector.

The vector in the following example is also a quasivector. It contains unquotes, and though the quasiquote is not directly applied to it, it is embedded in a quasiquote:

```
(let ((a 42))
  ^(a b c #(d ,a))) ; value is (a b c #(d 42))
```

Hash-table literals have two parts: the list of hash construction arguments and the key-value pairs. For instance:

```
#H((:eql-based) (a 1) (b 2))
```

where `(:eql-based)` indicates that this hash table's keys are treated using `eql` equality, and `(a 1)` and `(b 2)` are the key/value entries. Hash literals may be quasiquoted. In quasiquoting, the arguments and pairs are treated as separate syntax; it is not one big list. So the following is not a possible way to express the above hash:

```
;; not supported: splicing across the entire syntax
(let ((hash-syntax '((:eql-based) (a 1) (b 2))))
  ^#H(,*hash-syntax))
```

This is correct:

```
;; fine: splicing hash arguments and contents separately
(let ((hash-args '(:eql-based))
      (hash-contents '((a 1) (b 2))))
  ^#H(,hash-args ,*hash-contents))
```

8.2.10 Quasiquoting combined with Quasiliterals

When a quasiliteral is embedded in a quasiquote, it is possible to use splicing to insert material into the quasiliteral.

Example:

```
(eval (let ((a 3)) ^`abc @,a @{,a} @{(list 1 2 ,a)}`))

-> "abc 3 3 1 2 3"
```

8.2.11 Vector Literals

`#(...)`

A hash token followed by a list denotes a vector. For example `#(1 2 a)` is a three-element vector containing the numbers `1` and `2`, and the symbol `a`.

8.2.12 Struct Literals

`#S(`*`name {slot value}*`*`)`

The notation `#S` followed by a nested list syntax denotes a struct literal. The first item in the syntax is a symbol denoting the struct type name. This must be the name of a struct type, otherwise the literal is erroneous. Followed by the struct type are slot names interleaved with their values. The values are literal expressions, not subject to evaluation. Each slot name which is present in the literal must name a slot in the struct type, though not all slots in the struct type must be present in the literal.

When a struct literal is read, the denoted struct type is constructed as if by a call to `make-struct` with an empty *`plist`* argument, followed by a sequence of assignments which store into each *`slot`* the corresponding *`value`* expression.

8.2.13 Hash Literals

`#H((`*`hash-argument`*`*) (`*`key value`*`)*)`

The notation `#H` followed by list syntax denotes a hash-table literal. The first item in the syntax is a list of keywords. These are the same keywords as are used when calling the function hash to construct a hash table. Allowed keywords are: `:equal-based`, `:eql-based`, `:eq-based`, `:weak-keys`, `:weak-vals`, and `:userdata`. If the `:userdata` keyword is present, it must be followed by an object; that object specifies the hash table's user data, which can be retrieved using the `hash-userdata` function. The `:equal-based`, `:eql-based` and `:eq-based` keywords are mutually exclusive.

An empty list can be specified as `nil` or `()`, which defaults to a hash table based on the `eql` function, with no weak semantics or user data.

The entire syntax following `#H` may be an empty list; however, that empty list may not be specified as `nil`; the empty parentheses notation is required.

The hash table's key-value contents are specified as zero or more two-element lists, whose first element specifies the *key* and whose second specifies the *value*. Both expressions are literal objects, not subject to evaluation.

8.2.14 Range Literals

`#R(from to)`

The notation `#R` followed by a two-element list syntax denotes a range literal. It combines *from* and *to* expressions, themselves literals not subject to evaluation, producing the range object whose corresponding `to` and `from` fields are the objects denoted by these expressions.

8.2.15 Buffer Literals

`#b'hex-data'`

The notation `#b'` introduces a buffer object: a data representation for a block of bytes. This `#b'` prefix must be followed by a data section and a closing quote. The data section consists of hexadecimal digits, among which may be interspersed whitespace: tabs, spaces and newlines. There must be an even number of digits, or else the notation is ill-formed. The whitespace is ignored, and pairs of successive hex digits specify bytes. If there are no hex digits, then a zero length buffer is specified.

Buffers may be constructed by the `make-buf` function, and other means such as the `ffi-get` function.

Note that the `#b` prefix is also used for binary numbers. In that syntax, it is followed by an optional sign, and then a mixture of one or more of the digits `0` or `1`.

8.2.16 Tree Node Literals

`#N([key [left [right]]])`

The notation `#N` followed by list syntax denotes a tree node literal. The list syntax must be a proper list that has up to three elements. If the list is empty, it may not be written as `nil`.

A tree node is an object of type `tnode`. Every `tnode` has three elements: a *key*, a *left* link and a *right* link. They may be objects of any type. If the tree node literal syntax omits any of these, they default to `nil`.

8.2.17 Tree Literals

`#T([([keyfun [lessfun [equalfun]]]) item*])`

The notation `#T` followed by list syntax denotes a tree literal, which specifies an object of type `tree`. Objects of type `tree` are search trees.

The list syntax which follows `#T` may be empty. If so, it cannot be written as `nil`.

The first element of the `#T` syntax, if present, must be a list of zero to three elements. These elements are symbols giving the names of the `tree` object's *key abstraction functions*. *keyfun* specifies the key function which is applied to each element to retrieve its key. If it is omitted, the object shall use the `identity` function as its key. The *lessfun* specifies the name of the comparison function by which keys are compared for inequality. It defaults to `less`. The *equalfun* specifies the function by which keys are compared for equality. It defaults to `equal`. A symbol which is specified as the name of any of these three special functions must be an element of the list stored in the special variable `*tree-fun-whitelist*`, otherwise the string literal is diagnosed as erroneous. Note: this is due to security considerations, since these three functions are executed during the processing of tree syntax.

A tree object is constructed from a tree literal by first creating an empty tree endowed with the three key abstraction functions that are indicated in the syntax, either explicitly or as defaults. Then, every *element* object is constructed from its respective literal syntax and inserted into the tree.

Duplicate objects are preserved. For instance the tree literal `#T(() 1 1 1)` specifies a tree with three nodes which have the same key. Duplicates appear in the tree in the order that they appear in the literal.

8.2.18 JSON Literals

`#J`*`json-syntax`*
: Introduces a JSON literal.

`#J^`*`json-syntax`*
: Introduces a JSON quasiliteral, allowing unquoting and splicing of Lisp expressions.

The implementation of JSON syntax is based on, and intended to conform with the IETF RFC 8259 document. Only **TXR**'s extensions to JSON syntax are described in this manual, as well as the correspondence between JSON syntax and Lisp.

The *`json-syntax`* is translated into a **TXR Lisp** object as follows.

A JSON string corresponds to a Lisp string. A JSON number corresponds to a Lisp floating-point number. A JSON array corresponds to a Lisp vector. A JSON object corresponds to an `equal`-based hash table.

The JSON Boolean symbols `true` and `false` translate to the Lisp symbols `t` and `nil`, respectively, those being the standard ones in the `usr` package.

The JSON symbol `null` maps to the `null` symbol in the `usr` package.

The `#J`*`json-syntax`* expression produces the object:

```
(json quote lisp-object)
```

where *`lisp-object`* is the Lisp value which corresponds to the *`json-syntax`*.

Similarly, but with a key difference, the `#J^`*`json-syntax`* expression produces the object:

```
(json sys:qquote lisp-object)
```

in which `quote` has been replaced with `sys:qquote`.

The `json` symbol is bound as a macro, which is expanded when a `#J` expression is evaluated.

The following remarks indicate special treatment and extensions in the processing of JSON. Similar remarks regarding the production of JSON are given under the `put-json` function.

When an invalid UTF-8 byte is encountered inside a JSON string, its value is mapped into the code point range U+DC01 to U+DCFF. That byte is consumed, and decoding continues with the next byte. This treatment is consistent with the treatment of invalid UTF-8 bytes in **TXR Lisp** literals and I/O streams. If the valid UTF-8 byte U+0000 (ASCII NUL) occurs in a JSON string, it is also mapped to U+DC00, **TXR**'s pseudo-null character. This treatment is consistent with **TXR** string literals and I/O streams.

The JSON escape sequence `\u0000` denoting the U+0000 NUL character is also converted to U+DC00.

TXR Lisp does not impose the restriction that the keys in a JSON object must be strings: `#J{1:2,true:false}` is accepted.

TXR Lisp allows the circle notation to occur within JSON syntax. See the section Notation for Circular and Shared Structure.

TXR Lisp allows for JSON syntax to be quasiquoted, and provides two extensions for writing unquotes and splicing unquotes. Within a JSON quasiquote, the ~ (tilde) character introduces a Lisp expression whose value is to be substituted at that point. Thus, the tilde serves the role of the unquoting comma used in Lisp quasiquotes. Splicing is indicated by the character sequence `~*`, which introduces a Lisp expression that is expected to produce a list, whose elements are interpolated into the JSON value.

Note: quasiquoting allows Lisp values to be introduced into the resulting object which are outside of the JSON type system, such as integers, characters, symbols or structures. These objects have no representation in JSON syntax.

Examples:

```
;; Basic JSON:

#Jtrue -> t
#Jfalse -> nil
(list #J true #Jtrue #Jfalse) -> (t t nil)
#J[1, 2, 3.14] -> #(1.0 2.0 3.14)
#J{"foo":"bar"} -> #H(() ("foo" "bar"))

;; Quoting JSON shows the json expression

'#Jfalse -> (json quote ())
'#Jtrue -> (json quote t)
'#J["a", true, 3.0] -> (json quote #("a" t 3.0))
'#J^[~(+ 2 2), 3] -> (json sys:qquote #(,(+ 2 2) 3.0))

:; Circle notation:

#J[#1="abc", #1#, #1#] -> #("abc" "abc" "abc")

;; JSON Quasiquote:

#J^[~*(list 1.0 2.0 3.0), ~(* 2.0 2), 5.0]
--> #(1.0 2.0 3.0 4.0 5.0)

;; Lisp quasiquote around JSON quote: requires evaluation round.

^#J[~*(list 1.0 2.0 3.0), ~(* 2.0 2), 5.0]
--> (json quote #(1.0 2.0 3.0 4.0 5.0))

(eval ^#J[~*(list 1.0 2.0 3.0), ~(* 2.0 2), 5.0])
--> #(1.0 2.0 3.0 4.0 5.0)
```

8.2.19 The `..` notation

In **TXR Lisp**, there is a special "dotdot" notation consisting of a pair of dots. This can be written between successive atoms or compound expressions, and is a shorthand for `rcons`.

That is to say, `A .. B` translates to `(rcons A B)`, and so for instance `(a b .. (c d) e .. f . g)` means `(a (rcons b (c d)) (rcons e f) . g)`.

The `rcons` function constructs a range object, which denotes a pair of values. Range objects are most commonly used for referencing subranges of sequences.

For instance, if `L` is a list, then `[L 1 .. 3]` computes a sublist of `L` consisting of elements 1 through 2 (counting from zero).

Note that if this notation is used in the dot position of an improper list, the transformation still applies. That is, the syntax `(a . b .. c)` is valid and produces the object `(a . (rcons b c))` which is another way of writing `(a rcons b c)`, which is quite probably nonsense.

The notation's `..` operator associates right to left, so that `a..b..c` denotes `(rcons a (rcons b c))`.

Note that range objects are not printed using the dotdot notation. A range literal has the syntax of a two-element list, prefixed by `#R`. (See Range Literals above.)

In any context where the dotdot notation may be used, and where it is evaluated to its value, a range literal may also be specified. If an evaluated dotdot notation specifies two constant expressions, then an equivalent range literal can replace it. For instance the form `[L 1 .. 3]` can also be written `[L #R(1 3)]`. The two are syntactically different, and so if these expressions are being considered for their syntax rather than value, they are not the same.

8.2.20 The DWIM Brackets

TXR Lisp has a square bracket notation. The syntax `[...]` is a shorthand way of writing `(dwim ...)`. The `[]` syntax is useful for situations where the expressive style of a Lisp-1 dialect is useful.

For instance if `foo` is a variable which holds a function object, then `[foo 3]` can be used to call it, instead of `(call foo 3)`. If foo is a vector, then `[foo 3]` retrieves the fourth element, like `(vecref foo 3)`. Indexing over lists, strings and hash tables is possible, and the notation is assignable.

Furthermore, any arguments enclosed in `[]` which are symbols are treated according to a modified namespace lookup rule.

More details are given in the documentation for the `dwim` operator.

8.2.21 Compound Forms

In **TXR Lisp**, there are two types of compound forms: the Lisp-2 style compound forms, denoted by ordinary lists that are expressed with parentheses. There are Lisp-1 style compound forms denoted by the DWIM Brackets, described in the previous section.

The first position of an ordinary Lisp-2 style compound form, is expected to have a function or operator name. Then arguments follow. There may also be an expression in the dotted position, if the form is a function call.

If the form is a function call then the arguments are evaluated. If any of the arguments are symbols, they are

treated according to Lisp-2 namespacing rules.

A function name may be a symbol, or else any of the syntactic forms given in the description of the function `func-get-name`.

8.2.22 Dot Position in Function Calls

If there is an expression in the dotted position of a function call expression, it is also evaluated, and the resulting value is involved in the function call in a special way.

Firstly, note that a compound form cannot be used in the dot position, for obvious reasons, namely that `(a b c . (foo z))` does not mean that there is a compound form in the dot position, but denotes an alternate spelling for `(a b c foo z)`, where foo behaves as a variable.

If the dot position of a compound form is an atom, then the behavior may be understood according to the following transformations:

```
(f a b c ... . x)  -->  (apply (fun f) a b c ... x)
[f a b c ... . x]  -->  [apply f a b c ... x]
```

In addition to atoms, meta-expressions and meta-symbols can appear in the dot position, even though their underlying syntax is comprised of a compound expression. This appears to work according to a transformation pattern which superficially appears to be the same as that for atoms:

```
(f a b c ... . @x)  -->  (apply (fun f) a b c ... @x)
```

However, in this situation, the `@x` is actually the form `(sys:var x)` and the dotted form is actually a proper list. The transformation is in fact taking place over a proper list, like this:

```
(f a b c ... sys:var x)  -->  (apply (fun f) a b c ... (sys:var @x))
```

That is to say, the **TXR Lisp** form expander reacts to the presence of a `sys:var` or `sys:expr` atom in embedded in the form. That symbol and the items which follow it are wrapped in an additional level of nesting, converted into a single compound form element.

Effectively, in all these cases, the dot notation constitutes a shorthand for `apply`.

Examples:

```
;; a contains 3
;; b contains 4
;; c contains #(5 6 7)
;; s contains "xyz"

(foo a b . c)  ;; calls (foo 3 4 5 6 7)
(foo a)        ;; calls (foo 3)
(foo . s)      ;; calls (foo #\x #\y #\z)

(list . a)     ;; yields 3
(list a . b)   ;; yields (3 . 4)
(list a . c)   ;; yields (3 5 6 7)
(list* a c)    ;; yields (3 . #(5 6 7))

(cons a . b)   ;; error: cons isn't variadic.
(cons a b . c) ;; error: cons requires exactly two arguments.
```

```
[foo a b . c]  ;; calls (foo 3 4 5 6 7)

[c 1]          ;; indexes into vector #(5 6 7) to yield 6

(call (op list 1 . @1) 2) ;; yields (1 . 2)
```

Note that the atom in the dot position of a function call may be a symbol macro. Since the semantics works as if by transformation to an apply form in which the original dot position atom is an ordinary argument, the symbol macro may produce a compound form.

Thus:

```
(symacrolet ((x 2))
  (list 1 . x))  ;; yields (1 . 2)

(symacrolet ((x (list 1 2)))
  (list 1 . x))  ;; yields (1 1 2)
```

That is to say, the expansion of `x` is not substituted into the form `(list 1 . x)` but rather the transformation to `apply` syntax takes place first, and so the substitution of `x` takes place in a form resembling `(apply (fun list) 1 x)`.

Dialect Note:

In some other Lisp dialects like ANSI Common Lisp, the improper list syntax may not be used as a function call; a function called apply (or similar) must be used for application even if the expression which gives the trailing arguments is a symbol. Moreover, applying sequences other than lists is not supported.

8.2.23 Improper Lists as Macro Calls

TXR Lisp allows macros to be called using forms which are improper lists. These forms are simply destructured by the usual macro parameter list destructuring. To be callable this way, the macro must have an argument list which specifies a parameter match in the dot position. This dot position must either match the terminating atom of the improper list form, or else match the trailing portion of the improper list form.

For instance if a macro mac is defined as

```
(defmacro mac (a b . c) ...)
```

then it may not be invoked as `(mac 1 . 2)` because the required argument `b` is not satisfied, and so the `2` argument cannot match the dot position `c` as required. The macro may be called as `(mac 1 2 . 3)` in which case `c` receives the form `3`. If it is called as `(mac 1 2 3 . 4)` then `c` receives the improper list form `3 . 4`.

8.2.24 Regular-Expression Literals

In **TXR Lisp**, the `/` character can occur in symbol names, and the `/` token is a symbol. Therefore the `/regex/` syntax is not used for denoting regular expressions; rather, the `#/regex/` syntax is used.

8.2.25 Notation for Circular and Shared Structure

TXR Lisp supports a printed notation called *circle notation* which accurately articulates the representation of objects which contain shared substructures as well as circular references. The notation is supported as a means of input, and is also optionally produced as output, controlled by the `*print-circle*` variable.

Ordinarily, shared substructure in printed objects is not evident, except in the case of multiple occurrences of interned symbols, in whose semantics it is implicit that they refer to the same object. Other shared structure is printed as separate copies which look like distinct objects. For instance, the object produced by `(let ((shared '(1 2))) (list shared shared))` is printed as `((1 2) (1 2))`, where it is not clear that the two occurrences of `(1 2)` are actually the same object. Under the circle notation, this object can be represented as `(#5=(1 2) #5#)`. The `#5=` part introduces a reference label, associating the arbitrarily chosen nonnegative integer 5 with the object which follows. The subsequent notation `#5#` simply refers to the object labeled by 5, reproducing that object by reference. The result is a two-element list which has the same `(1 2)` in two places.

Circular structure presents a greater challenge to printing: namely, if it is printed by a naive recursive descent, it results in infinite output, and possibly stack exhaustion due to recursion. The circle notation detects and handles circular references. For instance, the object produced by `(let ((c (list 1))) (rplacd c c))` produces a circular list which looks like an infinite list of 1's: `(1 1 1 1 ...)`. This cannot be printed. However, under the circle notation, it can be represented as `#1=(1 . #1#)`. The entire object itself is labeled by the integer 1. Then, enclosed within the syntax of that labeled object itself, a reference occurs to the label. This circular label reference represents the corresponding circular reference in the object.

A detailed description of the notational elements follows:

`#`*`digits`*`= object`

The `#=` syntax introduces an object label which denotes the object whose printed representation follows. The label is identified by the integer value arising from digits *`digits`* which are one or more decimal digits. Note: the value zero is permitted; even though when the notation is produced by the **TXR Lisp** printer, labeling begins at 1. Negative values are not possible because a leading sign is not part of the syntax.

There may be no more than one definition for a given label within the syntactic scope being parsed, otherwise a syntax error occurs. In **TXR** pattern language code, an entire source file is parsed as one unit, and so scope for the circular notation's references is the entire source file. Files processed by `@(include)` have their own scope. The scope for labels in **TXR Lisp** source code is the top-level expression in which they appear. Consequently, references in one **TXR Lisp** top-level expression cannot reach definitions in another.

`#`*`digits`*`#`

The `##` syntax denotes a label reference: the repetition of an object that was previously labeled by the integer given by *`digits`*. If no such label had been introduced in the syntactic scope, a syntax error occurs. An object was previously labeled by *`digits`* if a `#=` definition occurs in the same syntactic scope as the reference, and is applied to an object which either encloses the reference, or lexically precedes the reference. Forward references such as `(#1# #1=(1 2))` are not supported.

Note:

Circular notation can span hash-table literals. The syntax `#1=#H((:eql-based) (#1# #1#))` denotes an `eql`-based hash table which contains one entry, in which that same table itself is both the key and value. This kind of circularity is not supported for `equal`-based hash tables. The analogous syntax `#1=#H(() (#1# #1#))` produces a hash table in an inconsistent state.

Dialect Note:

Circle notation is taken from Common Lisp, intended to be unsurprising to users familiar with that language. The implementation is based on descriptions in the ANSI Common Lisp document, judiciously

taking into account the content of the X3J13 Cleanup Issues named PRINT-CIRCLE-STRUCTURE:USER-FUNCTIONS-WORK and PRINT-CIRCLE-SHARED:RESPECT-PRINT-CIRCLE.

8.2.26 Notation for Erasing Objects

`#; expr`

The **TXR Lisp** notation `#;` in TXR Lisp indicates that the expression *expr* is to be read and then discarded, as if it were replaced by whitespace.

This is useful for temporarily "commenting out" an expression.

Notes:

Whereas it is valid for a **TXR Lisp** source file to be empty, it is a syntax error if a **TXR Lisp** source file contains nothing but one or more objects which are each suppressed by a preceding `#;`. In the interactive listener, an input line consisting of nothing but commented-out objects is similarly a syntax error.

The notation does not cascade; consecutive occurrences of `#;` trigger a syntax error.

The notation interacts with the circle notation. Firstly, if an object which is erased by `#;` contains circular-referencing instances of the label notation, those instances refer to `nil`. Secondly, commented-out objects may introduce labels which are subsequently referenced in *expr*. An example of the first situation occurs in:

```
#;(#1=(#1#))
```

Here the `#1#` label is a circular reference because it refers to an object which is a parent of the object which contains that reference. Such a reference is only satisfied by a "backpatching" process once the entire surrounding syntax is processed to the top level. The erasure perpetrated by `#;` causes the `#1#` label reference to be replaced by `nil`, and therefore the labeled object is the object `(nil)`.

An example of the second situation is

```
#;(#2=(a b c)) #2#
```

Here, even though the expression `(#2=(a b c))` is suppressed, the label definition which it has introduced persists into the following object, where the label reference `#2#` resolves to `(a b c)`.

A combination of the two situations occurs in

```
#;(#1=(#1#)) #1#
```

which yields `(nil)`. This is because the `#1=` label is available; but the earlier `#1#` reference, being a circular reference inside an erased object, had lapsed to `nil`.

8.3 Generalization of List Accessors

In ancient Lisp in the 1960's, it was not possible to apply the operations `car` and `cdr` to the `nil` symbol (empty list), because it is not a `cons` cell. In the InterLisp dialect, this restriction was lifted: these operations were extended to accept `nil` (and return `nil`). The convention was adopted in other Lisp dialects such as MacLisp and eventually in Common Lisp. Thus there exists an object which is not a cons, yet which takes `car` and `cdr`.

In **TXR Lisp**, this relaxation is extended further. For the sake of convenience, the operations `car` and `cdr`, are made to work with strings and vectors:

```
(cdr "") -> nil
(car "") -> nil

(car "abc") -> #\a
(cdr "abc") -> "bc"

(cdr #(1 2 3)) -> #(2 3)
(car #(1 2 3)) -> 1
```

Moreover, structure types which define the methods `car`, `cdr` and `nullify` can also be treated in the same way.

The `ldiff` function is also extended in a special way. When the right parameter a non-list sequence, then it uses the equal equality test rather than eq for detecting the tail of the list.

```
(ldiff "abcd" "cd") -> (#\a #\b)
```

The `ldiff` operation starts with `"abcd"` and repeatedly applies `cdr` to produce `"bcd"` and `"cd"`, until the suffix is equal to the second argument: `(equal "cd" "cd")` yields true.

Operations based on `car`, `cdr` and `ldiff`, such as `keep-if` and `remq` extend to strings and vectors.

Most derived list processing operations such as `remq` or `mapcar` obey the following rule: the returned object follows the type of the leftmost input list object. For instance, if one or more sequences are processed by `mapcar`, and the leftmost one is a character string, the function is expected to return characters, which are converted to a character string. However, in the event that the objects produced cannot be assembled into that type of sequence, a list is returned instead.

For example `[mapcar list "ab" "12"]` returns `((#\a #\b) (#\1 #\2))`, because a string cannot hold lists of characters. However `[mappend list "ab" "12"]` returns `"a1b2"`.

The lazy versions of these functions such as `mapcar*` do not have this behavior; they produce lazy lists.

8.4 Generalization of Iteration

TXR Lisp implements a unified paradigm for iterating over sequence-like container structures and abstract spaces such as bounded and unbounded ranges of integers. This concept is based around an iterator abstraction which is directly compatible with Lisp cons-cell traversal in the sense that when iteration takes place over lists, the iterator instance is nothing but a cons cell.

An iterator is created using the constructor function `iter-begin` which takes a single argument. The argument denotes a space to be traversed; the iterator provides the means for that traversal.

When the `iter-begin` function is applied to a list (a `cons` cell or the `nil` object), the return value is that object itself. The remaining functions in the iterator API then behave like aliases for list processing functions. The `iter-more` function behaves like `identity`, `iter-item` behaves like `car` and `iter-step` behaves like `cdr`.

For example, the following loops not only produce identical behavior, but the `iter` variable steps through the `cons` cells in the same manner in both:

```
;; print all symbols in the list (a b c d):

(let ((iter '(a b c d)))
  (while iter
```

```
    (prinl (car iter))
    (set iter (cdr iter))))

;; likewise:

(let ((iter (iter-begin '(a b c d))))
  (while (iter-more iter)
    (prinl (iter-item iter))
    (set iter (iter-step iter))))
```

There are three important differences.

Firstly, both examples will still work if the list `(a b c d)` is replaced by a different kind of sequence, such as the string `"abcd"` or the vector `#(a b c d)`. However, the former example will not execute efficiently on these objects. The reason is that the `cdr` function will construct successive suffixes of the string and list object. That requires not only the allocation of memory, but changes the running time complexity of the loop from linear to quadratic.

Secondly, the former example with `car`/`cdr` will not work correctly if the sequence is an empty non-list sequence, like the null string or empty vector. Rectifying this problem requires the `nullify` function to be used:

```
;; print all symbols in the list (a b c d):

(let ((iter (nullify "abcd")))
  (while iter
    (prinl (car iter))
    (set iter (cdr iter))))
```

The `nullify` function converts empty sequences of all kinds into the empty list `nil`.

Thirdly, the second example will work even if the input list is replaced with certain objects which are not sequences at all:

```
;; Print the integers from 0 to 3

(let ((iter (iter-begin 0..4)))
  (while (iter-more iter)
    (prinl (iter-item iter))
    (set iter (iter-step iter))))

;; Print incrementing integers starting at 1,
;; breaking out of the loop after 100.

(let ((iter (iter-begin 1)))
  (while (iter-more iter)
    (if (eql 100 (prinl (iter-item iter)))
      (return))
    (set iter (iter-step iter))))
```

In **TXR Lisp**, numerous functions that appear as list processing functions in other contemporary Lisp dialects, and historically, are actually sequence processing functions based on the above iterator paradigm.

8.5 Callable Objects

In **TXR Lisp**, sequences (strings, vectors and lists) as well as hashes and regular expressions can be used as functions everywhere, not just with the DWIM brackets.

Sequences work as one- or two-argument functions. With a single argument, an element is selected by position and returned. With two arguments, a range is extracted and returned.

Moreover, when a sequence is used as a function of one argument, and the argument is a range object rather than an integer, then the call is equivalent to the two-argument form. This is the basis for array slice syntax like `["abc" 0..1] .`

Hashes also work as one or two argument functions, corresponding to the arguments of the gethash function.

A regular expression behaves as a one, two, or three argument function, which operates on a string argument. It returns the leftmost matching substring, or else `nil`.

Example 1:

```
(mapcar "abc" '(2 0 1)) -> (#\c #\a #\b)
```

Here, `mapcar` treats the string `"abc"` as a function of one argument (since there is one list argument). This function maps the indices `0`, `1` and `2` to the corresponding characters of string `"abc"`. Through this function, the list of integer indices `(2 0 1)` is taken to the list of characters `(#\c #\a #\b)`.

Example 2:

```
(call '(1 2 3 4) 1..3) -> (2 3)
```

Here, the shorthand `1 .. 3` denotes `(rcons 1 3)`. A range used as an argument to a sequence performs range extraction: taking a slice starting at index 1, up to and not including index 3, as if by the call `(sub '(1 2 3 4) 1 3)`.

Example 3:

```
(call '(1 2 3 4) '(0 2)) -> (1 2)
```

A list of indices applied to a sequence is equivalent to using the select function, as if `(select '(1 2 3 4) '(0 2))` were called.

Example 4:

```
(call #/b./ "abcd") -> "bc"
```

Here, the regular expression, called as a function, finds the matching substring `"bc"` within the argument `"abcd"`.

8.6 Special Variables

Similarly to Common Lisp, **TXR Lisp** is lexically scoped by default, but also has dynamically scoped (a.k.a "special") variables.

When a variable is defined with `defvar` or `defparm`, a binding for the symbol is introduced in the global name space, regardless of in what scope the `defvar` form occurs.

Furthermore, at the time the defvar form is evaluated, the symbol which names the variable is tagged as special.

When a symbol is tagged as special, it behaves differently when it is used in a lexical binding construct like `let`, and all other such constructs such as function parameter lists. Such a binding is not the usual lexical binding, but a "rebinding" of the global variable. Over the dynamic scope of the form, the global variable takes on the value given to it by the rebinding. When the form terminates, the prior value of the variable is restored. (This is true no matter how the form terminates; even if by an exception.)

Because of this "pervasive special" behavior of a symbol that has been used as the name of a global variable, a good practice is to make global variables have visually distinct names via the "earmuffs" convention: beginning and ending the name with an asterisk.

Example:

```
(defvar *x* 42)     ;; *x* has a value of 42

(defun print-x ()
  (format t "~a\n" *x*))

(let ((*x* "abc"))  ;; this overrides *x*
  (print-x))        ;; *x* is now "abc" and so that is printed

(print-x)           ;; *x* is 42 again and so "42" is printed
```

Dialect Note 1:

The terms *bind* and *binding* are used differently in **TXR Lisp** compared to ANSI Common Lisp. In **TXR Lisp** binding is an association between a symbol and an abstract storage location. The association is registered in some namespace, such as the global namespace or a lexical scope. That storage location, in turn, contains a value. In ANSI Lisp, a binding of a dynamic variable is the association between the symbol and a value. It is possible for a dynamic variable to exist, and not have a value. A value can be assigned, which creates a binding. In **TXR Lisp**, an assignment is an operation which transfers a value into a binding, not one which creates a binding.

In ANSI Lisp, a dynamic variable can exist which has no value. Accessing the value signals a condition, but storing a value is permitted; doing so creates a binding. By contrast, in **TXR Lisp** a global variable cannot exist without a value. If a `defvar` form doesn't specify a value, and the variable doesn't exist, it is created with a value of `nil`.

Dialect Note 2:

Unlike ANSI Common Lisp, **TXR Lisp** has global lexical variables in addition to special variables. These are defined using `defvarl` and `defparml`. The only difference is that when variables are introduced by these macros, the symbols are not marked special, so their binding in lexical scopes is not altered to dynamic binding.

Many variables in **TXR Lisp**'s standard library are global lexicals. Those which are special variables obey the "earmuffs" convention in their naming. For instance `s-ifmt`, `log-emerg` and `sig-hup` are global lexicals, because they provide constant values for which overriding doesn't make sense. On the other hand the standard output stream variable `*stdout*` is special. Overriding it over a dynamic scope is useful, as a means of redirecting the output of functions which write to the `*stdout*` stream.

Dialect Note 3:

In Common Lisp, `defparm` is known as `defparameter`.

8.7 Syntactic Places and Accessors

The **TXR Lisp** feature known as *syntactic places* allows programs to use the syntax of a form which is used to *access* a value from an environment or object, as an expression which denotes a *place* where a value may be *stored.*

They are almost exactly the same concept as "generalized references" in Common Lisp, and are related to "lvalues" in languages in the C family, or "designators" in Pascal.

8.7.1 Symbolic Places

A symbol is a is a syntactic place if it names a variable. If `a` is a variable, then it may be assigned using the `set` operator: the form `(set a 42)` causes `a` to have the integer value 42.

8.7.2 Compound Places

A compound expression can be a syntactic place, if its leftmost constituent is as symbol which is specially registered, and if the form has the correct syntax for that kind of place, and suitable semantics. Such an expression is a compound place.

An example of a compound place is a `car` form. If `c` is an expression denoting a `cons` cell, then `(car c)` is not only an expression which retrieves the value of the `car` field of the cell. It is also a syntactic place which denotes that field as a storage location. Consequently, the expression `(set (car c) "abc")` stores the character string `"abc"` in that location. Although the same effect can be obtained with `(rplaca c "abc")` the syntactic place frees the programmer from having to remember different update functions for different kinds of places. There are various other advantages. **TXR Lisp** provides a plethora of operators for modifying a place in addition to `set`. Subject to certain usage restrictions, these operators work uniformly on all places. For instance, the expression `(rotate (car x) [str 3] y)` causes three different kinds of places to exchange contents, while the three expressions denoting those places are evaluated only once. New kinds of place update macros like `rotate` are quite easily defined, as are new kinds of compound places.

8.7.3 Accessor Functions

When a function call form such as the above `(car x)` is a syntactic place, then the function is called an *accessor*. This term is used throughout this document to denote functions which have associated syntactic places.

8.7.4 Macro Call Syntactic Places

Syntactic places can be macros (global and lexical), including symbol macros. So for instance in `(set x 42)` the `x` place can actually be a symbolic macro which expands to, say, `(cdr y)`. This means that the assignment is effectively `(set (cdr y) 42)`.

8.7.5 User-Defined Syntactic Places and Place Operators

Syntactic places, as well as operators upon syntactic places, are both open-ended. Code can be written quite easily in **TXR Lisp** to introduce new kinds of places, as well as new place-mutating operators. New places can be introduced with the help of the `defplace`, `define-accessor` or `defset` macros, or possibly the `define-place-macro` macro in simple cases when a new syntactic place can be expressed as a transformation to the syntax of an existing place. Three ways exist for developing new place update macros (place operators). They can be written using the ordinary macro definer ordinary macro definer

defmacro, with the help of special utility macros called with-update-expander, with-clobber-expander, and with-delete-expander. They can also be written using defmacro in conjunction with the operators placelet or placelet*. Simple update macros similar to inc and push can be written compactly using define-modify-macro.

8.7.6 Deletable Places

Unlike generalized references in Common Lisp, **TXR Lisp** syntactic places support the concept of deletion. Some kinds of places can be deleted, which is an action distinct from (but does not preclude) being overwritten with a value. What exactly it means for a place to be deleted, or whether that is even permitted, depends on the kind of place. For instance a place which denotes a lexical variable may not be deleted, whereas a global variable may be. A place which denotes a hash-table entry may be deleted, and results in the entry being removed from the hash table. Deleting a place in a list causes the trailing items, if any, or else the terminating atom, to move in to close the gap. Users may define new kinds of places which support deletion semantics.

8.7.7 Evaluation of Places

To bring about their effect, place operators must evaluate one or more places. Moreover, some of them evaluate additional forms which are not places. Which arguments of a place operator form are places and which are ordinary forms depends on its specific syntax. For all the built-in place operators, the position of an argument in the syntax determines whether it is treated as (and consequently required to be) a syntactic place, or whether it is an ordinary form.

All built-in place operators perform the evaluation of place and non-place argument forms in strict left-to-right order.

Place forms are evaluated not in order to compute a value, but in order to determine the storage location. In addition to determining a storage location, the evaluation of a place form may possibly give rise to side effects. Once a place is fully evaluated, the storage location can then be accessed. Access to the storage location is not considered part of the evaluation of a place. To determine a storage location means to compute some hidden referential object which provides subsequent access to that location without the need for a reevaluation of the original place form. (The subsequent access to the place through this referential object may still require a multi-step traversal of a data structure; minimizing such steps is a matter of optimization.)

Place forms may themselves be compounds, which contain subexpressions that must be evaluated. All such evaluation for the built-in places takes place in left to right order.

Certain place operators, such as shift and rotate, exhibit an unspecified behavior with regard to the timing of the access of the prior value of a place, relative to the evaluation of places which occur later in the same place operator form. Access to the prior values may be delayed until the entire form is evaluated, or it may be interleaved into the evaluation of the form. For example, in the form (shift a b c 1), the prior value of a can be accessed and saved as soon as a is evaluated, prior to the evaluation of b. Alternatively, a may be accessed and saved later, after the evaluation of b or after the evaluation of all the forms. This issue affects the behavior of place-modifying forms whose subforms contain side effects. It is recommended that such forms not be used in programs.

8.7.8 Nested Places

Certain place forms are required to have one or more arguments which are themselves places. The prime example of this, and the only example from among built-in syntactic places, are DWIM forms. A DWIM form has the syntax

```
(dwim obj-place index [alt])
```

and the square-bracket-notation equivalent:

```
[obj-place index [alt]]
```

Note that not only is the entire form a place, denoting some element or element range of *obj-place*, but there is the added constraint that *obj-place* must also itself be a syntactic place.

This requirement is necessary, because it supports the behavior that when the element or element range is updated, then *obj-place* is also potentially updated.

After the assignment `(set [obj 0..3] '("forty" "two"))` not only is the range of places denoted by `[obj 0..3]` replaced by the list of strings `("forty" "two")` but `obj` may also be overwritten with a new value.

This behavior is necessary because the DWIM brackets notation maintains the illusion of an encapsulated array-like container over several dissimilar types, including Lisp lists. But Lisp lists do not behave as fully encapsulated containers. Some mutations on Lisp lists return new objects, which then have to stored (or otherwise accepted) in place of the original objects in order to maintain the array-like container illusion.

8.7.9 Built-In Syntactic Places

The following is a summary of the built-in place forms, in addition to symbolic places denoting variables. New syntactic place forms can be defined by **TXR** programs.

```
(car object)
(first object)
(rest object)
(second object)
(third object)
...
(tenth object)
(last object [num])
(butlast object [num])
(cdr object)
(caar object)
(cadr object)
(cdar object)
(cddr object)
...
(cddddr object)
(nthcdr index obj)
(nthlast index obj)
(butlastn num obj)
(last num obj)
(nth index obj)
(ref seq idx)
(sub sequence [from [to]])
(vecref vec idx)
(chr-str str idx)
(gethash hash key [alt])
(hash-userdata hash)
(dwim obj-place index [alt])
(sub-list obj [from [to]])
(sub-vec obj [from [to]])
(sub-str str [from [to]])
```

```
[obj-place index [alt]] ;; equivalent to dwim
(symbol-value symbol-valued-form)
(symbol-function function-name-valued-form)
(symbol-macro symbol-valued-form)
(fun function-name)
(force promise)
(errno)
(slot struct-obj slot-name-valued-form)
(qref struct-obj slot-name) ;; by macro-expansion to (slot ...)
struct-obj.slot-name ;; equivalent to qref
(sock-peer socket)
(sock-opt socket level option [ffi-type])
(carray-sub carray [from [to]])
(sub-buf buf [from [to]])
(left node)
(right node)
(key node)
(read-once node)
```

8.7.10 Built-In Place-Mutating Operators

The following is a summary of the built-in place mutating macros. They are described in detail in their own sections.

`(set {place new-value}*)`
: Assigns the values of expressions to places, performing assignments in left-to-right order, returning the value assigned to the rightmost place.

`(pset {place new-value}*)`
: Assigns the values of expressions to places, performing the determination of places and evaluation of the expressions left to right, but the assignment in parallel. Returns the value assigned to the rightmost place.

`(zap place [new-value])`
: Assigns *new-value* to place, defaulting to `nil`, and returns the prior value.

`(flip place)`
: Logically toggles the Boolean value of *place*, and returns the new value.

`(test-set place)`
: If *place* contains `nil`, stores `t` into the place and returns `t` to indicate that the store took place. Otherwise does nothing and returns `nil`.

`(test-clear place)`
: If *place* contains a Boolean true value, stores `nil` into the place and returns `t` to indicate that the store took place. Otherwise does nothing and returns `nil`.

`(compare-swap place cmp-fun cmp-val store-val)`
: Examines the value of *place* and compares it to *cmp-val* using the comparison function given by the function name *cmp-fun*. If the comparison is false, returns `nil`. Otherwise, stores the *store-val* value into *place* and returns `t`.

`(inc place [delta])`
Increments *place* by *delta*, which defaults to 1, and returns the new value.

`(dec place [delta])`
Decrements *place* by *delta*, which defaults to 1, and returns the new value.

`(pinc place [delta])`
Increments *place* by *delta*, which defaults to 1, and returns the old value.

`(pdec place [delta])`
Decrements *place* by *delta*, which defaults to 1, and returns the old value.

`(test-inc place [delta [from-val]])`
Increments *place* by *delta* and returns `t` if the previous value was `eql` to *from-val*, where *delta* defaults to 1 and *from-val* defaults to zero.

`(test-dec place [delta [to-val]])`
Decrements *place* by *delta* and returns `t` if the new value is `eql` to *to-val*, where *delta* defaults to 1 and *to-val* defaults to 0.

`(swap left-place right-place)`
Exchanges the values of *left-place* and *right-place*.

`(push item place)`
Adds *item* to the front of the list which is currently stored in `place`, then stores the extended list back into `place` and returns it.

`(pop place)`
Pop the list stored in *place* and returns the popped value.

`(shift place+ shift-in-value)`
Treats one or more places as a "multi-place shift register". Values are shifted to the left among the places. The rightmost place receives *shift-in-value*, and the value of the leftmost place emerges as the return value.

`(rotate place*)`
Treats zero or more places as a "multi-place rotate register". The places exchange values among themselves, by a rotation by one place to the left. The value of the leftmost place goes to the rightmost place, and that value is returned.

`(del place)`
Deletes a place which supports deletion, and returns the value which existed in that place prior to deletion.

`(lset {place}+ sequence)`
Sets multiple places to values obtained from successive elements of *sequence*.

`(upd place opip-arg*)`
: Applies an `opip`-style operational pipeline to the value of *place* and stores the result back into *place*.

`(set-mask place integer*)`
: Sets to 1 the bits in *place* corresponding to bits that are equal to 1 in the mask made up of the *integer* arguments (by combining them together with the inclusive or operation).

`(clear-mask place integer*)`
: Clears (sets to 0) the bits in *place* corresponding to bits that are equal to 1 in the mask made up of the *integer* arguments (by combining them together with the inclusive or operation).

8.8 Namespaces and Environments

TXR Lisp is a Lisp-2 dialect: it features separate namespaces for functions and variables.

8.8.1 Global Functions and Operator Macros

In **TXR Lisp**, global functions and operator macros coexist, meaning that the same symbol can be defined as both a macro and a function.

There is a global namespace for functions, into which functions can be introduced with the `defun` macro. The global function environment can be inspected and modified using the `symbol-function` accessor.

There is a global namespace for macros, into which macros are introduced with the `defmacro` macro. The global function environment can be inspected and modified using the `symbol-macro` accessor.

If a name `x` is defined as both a function and a macro, then an expression of the form `(x ...)` is expanded by the macro, whereas an expression of the form `[x ...]` refers to the function. Moreover, the macro can produce a call to the function. The expression `(fun x)` will retrieve the function object.

8.8.2 Global and Dynamic Variables

There is a global namespace for variables also. The operators `defvar` and `defparm` introduce bindings into this namespace. These operators have the side effect of marking a symbol as a special variable, of the symbol are treated as dynamic variables, subject to rebinding. The global variable namespace together with the special dynamic rebinding is called the dynamic environment. The dynamic environment can be inspected and modified using the `symbol-value` accessor.

The operators `defvarl` and `defparml` introduce bindings into the global namespace without marking symbols as special variables. Such bindings are called global lexical variables.

8.8.3 Global Symbol Macros

Symbol macros may be defined over the global variable namespace using `defsymacro`.

Note that whereas a symbol may simultaneously have both a function and macro binding in the global namespace, a symbol may not simultaneously have a variable and symbol macro binding.

8.8.4 Lexical Environments

In addition to global and dynamic namespaces, **TXR Lisp** provides lexically scoped binding for functions, variables, macros, and symbol macros. Lexical variable binding are introduced with `let`, `let*` or various binding macros derived from these. Lexical functions are bound with `flet` and `labels`. Lexical macros

are established with `macrolet` and lexical symbol macros with `symacrolet`.

Macros receive an environment parameter with which they may expand forms in their correct environment, and perform some limited introspection over that environment in order to determine the nature of bindings, or the classification of forms in those environments. This introspection is provided by `lexical-var-p`, `lexical-fun-p`, and `lexical-lisp1-binding`.

Lexical operator macros and lexical functions can also coexist in the following way. A lexical function shadows a global or lexical macro completely. However, the reverse is not the case. A lexical macro shadows only those uses of a function which look like macro calls. This is succinctly demonstrated by the following form:

```
(flet ((foo () 43))
  (macrolet ((foo () 44))
    (list (fun foo) (foo) [foo])))

-> (#<interpreted fun: lambda nil> 44 43)
```

The `(fun foo)` and `[fun]` expressions are oblivious to the macro; the macro expansion process process the symbol `foo` in those contexts. However the form `(foo)` is subject to macro-expansion and replaced with `44`.

If the `flet` and `macrolet` are reversed, the behavior is different:

```
(macrolet ((foo () 44))
  (flet ((foo () 43))
    (list (fun foo) (foo) [foo])))

-> (#<interpreted fun: lambda nil> 43 43)
```

All three forms refer to the function, which lexically shadows the macro.

8.8.5 Pattern Language and Lisp Scope Nesting

TXR Lisp expressions can be embedded in the **TXR** pattern language in various ways. Likewise, the pattern language can be invoked from **TXR Lisp**. This brings about the possibility that Lisp code attempts to access pattern variables bound in the pattern language. The **TXR** pattern language can also attempt to access **TXR Lisp** variables.

The rules are as follows, but they have undergone historic changes. See the COMPATIBILITY section, in particular notes under 138 and 121, and also 124.

A Lisp expression evaluated from the **TXR** pattern language executes in a null lexical environment. The current set of pattern variables captured up to that point by the pattern language are installed as dynamic variables. They shadow any Lisp global variables (whether those are defined by `defvar` or `defvarl`).

In the reverse direction, a variable reference from the **TXR** pattern language searches the pattern variable space first. If a variable doesn't exist there, then the lookup refers to the **TXR Lisp** global variable space. The pattern language doesn't see Lisp lexical variables.

When Lisp code is evaluated from the pattern language, the pattern variable bindings are not only installed as dynamic variables for the sake of their visibility from Lisp, but they are also specially stored in a dynamic environment frame. When **TXR** pattern code is reentered from Lisp, these bindings are picked up from the closest such environment frame, allowing the nested invocation of pattern code to continue with the bindings captured by outer pattern code.

Concisely, in any context in which a symbol has both a binding as a Lisp global variable as well as a pattern variable, that symbol refers to the pattern variable. Pattern variables are propagated through Lisp evaluation into nested invocations of the pattern language.

The pattern language can also reference Lisp variables using the `@` prefix, which is a consequence of that prefix introducing an expression that is evaluated as Lisp, the name of a variable being such an expression.

9 LISP OPERATOR, FUNCTION AND MACRO REFERENCE

9.1 Conventions

The following sections list all of the special operators, macros and functions in **TXR Lisp**.

In these sections, syntax is indicated using these conventions:

word A symbol in *fixed-width-italic* font denotes some syntactic unit: it may be a symbol or compound form. The syntactic unit is explained in the corresponding Description section.

`{syntax}*` *word**
This indicates a repetition of zero or more of the given syntax enclosed in the braces or syntactic unit. The curly braces may be omitted if the scope of the `*` is clear.

`{syntax}+` *word+*
This indicates a repetition of one or more of the given syntax enclosed in the braces or syntactic unit. The curly braces may be omitted if the scope of the + is clear.

`{syntax | syntax | ...}`
This indicates a single, mandatory element, which is selected from among the indicated alternatives. May be combined with + or `*` repetition.

`[syntax] [`*word*`]`
Square brackets indicate optional syntax.

`[syntax | syntax | ...]`
Square brackets containing piped elements indicate an optional element, which, if present, must be chosen from among the indicated alternatives.

`'[' ']'`
The quoted square brackets indicate literal brackets which appear in the syntax, which they do without quotes. For instance `'['foo [ bar ]']'` is a pattern denotes the two possible expressions `[foo]` and `[foo bar]`.

`syntax -> `*result*
The arrow notation is used in examples to indicate that the evaluation of the given syntax produces a value, whose printed representation is *result*.

9.2 Form Evaluation

A compound expression with a symbol as its first element, if intended to be evaluated, denotes either an operator invocation or a function call. This depends on whether the symbol names an operator or a function.

When the form is an operator invocation, the interpretation of the meaning of that form is under the complete control of that operator.

If the compound form is a function call, the remaining forms, if any, denote argument expressions to the function. They are evaluated in left-to-right order to produce the argument values, which are passed to the function. An exception is thrown if there are not enough arguments, or too many. Programs can define named functions with the defun operator

Some operators are macros. There exist predefined macros in the library, and macro operators can also be user-defined using the macro-defining operator `defmacro`. Operators that are not macros are called special operators.

Macro operators work as functions which are given the source code of the form. They analyze the form, and translate it to another form which is substituted in their place. This happens during a code walking phase called the expansion phase, which is applied to each top-level expression prior to evaluation. All macros occurring in a form are expanded in the expansion phase, and subsequent evaluation takes place on a structure which is devoid of macros. All that remains are the executable forms of special operators, function calls, symbols denoting either variables or themselves, and atoms such as numeric and string literals.

Special operators can also perform code transformations during the expansion phase, but that is not considered macroexpansion, but rather an adjustment of the representation of the operator into an required executable form. In effect, it is post-macro compilation phase.

Note that Lisp forms occurring in **TXR** pattern language are not individual top-level forms. Rather, the entire **TXR** query is parsed at the same time, and the macros occurring in its Lisp forms are expanded at that time.

9.2.1 Operator `quote`

Syntax:

```
(quote form)
```

Description:

The `quote` operator, when evaluated, suppresses the evaluation of *`form`*, and instead returns *`form`* itself as an object. For example, if *`form`* is a symbol *`sym`*, then the value of `(quote sym)` is *`sym`* itself. Without `quote`, *`sym`* would evaluate to the value held by the variable which is named *`sym`*, or else throw an error if there is no such variable. The `quote` operator never raises an error, if it is given exactly one argument, as required.

The notation `'obj` is translated to the object `(quote obj)` providing a shorthand for quoting. Likewise, when an object of the form `(quote obj)` is printed, it appears as `'obj`.

Example:

```
;; yields symbol a itself, not value of variable a
(quote a) -> a

;; yields three-element list (+ 2 2), not 4.
(quote (+ 2 2)) -> (+ 2 2)
```

9.3 Variable Binding

Variables are associations between symbols and storage locations which hold values. These associations are called *bindings*.

Bindings are held in a context called an *environment*.

Lexical environments hold local variables, and nest according to the syntactic structure of the program. Lexical bindings are always introduced by a some form known as a *binding construct*, and the corresponding environment is instantiated during the evaluation of that construct. There also exist bindings outside of any binding construct, in the so-called *global environment*. Bindings in the global environment can be temporarily shadowed by lexically-established binding in the *dynamic environment*. See the Special Variables section above.

Certain special symbols cannot be used as variable names, namely the symbols `t` and `nil`, and all of the keyword symbols (symbols in the keyword package), which are denoted by a leading colon. When any of these symbols is evaluated as a form, the resulting value is that symbol itself. It is said that these special symbols are self-evaluating or self-quoting, similarly to all other atom objects such as numbers or strings.

When a form consisting of a symbol, other than the above special symbols, is evaluated, it is treated as a variable, and yields the value of the variable's storage location. If the variable doesn't exist, an exception is thrown.

Note: symbol forms may also denote invocations of symbol macros. (See the operators `defsymacro` and `symacrolet`). All macros, including symbol macros, which occur inside a form are fully expanded prior to the evaluation of a form, therefore evaluation does not consider the possibility of a symbol being a symbol macro.

9.3.1 Operator `defvar` and macro `defparm`

Syntax:

```
(defvar sym [value])
(defparm sym value)
```

Description:

The `defvar` operator binds a name in the variable namespace of the global environment. Binding a name means creating a binding: recording, in some namespace of some environment, an association between a name and some named entity. In the case of a variable binding, that entity is a storage location for a value. The value of a variable is that which has most recently been written into the storage location, and is also said to be a value of the binding, or stored in the binding.

If the variable named *sym* already exists in the global environment, the form has no effect; the *value* form is not evaluated, and the value of the variable is unchanged.

If the variable does not exist, then a new binding is introduced, with a value given by evaluating the *value* form. If the form is absent, the variable is initialized to `nil`.

The *value* form is evaluated in the environment in which the `defvar` form occurs, not necessarily in the global environment.

The symbols `t` and `nil` may not be used as variables, nor can they be keyword symbols (symbols denoted by a leading colon).

In addition to creating a binding, the `defvar` operator also marks *sym* as the name of a special variable. This changes what it means to bind that symbol in a lexical binding construct such as the `let` operator, or a function parameter list. See the section "Special Variables" far above.

The `defparm` macro behaves like `defvar` when a variable named *sym* doesn't already exist.

If *sym* already denotes a variable binding in the global namespace, `defparm` evaluates the *value* form and assigns the resulting value to the variable.

The following equivalence holds:

```
(defparm x y)  <-->  (prog1 (defvar x) (set x y))
```

The `defvar` and `defparm` forms return *sym*.

9.3.2 Macros `defvarl` and `defparml`

Syntax:

```
(defvarl sym [value])
(defparml sym value)
```

Description:

The `defvarl` and `defparml` macros behave, respectively, almost exactly like `defvar` and `defparm`.

The difference is that these operators do not mark *sym* as special.

If a global variable *sym* does not previously exist, then after the evaluation of either of these forms `(boundp sym)` is true, but `(special-var-p sym)` isn't.

If *sym* had been already introduced as a special variable, it stays that way after the evaluation of `defvarl` or `defparml`.

9.3.3 Operators `let` and `let*`

Syntax:

```
(let ({sym | (sym init-form)}*) body-form*)
(let* ({sym | (sym init-form)}*) body-form*)
```

Description:

The `let` and `let*` operators introduce a new scope with variables and evaluate forms in that scope. The operator symbol, either `let` or `let*`, is followed by a list which can contain any mixture of *sym* or `(sym init-form)` pairs. Each *sym* must be a symbol, and specifies the name of variable to be instantiated and initialized.

The `(sym init-form)` variant specifies that the new variable *sym* receives an initial value from the evaluation of *init-form*. The plain *sym* variant specifies a variable which is initialized to `nil`. The *init-form*s are evaluated in order, by both `let` and `let*`.

The symbols `t` and `nil` may not be used as variables, and neither can be keyword symbols: symbols denoted by a leading colon.

The difference between `let` and `let*` is that in `let*`, later `init-form`s are in scope of the variables established by earlier variables in the same `let*` construct. In plain `let`, the *init-form*s are evaluated in a scope which does not include any of the variables.

When the variables are established, the *body-form*s are evaluated in order. The value of the last *body-form* becomes the return value of the `let`. If there are no *body-form*s, then the return value `nil` is produced.

The list of variables may be empty.

The list of variables may contain duplicate *sym*s if the operator is `let*`. In that situation, a given *init-form* has in scope the rightmost duplicate of any given *sym* that has been previously established. The *body-form*s have in scope the rightmost duplicate of any *sym* in the construct. Therefore, the following form calculates the value 3:

```
(let* ((a 1)
       (a (succ a))
       (a (succ a)))
  a)
```

Each duplicate is a separately instantiated binding, and may be independently captured by a lexical closure placed in a subsequent `init-form`:

```
(let* ((a 0)
       (f1 (lambda () (inc a)))
       (a 0)
       (f2 (lambda () (inc a))))
  (list [f1] [f1] [f1] [f2] [f2] [f2]))

--> (1 2 3 1 2 3)
```

The preceding example shows that there are two mutable variables named `a` in independent scopes, each respectively captured by the separate closures `f1` and `f2`. Three calls to `f1` increment the first `a` while the second `a` retains its initial value.

Under `let`, the behavior of duplicate variables is unspecified.

Implementation note: the **TXR** compiler diagnoses and rejects duplicate symbols in `let` whereas the interpreter ignores the situation.

When the names of a special variables is specified in `let` or `let*` remain, a new binding is created for them in the dynamic environment, rather than the lexical environment. In `let*`, later *init-form*s are evaluated in a dynamic scope in which previous dynamic variables are established, and later dynamic variables are not yet established. A special variable may appear multiple times in a `let*`, just like a lexical variable. Each duplicate occurrence extends the dynamic environment with a new dynamic binding. All these dynamic environments are removed when the `let` or `let*` form terminates. Dynamic environments aren't captured by lexical closures, but are captured in delimited continuations.

Examples:

```
(let ((a 1) (b 2)) (list a b)) -> (1 2)
(let* ((a 1) (b (+ a 1))) (list a b (+ a b))) -> (1 2 3)
(let ()) -> nil
(let (:a nil)) -> error, :a and nil can't be used as variables
```

9.4 Functions

9.4.1 Operator `defun`

Syntax:

```
(defun name (param* [: opt-param*] [. rest-param])
   body-form)
```

Description:

The `defun` operator introduces a new function in the global function namespace. The function is similar to a lambda, and has the same parameter syntax and semantics as the `lambda` operator.

Note that the above syntax synopsis describes only the canonical parameter syntax which remains after parameter list macros are expanded. See the section Parameter List Macros.

Unlike in `lambda`, the *body-form*s of a `defun` are surrounded by a block. The name of this block is the same as the name of the function, making it possible to terminate the function and return a value using `(return-from` *name value*`)`. For more information, see the definition of the block operator.

A function may call itself by name, allowing for recursion.

The special symbols `t` and `nil` may not be used as function names. Neither can keyword symbols.

It is possible to define methods as well as macros with `defun`, as an alternative to the `defmeth` and `defmacro` forms.

To define a method, the syntax `(meth` *type name*`)` should be used as the argument to the *name* parameter. This gives rise to the syntax `(defun (meth` *type name*`)` *args form**`)` which is equivalent to the `(defmeth` *type name args form**`)` syntax.

Macros can be defined using `(macro` *name*`)` as the *name* parameter of `defun`. This way of defining a macro doesn't support destructuring; it defines the expander as an ordinary function with an ordinary argument list. To work, the function must accept two arguments: the entire macro call form that is to be expanded, and the macro environment. Thus, the macro definition syntax is `(defun (macro` *name*`)` *form env form**`)` which is equivalent to the `(defmacro` *name* `(:form` *form* `:env` *env*`)` *form**`)` syntax.

Dialect Note:

In ANSI Common Lisp, keywords may be used as function names. In TXR Lisp, they may not.

Dialect Note:

A function defined by `defun` may coexist with a macro defined by `defmacro`. This is not permitted in ANSI Common Lisp.

9.4.2 Operator `lambda`

Syntax:

```
(lambda (param* [: opt-param*] [. rest-param])
   body-form)
(lambda rest-param
   body-form)
```

Description:

The `lambda` operator produces a value which is a function. Like in most other Lisps, functions are objects in **TXR Lisp**. They can be passed to functions as arguments, returned from functions, aggregated into lists, stored in variables, etc.

Note that the above syntax synopsis describes only the canonical parameter syntax which remains

after parameter list macros are expanded. See the section Parameter List Macros.

The first argument of `lambda` is the list of parameters for the function. It may be empty, and it may also be an improper list (dot notation) where the terminating atom is a symbol other than `nil`. It can also be a single symbol.

The second and subsequent arguments are the forms making up the function body. The body may be empty.

When a function is called, the parameters are instantiated as variables that are visible to the body forms. The variables are initialized from the values of the argument expressions appearing in the function call.

The dotted notation can be used to write a function that accepts a variable number of arguments. There are two ways write a function that accepts only a variable argument list and no required arguments:

```
(lambda (. rest-param) ...)
(lambda rest-param ...)
```

(These notations are syntactically equivalent because the list notation `(. X)` actually denotes the object *X* which isn't wrapped in any list).

The keyword symbol **:** (colon) can appear in the parameter list. This is the symbol in the keyword package whose name is the empty string. This symbol is treated specially: it serves as a separator between required parameters and optional parameters. Furthermore, the **:** symbol has a role to play in function calls: it can be specified as an argument value to an optional parameter by which the caller indicates that the optional argument is not being specified. It will be processed exactly that way.

An optional parameter can also be written in the form `(name expr [sym])`. In this situation, if the call does not specify a value for the parameter, or specifies a value as the **:** (colon) keyword symbol, then the parameter takes on the value of the expression *expr*. This expression is only evaluated when its value is required.

If *sym* is specified, then *sym* will be introduced as an additional binding with a Boolean value which indicates whether or not the optional parameter had been specified by the caller.

Each *expr* that is evaluated is evaluated in an environment in which all of the previous parameters are visible, in addition to the surrounding environment of the `lambda`. For instance:

```
(let ((default 0))
  (lambda (str : (end (length str)) (counter default))
    (list str end counter)))
```

In this `lambda`, the initializing expression for the optional parameter end is `(length str)`, and the *str* variable it refers to is the previous argument. The initializer for the optional variable counter is the expression default, and it refers to the binding established by the surrounding let. This reference is captured as part of the `lambda`'s lexical closure.

Keyword symbols, and the symbols `t` and `nil` may not be used as parameter names. The behavior is unspecified if the same symbol is specified more than once anywhere in the parameter list, whether as a parameter name or as the indicator *sym* in an optional parameter or any combination.

Implementation note: the **TXR** compiler diagnoses and rejects duplicate symbols in `lambda` whereas the interpreter ignores the situation.

Note: it is not always necessary to use the `lambda` operator directly in order to produce an anonymous function.

In situations when `lambda` is being written in order to simulate partial evaluation, it may be possible to instead make use of the `op` macro. For instance the function `(lambda (. args) [apply + a args])` which adds the values of all of its arguments together, and to the lexically captured variable `a` can be written more succinctly as `(op + a)`. The `op` operator is the main representative of a family of operators: `lop`, `ap`, `ip`, `do`, `ado`, `opip` and `oand`.

In situations when functions are simply combined together, the effect may be achieved using some of the available functional combinators, instead of a `lambda`. For instance chaining together functions as in `(lambda (x) (square (cos x)))` is achievable using the `chain` function: `[chain cos square]`. The `opip` operator can also be used: `(opip cos square)`. Numerous combinators are available; see the section Partial Evaluation and Combinators.

When a function is needed which accesses an object, there are also alternatives. Instead of `(lambda (obj) obj.slot)` and `(lambda (obj arg) obj.(slot arg))`, it is simpler to use the `.slot` and `.(slot arg)` notations. See the section Unbound Referencing Dot. Also see the functions `umethod` and `uslot` as well as the related convenience macros `umeth` and `usl`.

If a function is needed which partially applies, to some arguments, a method invoked on a specific object, the `method` function or `meth` macro may be used. For instance, instead of `(lambda (arg) obj.(method 3 arg))`, it is possible to write `(meth obj 3)` except that the latter produces a variadic function.

Examples:

The following expression returns a function which captures the variable `counter`. Whenever the returned function is called, it increments `counter` by one, and returns the incremented value.

```
(let ((counter 0))
  (lambda () (inc counter)))
```

The following produces a variadic function which requires at least two arguments. The third and subsequent arguments are aggregated into a list passed as the single parameter `z`:

```
(lambda (x y . z) (list 'my-arguments-are x y z))
```

A variadic function with no required arguments. The parameter name for the received arguments appears alone in place of the parameter list.

```
(lambda args (list 'my-list-of-arguments args))
```

Same as the previous example, using a dotted notation specific to **TXR Lisp**.

```
(lambda (. args) (list 'my-list-of-arguments args))
```

Note that `(. args)` is just a written notation equivalent to `args` and not a different object structure.

Optional arguments:

```
[(lambda (x : y) (list x y)) 1] -> (1 nil)
[(lambda (x : y) (list x y)) 1 2] -> (1 2)
```

Passing `:` (colon symbol) to request default value of optional parameter:

```
[(lambda (x : (y 42) z) (list x y z)) 1 2 3] -> (1 2 3)
[(lambda (x : (y 42) z) (list x y z)) 1 : 3] -> (1 42 3)
[(lambda (x : (y 42) z) (list x y z)) 1] -> (1 42 nil)
```

Presence-indicating variable accompanying optional parameter:

```
[(lambda (x : (y 42 have-y)) (list x y have-y)) 1 2]
-> (1 2 t)

[(lambda (x : (y 42 have-y)) (list x y have-y)) 1]
-> (1 42 nil)

;; defaulting via : is indistinguishable from missing
[(lambda (x : (y 42 have-y)) (list x y have-y)) 1 :]
-> (1 42 nil)
```

9.4.3 Macros `flet` and `labels`

Syntax:

```
(flet ({(name param-list function-body-form*)}*)
   body-form*)
(labels ({(name param-list function-body-form*)}*)
   body-form*)
```

Description:

The `flet` and `labels` macros bind local, named functions in the lexical scope.

Note that the above syntax synopsis describes only the canonical parameter syntax which remains after parameter list macros are expanded. See the section Parameter List Macros.

The difference between `flet` and `labels` is that a function defined by `labels` can see itself, and therefore recurse directly by name. Moreover, if multiple functions are defined by the same labels construct, they all have each other's names in scope of their bodies. By contrast, a `flet`-defined function does not have itself in scope and cannot recurse. Multiple functions in the same `flet` do not have each other's names in their scopes.

More formally, the *`function-body-form`*s and *`param-list`* of the functions defined by `labels` are in a scope in which all of the function names being defined by that same `labels` construct are visible.

Under both `labels` and `flet`, the local functions that are defined are lexically visible to the main *`body-form`*s.

Note that `labels` and `flet` are properly scoped with regard to macros. During macro expansion, function bindings introduced by these macro operators shadow macros defined by `macro-let` and `defmacro`.

Furthermore, function bindings introduced by `labels` and `flet` also shadow symbol macros defined by `symacrolet`, when those symbol macros occur as arguments of a `dwim` form.

See also: the `macrolet` operator.

Dialect Note:

The `flet` and `labels` macros do not establish named blocks around the body forms of the local functions which they bind. This differs from ANSI Common Lisp, whose local function have implicit named blocks, allowing for `return-from` to be used.

Examples:

```
;; Wastefully slow algorithm for determining evenness.
;; Note:
;; - mutual recursion between labels-defined functions
;; - inner is-even bound by labels shadows the outer
;;   one bound by defun so the (is-even n) call goes
;;   to the local function.

(defun is-even (n)
 (labels ((is-even (n)
            (if (zerop n) t (is-odd (- n 1))))
          (is-odd (n)
            (if (zerop n) nil (is-even (- n 1)))))
   (is-even n)))
```

9.4.4 Function `call`

Syntax:

```
(call function argument*)
```

Description:

The `call` function invokes *`function`*, passing it the given arguments, if any.

Examples:

Apply `lambda` to `1 2` arguments, adding them to produce `3`:

```
(call (lambda (a b) (+ a b)) 1 2)
```

Useless use of `call` on a named function; equivalent to `(list 1 2)`:

```
(call (fun list) 1 2)
```

9.4.5 Functions `apply` and `iapply`

Syntax:

```
(apply function [arg* trailing-args])
(iapply function [arg* trailing-args])
```

Description:

The `apply` function invokes *`function`*, optionally passing to it an argument list. The return value of the `apply` call is that of *`function`*.

If no arguments are present after *`function`*, then *`function`* is invoked without arguments.

If one argument is present after *function*, then it is interpreted as *trailing-args*. If this is a sequence (a list, vector or string), then the elements of the sequence are passed as individual arguments to *function*. If *trailing-args* is not a sequence, then *function* is invoked with an improper argument list, terminated by the *trailing-args* atom.

If two or more arguments are present after *function*, then the last of these arguments is interpreted as *trailing-args*. The previous arguments represent leading arguments which are applied to *function*, prior to the arguments taken from *trailing-args*.

Note that if *trailing-args* value is an atom or an improper list, the function is then invoked with an improper argument list. Only a variadic function may be invoked with an improper argument list. Moreover, all of the function's required and optional parameters must be satisfied by elements of the improper list, such that the terminating atom either matches the *rest-param* directly (see the `lambda` operator) or else the *rest-param* receives an improper list terminated by that atom. To treat the terminating atom of an improper list as an ordinary element which can satisfy a required or optional function parameter, the `iapply` function may be used, described next.

The `iapply` function ("improper apply") is similar to `apply`, except with regard to the treatment of *trailing-args*. Firstly, under `iapply`, if *trailing-args* is an atom other than `nil` (possibly a sequence, such as a vector or string), then it is treated as an ordinary argument: *function* is invoked with a proper argument list, whose last element is *trailing-args*. Secondly, if *trailing-args* is a list, but an improper list, then the terminating atom of *trailing-args* becomes an individual argument. This terminating atom is not split into multiple arguments, even if it is a sequence. Thus, in all possible cases, `iapply` treats an extra non-`nil` atom as an argument, and never calls *function* with an improper argument list.

Examples:

```
;; '(1 2 3) becomes arguments to list, thus (list 1 2 3).
(apply (fun list) '(1 2 3)) -> (1 2 3)

;; this effectively invokes (list 1 2 3 4)
(apply (fun list) 1 2 '(3 4)) -> (1 2 3 4)

;; this effectively invokes (list 1 2 . 3)
(apply (fun list) 1 2 3)) -> (1 2 . 3)

;; "abc" is separated into characters
;; which become arguments of list
(apply (fun list) "abc") -> (#\a #\b #\c)
```

Dialect Note:

Note that some uses of this function that are necessary in other Lisp dialects are not necessary in **TXR Lisp**. The reason is that in **TXR Lisp**, improper list syntax is accepted as a compound form, and performs application:

```
(foo a b . x)
```

Here, the variables `a` and `b` supply the first two arguments for `foo`. In the dotted position, `x` must evaluate to a list or vector. The list or vector's elements are pulled out and treated as additional arguments for `foo`. This syntax can only be used if `x` is a symbolic form or an atom. It cannot be a compound form, because `(foo a b . (x))` and `(foo a b x)` are equivalent structures.

9.4.6 Operator `fun`

Syntax:

```
(fun function-name)
```

Description:

The `fun` operator retrieves the function object corresponding to a named function in the current lexical environment.

The *`function-name`* may be a symbol denoting a named function: a built in function, or one defined by `defun`.

The *`function-name`* may also take any of the forms specified in the description of the `func-get-name` function. If such a *`function-name`* refers to a function which exists, then the `fun` operator yields that function.

Note: the `fun` operator does not see macro bindings via their symbolic names with which they are defined by `defmacro`. However, the name syntax `(macro name)` may be used to refer to macros. This syntax is documented in the description of `func-get-name`. It is also possible to retrieve a global macro expander using the function `symbol-macro`.

9.4.7 Operator `dwim`

Syntax:

```
(dwim argument*)
'['argument*']'
(set (dwim obj-place index [alt]) new-value)
(set '['obj-place index [alt]']' new-value)
```

Description:

The `dwim` operator's name is an acronym: DWIM may be taken to mean "Do What I Mean", or alternatively, "Dispatch, in a Way that is Intelligent and Meaningful".

The notation `[...]` is a shorthand which denotes `(dwim ...)`.

Note that since the `[` and `]` are used in this document for indicating optional syntax, in the above Syntax synopsis the quoted notation `'['` and `']'` denotes bracket tokens which literally appear in the syntax.

The `dwim` operator takes a variable number of arguments, which are treated as expressions to be individually macro-expanded and evaluated, using the same rules.

This means that the first argument isn't a function name, but an ordinary expression which can simply compute a function object (or, more generally, a callable object).

Furthermore, for those arguments of `dwim` which are symbols (after all macro-expansion is performed), the evaluation rules are altered. For the purposes of resolving symbols to values, the function and variable binding namespaces are considered to be merged into a single space, creating a situation that is similar to a Lisp-1 style dialect.

This special Lisp-1 evaluation is not recursively applied. All arguments of `dwim` which, after macro expansion, are not symbols are evaluated using the normal Lisp-2 evaluation rules. Thus, the DWIM operator must be used in every expression where the Lisp-1 rules for reducing symbols to values are desired.

If a symbol has bindings both in the variable and function namespace in scope, and is referenced by a dwim argument, this constitutes a conflict which is resolved according to two rules. When nested scopes are concerned, then an inner binding shadows an outer binding, regardless of their kind. An inner variable binding for a symbol shadows an outer or global function binding, and vice versa.

If a symbol is bound to both a function and variable in the global namespace, then the variable binding is favored.

Macros do not participate in the special scope conflation, with one exception. What this means is that the space of symbol macros is not folded together with the space of operator macros. An argument of `dwim` that is a symbol might be symbol macro, variable or function, but it cannot be interpreted as the name of a operator macro.

The exception is this: from the perspective of a `dwim` form, function bindings can shadow symbol macros. If a function binding is defined in an inner scope relative to a symbol macro for the same symbol, using `flet` or `labels`, the function hides the symbol macro. In other words, when macro expansion processes an argument of a `dwim` form, and that argument is a symbol, it is treated specially in order to provide a consistent name lookup behavior. If the innermost binding for that symbol is a function binding, it refers to that function binding, even if a more outer symbol macro binding exists, and so the symbol is not expanded using the symbol macro. By contrast, in an ordinary form, a symbolic argument never resolves to a function binding. The symbol refers to either a symbol macro or a variable, whichever is nested closer.

If, after macro expansion, the leftmost argument of the `dwim` is the name of a special operator or macro, the `dwim` form doesn't denote an invocation of that operator or macro. A `dwim` form is an invocation of the `dwim` operator, and the leftmost argument of that operator, if it is a symbol, is treated as a binding to be resolved in the variable or function namespace, like any other argument. Thus `[if x y]` is an invocation of the `if` function, not the `if` operator.

How many arguments are required by the `dwim` operator depends on the type of object to which the first argument expression evaluates. The possibilities are:

`[`*`function argument*`*`]`
: Call the given function object with the given arguments.

`[`*`symbol argument*`*`]`
: If the first expression evaluates to a symbol, that symbol is resolved in the function namespace, and then the resulting function, if found, is called with the given arguments.

`[`*`sequence index`*`]`
: Retrieve an element from *`sequence`*, at the specified *`index`*, which is a nonnegative integer.

  This form is also a syntactic place. If a value is stored to this place, it replaces the element.

  The place may also be deleted, which has the effect of removing the element from the sequence, shifting the elements at higher indices, if any, down one element position, and shortening the sequence by one. If the place is deleted, and if *`sequence`* is a list, then the *`sequence`* form itself must be a place.

  This form is implemented using the `ref` accessor such that, except for the argument evaluation semantics of the DWIM brackets, it is equivalent to using the `(ref` *`sequence`*

index) syntax.

[*sequence from-index..to-below-index*]
Retrieve the specified range of elements. The range of elements is specified in the `from` and `to` fields of a range object. The `..` (dotdot) syntactic sugar denotes the construction of the range object via the `rcons` function. See the section on Range Indexing below.

This form is also a syntactic place. Storing a value in this place has the effect of replacing the subsequence with a new subsequence. Deleting the place has the effect of removing the specified subsequence from *sequence*. If *sequence* is a list, then the *sequence* form must itself be a place. The *new-value* argument in a range assignment can be a string, vector or list, regardless of whether the target is a string, vector or list. If the target is a string, the replacement sequence must be a string, or a list or vector of characters.

The semantics is implemented using the `sub` accessor, such that the following equivalence holds:

```
[seq from..to]  <--> (sub seq from..to)
```

For this reason, *sequence* may be any object that is iterable by `iter-begin`.

[*sequence index-list*]
Elements specified by *index-list*, which may be a list or vector, are extracted from *sequence* and returned as a sequence of the same kind as *sequence*.

This form is equivalent to (`select` *sequence where-index*) except when the target of an assignment operation.

This form is a syntactic place if *sequence* is one. If a sequence is assigned to this place, then elements of the sequence are distributed to the specified locations.

The following equivalences hold between index-list-based indexing and the `select` and `replace` functions, except that `set` always returns the value assigned, whereas `replace` returns its first argument:

```
[seq idx-list] <--> (select seq idx-list)

(set [seq idx-list] new) <--> (replace seq new idx-list)
```

Note that unlike the select function, this does not support [*hash index-list*] because since hash keys may be lists, that syntax is indistinguishable from a simple hash lookup where *index-list* is the key.

[*hash key* [*alt*]]
Retrieve a value from the hash table corresponding to *key*, or else return *alt* if there is no such entry. The expression *alt* is always evaluated, whether or not its value is used.

[*search-tree key*]
Retrieves an element from the search tree as if by applying the `tree-lookup` function to *key*.

[*search-tree from-key..to-below-key*]
: Retrieves a list of elements from the search tree as if by evaluating the `(sub-tree` *`search-tree from-key to-below-key`*`)` expression.

[*regex* [*start* [*from-end*]] *string*]
: Determine whether regular expression *regex* matches *string*, and in that case return the (possibly empty) leftmost matching substring. Otherwise, return `nil`.

  If *start* is specified, it gives the starting position where the search begins, and if *from-end* is given, and has a value other than `nil`, it specifies a search from right to left. These optional arguments have the same conventions and semantics as their equivalents in the `search-regst` function.

  Note that *string* is always required, and is always the rightmost argument.

[*struct arg**]
: The structure instance *struct* is inquired whether it supports a method named by the symbol *lambda*. If so, that method is invoked on the object. The method receives *struct* followed by the value of every *arg*. If this form is used as a place, then the object must support a `lambda-set` method.

[*carray index*]

[*carray from-index..to-below-index*]
: Element and range indexing is possible on object of type `carray` which manipulate arrays in a foreign ("C language") representation, and are closely associated with the Foreign Function Interface (FFI). Just like in the case of sequences, the semantics of referencing `carray` objects with the bracket notation is based on the functions `ref`, `refset`, `sub` and `replace`. These, in turn, rely on the specialized functions. `carray-ref`, `carray-refset`, `carray-sub` and `carray-replace`.

[*buf index*]
: Indexing is supported for objects of type `buf`. This provides a way to access and store the individual bytes of a buffer.

Range Indexing:

Vector and list range indexing is based from zero, meaning that the first element is numbered zero, the second one and so on. zero. Negative values are allowed; the value `-1` refers to the last element of the vector or list, and `-2` to the second last and so forth. Thus the range `1 .. -2` means "everything except for the first element and the last two".

The symbol `t` represents the position one past the end of the vector, string or list, so `0..t` denotes the entire list or vector, and the range `t..t` represents the empty range just beyond the last element. It is possible to assign to `t..t`. For instance:

```
(defvar list '(1 2 3))
(set [list t..t] '(4)) ;; list is now (1 2 3 4)
```

The value zero has a "floating" behavior when used as the end of a range. If the start of the range is a negative value, and the end of the range is zero, the zero is interpreted as being the position past the end of the sequence, rather than the first element. For instance the range `-1..0` means the same thing as `-1..t`. Zero at the start of a range always means the first element, so that `0..-1` refers to all the elements except for the last one.

Notes:

The dwim operator allows for a Lisp-1 flavor of programming in **TXR Lisp**, which is principally a Lisp-2 dialect.

A Lisp-1 dialect is one in which an expression like `(a b)` treats both a and b as expressions subject to the same evaluation rules—at least, when `a` isn't an operator or an operator macro. This means that the symbols `a` and `b` are resolved to values in the same namespace. The form denotes a function call if the value of variable `a` is a function object. Thus in a Lisp-1, named functions do not exist as such: they are just variable bindings. In a Lisp-1, the form `(car 1)` means that there is a variable called `car`, which holds a function, which is retrieved from that variable and the argument `1` is applied to it. In the expression `(car car)`, both occurrences of `car` refer to the variable, and so this form applies the `car` function to itself. It is almost certainly meaningless. In a Lisp-2 `(car 1)` means that there is a function called `car`, in the function namespace. In the expression `(car car)` the two occurrences refer to different bindings: one is a function and the other a variable. Thus there can exist a variable `car` which holds a cons-cell object, rather than the `car` function, and the form makes sense.

The Lisp-1 approach is useful for functional programming, because it eliminates cluttering occurrences of the call and fun operators. For instance:

```
;; regular notation

(call foo (fun second) '((1 a) (2 b)))

;; [] notation

[foo second '((1 a) (2 b))]
```

Lisp-1 dialects can also provide useful extensions by giving a meaning to objects other than functions in the first position of a form, and the `dwim/[...]` syntax does exactly this.

TXR Lisp is a Lisp-2 because Lisp-2 also has advantages. Lisp-2 programs which use macros naturally achieve hygiene because lexical variables do not interfere with the function namespace. If a Lisp-2 program has a local variable called `list`, this does not interfere with the hidden use of the function `list` in a macro expansion in the same block of code. Lisp-1 dialects have to provide hygienic macro systems to attack this problem. Furthermore, even when not using macros, Lisp-1 programmers have to avoid using the names of functions as lexical variable names, if the enclosing code might use them.

The two namespaces of a Lisp-2 also naturally accommodate symbol macros and operator macros. Whereas functions and variables can be represented in a single namespace readily, because functions are data objects, this is not so with symbol macros and operator macros, the latter of which are distinguished syntactically by their position in a form. In a Lisp-1 dialect, given `(foo bar)`, either of the two symbols could be a symbol macro, but only `foo` can possibly be an operator macro. Yet, having only a single namespace, a Lisp-1 doesn't permit `(foo foo)`, where `foo` is simultaneously a symbol macro and an operator macro, though the situation is unambiguous by syntax even in Lisp-1. In other words, Lisp-1 dialects do not entirely remove the special syntactic recognition given to the leftmost position of a compound form, yet at the same time they prohibit the user from taking full advantage of it by providing only one namespace.

TXR Lisp provides the "best of both worlds": the DWIM brackets notation provides a model of Lisp-1 computation that is purer than Lisp-1 dialects (since the leftmost argument is not given any special syntactic treatment at all) while the Lisp-2 foundation provides a traditional Lisp environment with its "natural hygiene".

9.4.8 Function `functionp`

Syntax:

```
(functionp obj)
```

Description:

The `functionp` function returns `t` if *`obj`* is a function, otherwise it returns `nil`.

9.4.9 Function `copy-fun`

Syntax:

```
(copy-fun function)
```

Description:

The `copy-fun` function produces and returns a duplicate of *`function`*, which must be a function.

A duplicate of a function is a distinct function object not `eq` to the original function, yet which accepts the same arguments and behaves exactly the same way as the original.

If a function contains no captured environment, then a copy made of that function by `copy-fun` is indistinguishable from the original function in every regard, except for being a distinct object that compares unequal to the original under the `eq` function.

If a function contains a captured environment, then a copy of that function made by `copy-fun` has its own copy of that environment. If the copied function changes the values of captured lexical variables, the original function is not affected by these changes and vice versa.

The entire lexical environment is copied; the copy and original function do not share any portion of the environment at any level of nesting.

9.5 Sequencing, Selection and Iteration

9.5.1 Operators/functions `progn` and `prog1`

Syntax:

```
(progn form*)
(prog1 form*)
```

Description:

The `progn` operator evaluates each *`form`* in left-to-right order, and returns the value of the last form. The value of the form `(progn)` is `nil`.

The `prog1` operator evaluates each *`form`* in left-to-right order, and returns the value of the first form. The value of the form `(prog1)` is `nil`.

Various other operators such as `let` also arrange for the evaluation of a body of forms, the value of the last of which is returned. These operators are said to feature an implicit `progn`.

These special operators are also functions. The `progn` function accepts zero or more arguments. It returns its last argument, or `nil` if called with no arguments. The `prog1` function likewise accepts zero or more arguments. It returns its first argument, or `nil` if called with no arguments.

Dialect Notes:

In ANSI Common Lisp, `prog1` requires at least one argument. Neither `prog` nor `prog1` exist as functions.

9.5.2 Macro/function `prog2`

Syntax:

```
(prog2 form*)
```

Description:

The `prog2` evaluates each *form* in left-to-right order. The value is that of the second form, if present, otherwise it is `nil`.

The form `(prog2 1 2 3)` yields `2`. The value of `(prog2 1 2)` is also `2`; `(prog2 1)` and `(prog2)` yield `nil`.

The `prog2` symbol also has a function binding. The `prog2` function accepts any number of arguments. If invoked with at least two arguments, it returns the second one. Otherwise it returns `nil`.

Dialect Notes:

In ANSI Common Lisp, `prog2` requires at least two arguments. It does not exist as a function.

9.5.3 Operator `cond`

Syntax:

```
(cond {(test form*)}*)
```

Description:

The `cond` operator provides a multi-branching conditional evaluation of forms. Enclosed in the cond form are groups of forms expressed as lists. Each group must be a list of at least one form.

The forms are processed from left to right as follows: the first form, *test*, in each group is evaluated. If it evaluates true, then the remaining forms in that group, if any, are also evaluated. Processing then terminates and the result of the last form in the group is taken as the result of cond. If *test* is the only form in the group, then result of *test* is taken as the result of `cond`.

If the first form of a group yields `nil`, then processing continues with the next group, if any. If all form groups yield `nil`, then the cond form yields `nil`. This holds in the case that the syntax is empty: `(cond)` yields `nil`.

9.5.4 Macros `caseq`, `caseql` and `casequal`

Syntax:

```
(caseq test-form normal-clause* [else-clause])
(caseql test-form normal-clause* [else-clause])
(casequal test-form normal-clause* [else-clause])
```

Description:

These three macros arrange for the evaluation of *test-form*, whose value is then compared against the key or keys in each *normal-clause* in turn. When the value matches a key, then the remaining forms of *normal-clause* are evaluated, and the value of the last form is returned; subsequent clauses are not evaluated. When the value doesn't match any of the keys of a

normal-clause then the next *normal-clause* is tested. If all these clauses are exhausted, and there is no *else-clause*, then the value nil is returned. Otherwise, the forms in the *else-clause* are evaluated, and the value of the last one is returned. If there are no forms, then nil is returned.

The syntax of a *normal-clause* takes on these two forms:

```
(key form*)
```

where *key* may be an atom which denotes a single key, or else a list of keys. There is a restriction that the symbol t may not be used as *key*. The form (t) may be used as a key to match that symbol.

The syntax of an *else-clause* is:

```
(t form*)
```

which resembles a form that is often used as the final clause in the cond syntax.

The three forms of the case construct differ from what type of test they apply between the value of *test-form* and the keys. The caseq macro generates code which uses the eq function's equality. The caseql macro uses eql, and casequal uses equal.

Example

```
(let ((command-symbol (casequal command-string
                        (("q" "quit") 'quit)
                        (("a" "add") 'add)
                        (("d" "del" "delete") 'delete)
                        (t 'unknown))))
  ...)
```

9.5.5 Macros caseq*, caseql* and casequal*

Syntax:

```
(caseq* test-form normal-clause* [else-clause])
(caseql* test-form normal-clause* [else-clause])
(casequal* test-form normal-clause* [else-clause])
```

Description:

The caseq*, caseql*, and casequal* macros are similar to the macros caseq, caseql, and casequal, differing from them in only the following regard. The *normal-clause*, of these macros has the form (*evaluated-key form**) where evaluated-key is either an atom, which is evaluated to produce a key, or else else a compound form, whose elements are evaluated as forms, producing multiple keys. This evaluation takes place at macro-expansion time, in the global environment.

The *else-clause* works the same way under these macros as under caseq et al.

Note that although in a *normal-clause*, *evaluated-key* must not be the atom t, there is no restriction against it being an atom which evaluates to t. In this situation, the value t has no special meaning.

The *evaluated-key* expressions are evaluated in the order in which they appear in the

construct, at the time the `caseq*`, `caseql*` or `casequal*` macro is expanded.

Note: these macros allow the use of variables and global symbol macros as case keys.

Example:

```
(defvarl red 0)
(defvarl green 1)
(defvarl blue 2)

(let ((color blue))
  (caseql* color
    (red "hot")
    ((green blue) "cool")))
--> "cool"
```

9.5.6 Macros `ecaseq`, `ecaseql`, `ecasequal`, `ecaseq*`, `ecaseql*` and `ecasequal*`

Syntax:

```
(ecaseq test-form normal-clause* [else-clause])
(ecaseql test-form normal-clause* [else-clause])
(ecasequal test-form normal-clause* [else-clause])
(ecaseq* test-form normal-clause* [else-clause])
(ecaseql* test-form normal-clause* [else-clause])
(ecasequal* test-form normal-clause* [else-clause])
```

Description:

These macros are error-catching variants of, respectively, `caseq`, `caseql`, `casequal`, `caseq*`, `caseql*` and `casequal*`.

If the *`else-clause`* is present in the invocation of an error-catching case macro, then the the invocation is precisely equivalent to the corresponding non-error-trapping variant.

If the *`else-clause`* is missing in the invocation of an error-catching variant, then a default *`else-clause`* is inserted which throws an exception of type `case-error`, derived from `error`. After this insertion, the semantics follows that of the non-error-trapping variant.

For instance, `(ecaseql 3)`, which has no *`else-clause`*, is equivalent to `(caseql 3 (t expr))` where *`expr`* indicates the inserted expression which throws `case-error`. However, `(ecaseql 3 (t 42))` is simply equivalent to `(caseql 3 (t 42))`, since it has an *`else-clause`*.

Note: the error-catching case macros are intended for situations in which it is a matter of program correctness that every possible value of *`test-form`* matches a *`normal-clause`*, such that if a failure to match occurs, it indicates a software defect. The error-throwing *`else-clause`* helps to ensure that the error situation is noticed. Without this clause, the case macro terminates with a value of `nil`, which may conceal the defect and delay its identification.

9.5.7 Operator/function `if`

Syntax:

```
(if cond t-form [e-form])
'['if cond then [else]']'
```

Description:

There exist both an `if` operator and an `if` function. A list form with the symbol `if` in the first position is interpreted as an invocation of the `if` operator. The function can be accessed using the DWIM bracket notation and in other ways.

The `if` operator provides a simple two-way-selective evaluation control. The *cond* form is evaluated. If it yields true then *t-form* is evaluated, and that form's return value becomes the return value of the `if`. If *cond* yields false, then *e-form* is evaluated and its return value is taken to be that of `if`. If *e-form* is omitted, then the behavior is as if *e-form* were specified as `nil`.

The `if` function provides no evaluation control. All of its arguments are evaluated from left to right. If the *cond* argument is true, then it returns the *then* argument, otherwise it returns the value of the *else* argument if present, otherwise it returns `nil`.

9.5.8 Operator/function `and`

Syntax:

```
(and form*)
'['and arg*']'
```

Description:

There exist both an `and` operator and an `and` function. A list form with the symbol `and` in the first position is interpreted as an invocation of the operator. The function can be accessed using the DWIM bracket notation and in other ways.

The `and` operator provides three functionalities in one. It computes the logical "and" function over several forms. It controls evaluation (a.k.a. "short-circuiting"). It also provides an idiom for the convenient substitution of a value in place of `nil` when some other values are all true.

The `and` operator evaluates as follows. First, a return value is established and initialized to the value `t`. The *form*s, if any, are evaluated from left to right. The return value is overwritten with the result of each *form*. Evaluation stops when all *form*s are exhausted, or when `nil` is stored in the return value. When evaluation stops, the operator yields the return value.

The `and` function provides no evaluation control: it receives all of its arguments fully evaluated. If it is given no arguments, it returns `t`. If it is given one or more arguments, and any of them are `nil`, it returns `nil`. Otherwise, it returns the value of the last argument.

Examples:

```
(and) -> t
(and (> 10 5) (stringp "foo")) -> t
(and 1 2 3) -> 3  ;; shorthand for (if (and 1 2) 3).
```

9.5.9 Macro/function `nand`

Syntax:

```
(nand form*)
'['nand arg*']'
```

Description:

There exist both a `nand` macro and a `nand` function. A list form with the symbol `nand` in the first position is interpreted as an invocation of the macro. The function can be accessed using the

DWIM bracket notation and in other ways.

The `nand` macro and function are the logical negation of the `and` operator and function. They are related according to the following equivalences:

```
(nand f0 f1 f2 ...) <--> (not (and f0 f1 f2 ...))
[nand f0 f1 f2 ...] <--> (not [and f0 f1 f2 ...])
```

9.5.10 Operator/function `or`

Syntax:

```
(or form*)
'['or arg*']'
```

Description:

There exist both an `or` operator and an `or` function. A list form with the symbol `or` in the first position is interpreted as an invocation of the operator. The function can be accessed using the DWIM bracket notation and in other ways.

The `or` operator provides three functionalities in one. It computes the logical "or" function over several forms. It controls evaluation (a.k.a. "short-circuiting"). The behavior of `or` also provides an idiom for the selection of the first non-`nil` value from a sequence of forms.

The `or` operator evaluates as follows. First, a return value is established and initialized to the value `nil`. The *form*s, if any, are evaluated from left to right. The return value is overwritten with the result of each *form*. Evaluation stops when all *form*s are exhausted, or when a true value is stored into the return value. When evaluation stops, the operator yields the return value.

The `or` function provides no evaluation control: it receives all of its arguments fully evaluated. If it is given no arguments, it returns `nil`. If all of its arguments are `nil`, it also returns `nil`. Otherwise, it returns the value of the first argument which isn't `nil`.

Examples:

```
(or) -> nil
(or 1 2) -> 1
(or nil 2) -> 2
(or (> 10 20) (stringp "foo")) -> t
```

9.5.11 Macro/function `nor`

Syntax:

```
(nor form*)
'['nor arg*']'
```

Description:

There exist both a `nor` macro and a `nor` function. A list form with the symbol `nor` in the first position is interpreted as an invocation of the macro. The function can be accessed using the DWIM bracket notation and in other ways.

The `nor` macro and function are the logical negation of the `or` operator and function. They are related according to the following equivalences:

```
(nor f0 f1 f2 ...) <--> (not (or f0 f1 f2 ...))
```

```
[nor f0 f1 f2 ...] <--> (not [or f0 f1 f2 ...])
```

9.5.12 Macros `when` and `unless`

Syntax:

```
(when expression form*)
(unless expression form*)
```

Description:

The `when` macro operator evaluates *`expression`*. If *`expression`* yields true, and there are additional forms, then each *`form`* is evaluated. The value of the last form becomes the result value of the `when` form. If there are no forms, then the result is `nil`.

The `unless` operator is similar to `when`, except that it reverses the logic of the test. The forms, if any, are evaluated if and only if *`expression`* is false.

9.5.13 Macros `while` and `until`

Syntax:

```
(while expression form*)
(until expression form*)
```

Description:

The `while` macro operator provides a looping construct. It evaluates *`expression`*. If *`expression`* yields `nil`, then the evaluation of the `while` form terminates, producing the value `nil`. Otherwise, if there are additional forms, then each *`form`* is evaluated. Next, evaluation returns to *`expression`*, repeating all of the previous steps.

The `until` macro operator is similar to `while`, except that the `until` form terminates when *`expression`* evaluates true, rather than false.

These operators arrange for the evaluation of all their enclosed forms in an anonymous block. Any of the *`form`*s, or *`expression`*, may use the `return` operator to terminate the loop, and optionally to specify a result value for the form.

The only way these forms can yield a value other than `nil` is if the `return` operator is used to terminate the implicit anonymous block, and is given an argument, which becomes the result value.

9.5.14 Macros `while*` and `until*`

Syntax:

```
(while* expression form*)
(until* expression form*)
```

Description:

The `while*` and `until*` macros are similar, respectively, to the macros `while` and `until`.

They differ in one respect: they begin by evaluating the *`form`*s one time unconditionally, without first evaluating *`expression`*. After this evaluation, the subsequent behavior is like that of `while` or `until`.

Another way to regard the behavior is that that these forms execute one iteration unconditionally, without evaluating the termination test prior to the first iteration. Yet another view is that these

constructs relocate the test from the top of the loop to the bottom of the loop.

9.5.15 Macro `whilet`

Syntax:

```
(whilet ({sym | (sym init-form)}+)
   body-form*)
```

Description:

The `whilet` macro provides a construct which combines iteration with variable binding.

The evaluation of the form takes place as follows. First, fresh bindings are established for *sym*s as if by the `let*` operator. It is an error for the list of variable bindings to be empty.

After the establishment of the bindings, the value of the last *sym* is tested. If the value is `nil`, then `whilet` terminates. Otherwise, *body-form*s are evaluated in the scope of the variable bindings, and then `whilet` iterates from the beginning, again establishing fresh bindings for the *sym*s, and testing the value of the last *sym*.

All evaluation takes place in an anonymous block, which can be terminated with the `return` operator. Doing so terminates the loop. If the `whilet` loop is thus terminated by an explicit `return`, a return value can be specified. Under normal termination, the return value is `nil`.

In the syntax, a small convenience is permitted. Instead of the last `(`*sym init-form*`)` it is permissible for the syntax `(`*init-form*`)` to appear, the *sym* being omitted. A machine-generated variable is substituted in place of the missing *sym* and that variable is then initialized from *init-form* and used as the basis of the test.

Examples:

```
;; read lines of text from *stdin* and print them,
;; until the end-of-stream condition:

(whilet ((line (get-line)))
  (put-line line))

;; read lines of text from *stdin* and print them,
;; until the end-of-stream condition occurs or
;; a line is identical to the character string "end".

(whilet ((line (get-line))
         (more (and line (nequal line "end"))))
  (put-line line))
```

9.5.16 Macros `iflet` and `whenlet`

Syntax:

```
(iflet {({sym | (sym init-form)}+) | atom-form}
   then-form [else-form])
(whenlet {({sym | (sym init-form)}+) | atom-form}
   body-form*)
```

Description:

The `iflet` and `whenlet` macros combine the variable binding of `let*` with conditional evaluation of `if` and `when`, respectively.

In either construct's syntax, a non-compound form *atom-form* may appear in place of the variable binding list. In this case, *atom-form* is evaluated as a form, and the construct is equivalent to its respective ordinary `if` or `when` counterpart.

If the list of variable bindings is empty, it is interpreted as the atom `nil` and treated as an *atom-form*.

If one or more bindings are specified rather than *atom-form*, then the evaluation of these forms takes place as follows. First, fresh bindings are established for *sym*s as if by the `let*` operator.

Then, the last variable's value is tested. If it is not `nil` then the test is true, otherwise false.

In the syntax, a small convenience is permitted. Instead of the last `(`*sym init-form*`)` it is permissible for the syntax `(`*init-form*`)` to appear, the *sym* being omitted. A machine-generated variable is substituted in place of the missing *sym* and that variable is then initialized from *init-form* and used as the basis of the test. This is intended to be useful in situations in which *then-form* or *else-form* do not require access to the tested value.

In the case of the `iflet` operator, if the test is true, the operator evaluates *then-form* and yields its value. Otherwise the test is false, and if the optional *else-form* is present, that is evaluated instead and its value is returned. If this form is missing, then `nil` is returned.

In the case of the `whenlet` operator, if the test is true, then the *body-form*s, if any, are evaluated. The value of the last one is returned, otherwise `nil` if the forms are missing. If the test is false, then evaluation of *body-form*s is skipped, and `nil` is returned.

Examples:

```
;; dispose of foo-resource if present
(whenlet ((foo-res (get-foo-resource obj)))
  (foo-shutdown foo-res)
  (set-foo-resource obj nil))

;; Contrast with: above, using when and let
(let ((foo-res (get-foo-resource obj)))
  (when foo-res
    (foo-shutdown foo-res)
    (set-foo-resource obj nil)))

;; print frobosity value if it exceeds 150
(whenlet ((fv (get-frobosity-value))
          (exceeds-p (> fv 150)))
  (format t "frobosity value ~a exceeds 150\n" fv))

;; same as above, taking advantage of the
;; last variable being optional:
(whenlet ((fv (get-frobosity-value))
          ((> fv 150)))
  (format t "frobosity value ~a exceeds 150\n" fv))
```

```
;; yield 4: 3 interpreted as atom-form
(whenlet 3 4)

;; yield 4: nil interpreted as atom-form
(iflet () 3 4)
```

9.5.17 Macro `condlet`

Syntax:

```
(condlet
  ([({ sym | (sym init-form)}+) | atom-form]
   body-form*)*)
```

Description:

The `condlet` macro generalizes `iflet`.

Each argument is a compound consisting of at least one item: a list of bindings or *`atom-form`*. This item is followed by zero or more *`body-form`*s.

If there are no *`body-form`*s then the situation is treated as if there were a single *`body-form`* specified as `nil`.

The arguments of `condlet` are considered in sequence, starting with the leftmost.

If the argument's left item is an *`atom-form`* then the form is evaluated. If it yields true, then the *`body-form`*s next to it are evaluated in order, and the `condlet` form terminates, yielding the value obtained from the last *`body-form`*. If *`atom-form`* yields false, then the next argument is considered, if there is one.

If the argument's left item is a list of bindings, then it is processed with exactly the same logic as under the `iflet` macro. If the last binding contains a true value, then the adjoining *`body-form`*s are evaluated in a scope in which all of the bindings are visible, and `condlet` terminates, yielding the value of the last *`body-form`*. Otherwise, the next argument of `condlet` is considered (processed in a scope in which the bindings produced by the current item are no longer visible).

If `condlet` runs out of arguments, it terminates and returns `nil`.

Example:

```
(let ((l '(1 2 3)))
  (condlet
    ;; first arg
    (((a (first l)   ;; a binding gets 1
      (b (second l)) ;; b binding gets 2
      (g (> a b))))  ;; last variable g is nil
     'foo)           ;; not evaluated
    ;; second arg
    (((b (second l)  ;; b gets 2
      (c (third l))  ;; c gets 3
      (g (> b c))))  ;; last variable g is true
     'bar)))         ;; condlet terminates
--> bar              ;; result is bar
```

9.5.18 Macro `ifa`

Syntax:

```
(ifa cond then [else])
```

Description:

The `ifa` macro provides an anaphoric conditional operator resembling the `if` operator. Around the evaluation of the *`then`* and *`else`* forms, the symbol `it` is implicitly bound to a subexpression of *`cond`*, a subexpression which is thereby identified as the *it-form*. This `it` alias provides a convenient reference to that place or value, similar to the word "it" in the English language, and similar anaphoric pronouns in other languages.

If `it` is bound to a place form, the binding is established as if using the `placelet` operator: the form is evaluated only once, even if the `it` alias is used multiple times in the *`then`* or *`else`* expressions. Furthermore, the place form is implicitly surrounded with `read-once` so that the place's value is accessed only once, and multiple references to `it` refer to a copy of the value cached in a hidden variable, rather than generating multiple accesses to the place. Otherwise, if the form is not a syntactic place `it` is bound as an ordinary lexical variable to the form's value.

An *it-candidate* is an an expression viable for having its value or storage location bound to the `it` symbol. An it-candidate is any expression which is not a constant expression according to the `constantp` function, and not a symbol.

The `ifa` macro imposes applies several rules to the *`cond`* expression:

1. The *`cond`* expression must be either an atom, a function call form, or a `dwim` form. Otherwise the `ifa` expression is ill-formed, and throws an exception at macro-expansion time. For the purposes of these rules, a `dwim` form is considered as a function call expression, whose first argument is the second element of the form. That is to say, `[f x]` which is equivalent to `(dwim f x)` is treated similarly to `(f x)` as a one-argument call.

2. If the *`cond`* expression is a function call with two or more arguments, at most one of them may be an it-candidate. If two or more arguments are it-candidates, the situation is ambiguous. The `ifa` expression is ill-formed and throws an exception at macro-expansion time.

3. If *`cond`* is an atom, or a function call expression with no arguments, then the `it` symbol is not bound. Effectively, `ifa` macro behaves like the ordinary `if` operator.

4. If *`cond`* is a function call or `dwim` expression with exactly one argument, then the `it` variable is bound to the argument expression, except when the function being called is `not`, `null`, or `false`. This binding occurs regardless of whether the expression is an it-candidate.

5. If *`cond`* is a function call with exactly one argument to the Boolean negation function which goes by one of the three names `not`, `null`, or `false`, then that situation is handled by a rewrite according to the following pattern:

   ```
   (ifa (not expr) then else) -> (ifa expr else then)
   ```

   which applies likewise for `null` or `false` substituted for `not`. The Boolean inverse function is removed, and the *`then`* and *`else`* expressions are exchanged.

6. If *`cond`* is a function call with two or more arguments, then it is only well-formed if at most one of those arguments is an it-candidate. If there is one such argument, then the `it` variable is bound to it.

7. Otherwise the variable is bound to the leftmost argument expression, regardless of whether that argument expression is an it-candidate.

In all other regards, the `ifa` macro behaves similarly to `if`.

The *cond* expression is evaluated, and, if applicable, the value of, or storage location denoted by the appropriate argument is captured and bound to the variable `it` whose scope extends over the *then* form, as well as over *else*, if present.

If *cond* yields a true value, then *then* is evaluated and the resulting value is returned, otherwise *else* is evaluated if present and its value is returned. A missing *else* is treated as if it were the `nil` form.

Examples:

```
(ifa t 1 0)  ->  1

;; Rule 6: it binds to (* x x), which is
;; the only it-candidate.
(let ((x 6) (y 49))
  (ifa (> y (* x x)) ;; it binds to (* x x)
    (list it)))
-> (36)

;; Rule 4: it binds to argument of evenp,
;; even though 4 isn't an it-candidate.
(ifa (evenp 4)
  (list it))
-> (4)

;; Rule 5:
(ifa (not (oddp 4))
  (list it))
-> (4)

;; Rule 7: no candidates: choose leftmost
(let ((x 6) (y 49))
  (ifa (< 1 x y)
    (list it)))
-> (1)

-> (4)
;; Violation of Rule 1:
;; while is not a function
(ifa (while t (print 42))
  (list it))
--> exception!

;; Violation of Rule 2:
(let ((x 6) (y 49))
  (ifa (> (* y y y) (* x x)))
    (list it))
--> exception!
```

9.5.19 Macro conda

Syntax:

```
(conda {(test form*)}*)
```

Description:

The conda operator provides a multi-branching conditional evaluation of forms, similarly to the cond operator. Enclosed in the cond form are groups of forms expressed as lists. Each group must be a list of at least one form.

The conda operator is anaphoric: it expands into a nested structure of zero or more ifa invocations, according to these patterns:

```
(conda) -> nil
(conda (x y ...) ...) -> (ifa x (progn y ...) (conda ...))
```

Thus, conda inherits all the restrictions on the *test* expressions from ifa, as well as the anaphoric it variable feature.

9.5.20 Macro whena

Syntax:

```
(whena test form*)
```

Description:

The whena macro is similar to the when macro, except that it is anaphoric in exactly the same manner as the ifa macro. It may be understood as conforming to the following equivalence:

```
(whena x f0 f2 ...)  <-->  (if x (progn f0 f2 ...))
```

9.5.21 Macro dotimes

Syntax:

```
(dotimes (var count-form [result-form])
   body-form*)
```

Description:

The dotimes macro implements a simple counting loop. *var* is established as a variable, and initialized to zero. *count-form* is evaluated one time to produce a limiting value, which should be a number. Then, if the value of *var* is less than the limiting value, the *body-form*s are evaluated, *var* is incremented by one, and the process repeats with a new comparison of *var* against the limiting value possibly leading to another evaluation of the forms.

If *var* is found to equal or exceed the limiting value, then the loop terminates.

When the loop terminates, its return value is nil unless a *result-form* is present, in which case the value of that form specifies the return value.

*body-form*s as well as *result-form* are evaluated in the scope in which the binding of *var* is visible.

9.5.22 Operators each, each*, collect-each, collect-each*, append-each and append-each*

Syntax:

```
(each ({(sym init-form)}*) body-form*)
(each* ({(sym init-form)}*) body-form*)
(collect-each ({(sym init-form)}*) body-form*)
(collect-each* ({(sym init-form)}*) body-form*)
(append-each ({(sym init-form)}*) body-form*)
(append-each* ({(sym init-form)}*) body-form*)
```

Description:

These operators establish a loop for iterating over the elements of one or more sequences. Each *init-form* must evaluate to an iterable object that is suitable as an argument for the `iter-begin` function. The sequences are then iterated in parallel over repeated evaluations of the *body-form*s, with each *sym* variable being assigned to successive elements of its sequence. The shortest list determines the number of iterations, so if any of the *init-form*s evaluate to an empty sequence, the body is not executed.

If the list of `(sym init-form)` pairs itself is empty, then an infinite loop is specified.

The body forms are enclosed in an anonymous block, allowing the `return` operator to terminate the loop prematurely and optionally specify the return value.

The `collect-each` and `collect-each*` variants are like `each` and `each*`, except that for each iteration, the resulting value of the body is collected into a list. When the iteration terminates, the return value of the `collect-each` or `collect-each*` operator is this collection.

The `append-each` and `append-each*` variants are like `each` and `each*`, except that for each iteration other than the last, the resulting value of the body must be a list. The last iteration may produce either an atom or a list. The objects produced by the iterations are combined together as if they were arguments to the append function, and the resulting value is the value of the `append-each` or `append-each*` operator.

The alternate forms denoted by the adorned symbols `each*`, `collect-each*` and `append-each*`, differ from `each`, `collect-each` and `append-each` in the following way. The plain forms evaluate the *init-form*s in an environment in which none of the *sym* variables are yet visible. By contrast, the alternate forms evaluate each *init-form* in an environment in which bindings for the previous *sym* variables are visible. In this phase of evaluation, *sym* variables are list-valued: one by one they are each bound to the list object emanating from their corresponding *init-form*. Just before the first loop iteration, however, the *sym* variables are assigned the first item from each of their lists.

Note:

The semantics of `collect-each` may be understood in terms of an equivalence to a code pattern involving `mapcar`:

```
(collect-each ((x xinit)         (mapcar (lambda (x y)
               (y yinit))  <-->            body)
  body)                                  xinit yinit)
```

The `collect-each*` variant may be understood in terms of the following equivalence involving `let*` for sequential binding and `mapcar`:

```
(collect-each* ((x xinit)         (let* ((x xinit)
                (y yinit))  <-->         (y yinit))
```

```
    body)                                   (mapcar (lambda (x y)
                                                       body)
                                                    x y))
```

However, note that the `let*` as well as each invocation of the `lambda` binds fresh instances of the variables, whereas these operators are permitted to bind a single instance of the variables, which are first initialized with the initializing expressions, and then reused as iteration variables which are stepped by assignment.

The other operators may be understood likewise, with the substitution of the `mapdo` function in the case of `each` and `each*` and of the `mappend` function in the case of `append-each` and `append-each*`.

Example:

```
;; print numbers from 1 to 10 and whether they are even or odd
(each* ((n 1..11) ;; n is just a range object in this scope
        (even (collect-each ((m n)) (evenp m))))
  ;; n is an integer in this scope
  (format t "~s is ~s\n" n (if even "even" "odd")))
```

Output:

```
1 is "odd"
2 is "even"
3 is "odd"
4 is "even"
5 is "odd"
6 is "even"
7 is "odd"
8 is "even"
9 is "odd"
10 is "even"
```

9.5.23 Macros `for` and `for*`

Syntax:

```
({for | for*} ({sym | (sym init-form)}*)
              ([test-form result-form*])
              [(inc-form*)]
   body-form*)

({for | for*} ({sym | (sym init-form)}*)
              ([test-form result-form*]))

({for | for*} ({sym | (sym init-form)}*))
```

Description:

The macros `for` and `for*` combine variable binding with loop iteration. The first argument is a list of variables with optional initializers, exactly the same as in the `let` and `let*` operators. Furthermore, the difference between `for` and `for*` is like that between `let` and `let*` with regard to this list of variables.

The second variant in the above syntax synopsis shows that when *body-form*s are absent, then a list of *inc-form*s which is empty may be omitted from the syntax.

The `for` and `for*` macros execute these steps:

1. Establish an anonymous block over the entire form, allowing the `return` operator to be used to terminate the loop.
2. Establish bindings for the specified variables similarly to `let` and `let*`. The variable bindings are visible over the *test-form*, each *result-form*, each *inc-form* and each *body-form*.
3. Evaluate *test-form*. If *test-form* yields `nil`, then the loop terminates. Each *result-form* is evaluated, and the value of the last of these forms is the result value of the loop. If there are no *result-form*s then the result value is `nil`. If the *test-form* is omitted, then the test is taken to be true, and the loop does not terminate.
4. Otherwise, if *test-form* yields true, then each *body-form* is evaluated in turn. Then, each `inc-form` is evaluated in turn and processing resumes at step 2.

9.5.24 Macros `doloop` and `doloop*`

Syntax:

```
({doloop | doloop*}
   ({ sym | (sym [init-form [step-form])}*)
   ([test-form result-form*])
   tagbody-form*)
```

Description:

The `doloop` and `doloop*` macros provide an iteration construct inspired by the ANSI Common Lisp `do` and `do*` macros.

Each *sym* element in the form must be a symbol suitable for use as a variable name.

The *tagbody-form*s are placed into an implicit `tagbody`, meaning that a *tagbody-form* which is an integer, character or symbol is interpreted as a `tagbody` label which may be the target of a control transfer via the `go` macro.

The `doloop` macro binds each *sym* to the value produced by evaluating the adjacent *init-form*. Then, in the environment in which these variables now exist, *test-form* is evaluated. If that form yields `nil`, then the loop terminates. The *result-form*s are evaluated, and the value of the last one is returned.

If *result-form*s are absent, then `nil` is returned.

If *test-form* is also absent, then the loop terminates and returns `nil`.

If *test-form* produces a true value, then *result-form*s are not evaluated. Instead, the implicit `tagbody` comprised of the *tagbody-form*s is evaluated. If that evaluation terminates normally, the loop variables are then updated by assigning to each *sym* the value of *step-form*.

The following defaulting behaviors apply in regard to the variable syntax. For each *sym* which has an associated *init-form* but no *step-form*, the *init-form* is duplicated and taken as the *step-form*. Thus a variable specification like `(x y)` is equivalent to `(x y y)`. If both forms are omitted, then the *init-form* is taken to be `nil`, and the *step-form* is taken to be *sym*. This means that the variable form `(x)` is equivalent to `(x nil x)` which has the effect that `x` retains its current value when the next loop iteration begins. Lastly, the *sym* variant is equivalent to `(sym)` so that `x` is also equivalent to `(x nil x)`.

The differences between `doloop` and `doloop*` are: `doloop` binds the variables in parallel,

similarly to `let`, whereas `doloop*` binds sequentially, like `let*`; moreover, `doloop` performs the *`step-form`* assignments in parallel as if using a single `(pset` *`sym0 step-form-0 sym1 step-form-1 ...`*`)` form, whereas `doloop*` performs the assignment sequentially as if using `set` rather than `pset`.

The `doloop` and `doloop*` macros establish an anonymous `block`, allowing early return from the loop, with a value, via the `return` operator.

Dialect Note:

These macros are substantially different from the ANSI Common Lisp `do` and `do*` macros. Firstly, the termination logic is inverted; effectively they implement "while" loops, whereas their ANSI CL counterparts implement "until" loops. Secondly, in the ANSI CL macros, the defaulting of the missing *`step-form`* is different. Variables with no *`step-form`* are not updated. In particular, this means that the form `(x y)` is not equivalent to `(x y y)`; the ANSI CL macros do not feature the automatic replication of *`init-form`* into the *`step-form`* position.

9.5.25 Macros `sum-each`, `sum-each*`, `mul-each` and `mul-each*`

Syntax:

```
(sum-each ({(sym init-form)}*) body-form*)
(sum-each* ({(sym init-form)}*) body-form*)
(mul-each ({(sym init-form)}*) body-form*)
(mul-each* ({(sym init-form)}*) body-form*)
```

Description:

The macros `sum-each`, and `mul-each` behave very similarly to the `each` operator. Whereas the `each` operator form returns `nil` as its result, the `sum-each` and `mul-each` forms, if they execute to completion and return normally, return an accumulated value.

The `sum-each` macro initializes newly instantiated, hidden accumulator variable to the value `0`. For each iteration of the loop, the *`body-form`*s are evaluated, and are expected to produce a value. This value is added to the current value of the hidden accumulator using the + function, and the result is stored into the accumulator. If `sum-each` returns normally, then the value of this accumulator becomes its resulting value.

The `mul-each` macro similarly initializes a hidden accumulator to the value `1`. The value from each iteration of the body is multiplied with the accumulator using the `*` function, and the result is stored into the accumulator. If `mul-each` returns normally, then the value of this accumulator becomes its resulting value.

The `sum-each*` and `mul-each*` variants of the macros implement the sequential scoping rule for the variable bindings, exactly the way `each*` alters the semantics of `each`.

The *`body-form`*s are enclosed in an implicit anonymous block. If the forms terminate by returning from the anonymous block then these macros terminate with the specified value.

When `sum-each*` and `sum-each` are specified with variables whose values specify zero iterations, or with no variables at all, the form terminates with a value of `0`. In this situation, `mul-each` and `mul-each*` terminate with `1`. Note that this behavior differs from `each`, and its closely-related operators, which loop infinitely when no variables are specified.

It is unspecified whether `mul-each` and `mul-each*` continue iterating when the accumulator takes on a value satisfying the `zerop` predicate.

9.5.26 Macros `each-true`, `some-true`, `each-false` and `some-false`

Syntax:

```
(each-true ({(sym init-form)}*) body-form*)
(some-true ({(sym init-form)}*) body-form*)
(each-false ({(sym init-form)}*) body-form*)
(some-false ({(sym init-form)}*) body-form*)
```

Description:

These macros iterate zero or more variables over sequences, similarly to the `each` operator, and calculate logical results, with short-circuiting semantics.

The `each-true` macro initializes an internal result variable to the `t` value. It then evaluates the *`body-form`*s for each tuple of variable values, replacing the result variable with the value produced by these forms. If that value is `nil`, the iteration stops. When the iteration terminates normally, the value of the result variable is returned.

If no variables are specified, termination occurs immediately. Note that this is different from the `each` operator, which iterates indefinitely if no variables are specified.

The *`body-form`*s are surrounded by an implicit anonymous block, making it possible to terminate via `return` or `return-from`. In these cases, the form terminates with `nil` or the specified return value. The internal result is ignored.

The `some-true` macro is similar to `each-true`, with the following differences. The internal result variable is initialized to `nil` rather than `t`. The iteration stops whenever the *`body-form`*s produce a true value, and that value is returned.

The `each-false` and `some-false` macros are, respectively, similar to `each-true` and `some-true`, with one difference. After each iteration, the value produced by the *`body-form`*s is logically inverted using the `not` function prior to being assigned to the result variable.

Examples:

```
(each-true ()) -> t
(each-true ((a ()))) -> t
(each-true ((a '(1 2 3))) a) -> 3

(each-true ((a '(1 2 3))
            (b '(4 5 6)))
  (< a b))
-> t

(each-true ((a '(1 2 3))
            (b '(4 0 6)))
  (< a b))
-> nil

(some-true ((a '(1 2 3))) a) -> 1
(some-true ((a '(nil 2 3))) a) -> 2
(some-true ((a '(nil nil nil))) a) -> nil

(some-true ((a '(1 2 3))
            (b '(4 0 6)))
  (< a b))
```

```
-> t

(some-true ((a '(1 2 3))
            (b '(0 1 2)))
  (< a b))
-> nil

(each-false ((a '(1 2 3))
             (b '(4 5 6)))
  (> a b))
-> t

(each-false ((a '(1 2 3))
             (b '(4 0 6)))
  (> a b))
-> nil

(some-false ((a '(1 2 3))
             (b '(4 0 6)))
  (> a b))
-> t

(some-false ((a '(1 2 3))
             (b '(0 1 2)))
  (> a b))
-> nil
```

9.5.27 Macros `each-prod`, `collect-each-prod` and `append-each-prod`

Syntax:

```
(each-prod ({(sym init-form)}*) body-form*)
(collect-each-prod ({(sym init-form)}*) body-form*)
(append-each-prod ({(sym init-form)}*) body-form*)
```

Description:

The macros `each-prod`, `collect-each-prod` and `append-each-prod` have a similar syntax to `each`, `collect-each` and `collect-each-prod`. However, instead of iterating over sequences in parallel, they iterate over the Cartesian product of the elements from the sequences. The difference between `collect-each` and `collect-each-prod` is analogous to that between the functions `mapcar` and `maprod`.

Like in the `each` operator family, the *`body-form`*s are surrounded by an anonymous block. If these forms execute a return from this block, then these macros terminate with the specified return value.

When no iterations are performed, including in the case when an empty list of variables is specified, all these macro forms terminate and return `nil`. Note that this behavior differs from `each`, and its closely-related operators, which loop infinitely when no variables are specified.

With one caveat noted below, these macros can be understood as providing syntactic sugar according to the pattern established by the following equivalences:

```
(each-prod                  (block nil
  ((x xinit)                  (let ((#:gx xinit) (#:gy yinit))
```

```
   (y yinit))       <-->     (maprodo (lambda (x y)
  body)                                body)
                                     #:gx #:gy))

(collect-each-prod       (block nil
  ((x xinit)               (let ((#:gx xinit) (#:gy yinit))
   (y yinit))       <-->     (maprod (lambda (x y)
  body)                                  body)
                                       #:gx #:gy))

(append-each-prod        (block nil
  ((x xinit)               (let ((#:gx xinit) (#:gy yinit))
   (y yinit))       <-->     (maprend (lambda (x y)
  body)                                  body)
                                       #:gx #:gy))
```

However, note that each invocation of the `lambda` binds fresh instances of the variables, whereas these operators are permitted to bind a single instance of the variables, which are then stepped by assignment.

Example:

```
(collect-each-prod ((a '(a b c))
                    (n #(1 2)))
  (cons a n))

--> ((a . 1) (a . 2) (b . 1) (b . 2) (c . 1) (c . 2))
```

9.5.28 Macros `each-prod*`, `collect-each-prod*` and `append-each-prod*`

Syntax:

```
(each-prod* ({(sym init-form)}*) body-form*)
(collect-each-prod* ({(sym init-form)}*) body-form*)
(append-each-prod* ({(sym init-form)}*) body-form*)
```

Description:

The macros `each-prod*`, `collect-each-prod*` and `append-each-prod*` are variants of `each-prod`, `collect-each-prod` and `append-each-prod` with sequential binding.

These macros can be understood as providing syntactic sugar according to the pattern established by the following equivalences:

```
(each-prod*              (let* ((x xinit)
  ((x xinit)                    (y yinit))
   (y yinit))       <-->   (maprodo (lambda (x y) body)
  body)                             x y)

(collect-each-prod*      (let* ((x xinit)
  ((x xinit)                    (y yinit))
   (y yinit))       <-->   (maprod (lambda (x y) body)
  body)                            x y)

(append-each-prod*       (let* ((x xinit)
  ((x xinit)                    (y yinit))
```

```
  (y yinit))        <-->   (maprend (lambda (x y) body)
 body)                                x y)
```

However, note that the `let*` as well as each invocation of the `lambda` binds fresh instances of the variables, whereas these operators are permitted to bind a single instance of the variables, which are first initialized with the initializing expressions, and then reused as iteration variables which are stepped by assignment.

Example:

```
(collect-each-prod* ((a "abc")
                     (b (upcase-str a)))
  `@a@b`)

--> ("aA" "aB" "aC" "bA" "bB" "bC" "cA" "cB" "cC")
```

9.5.29 Macros `sum-each-prod`, `sum-each-prod*`, `mul-each-prod` and `mul-each-prod*`

Syntax:

```
(sum-each-prod ({(sym init-form)}*) body-form*)
(sum-each-prod* ({(sym init-form)}*) body-form*)
(mul-each-prod ({(sym init-form)}*) body-form*)
(mul-each-prod* ({(sym init-form)}*) body-form*)
```

Description:

The macros `sum-each-prod` and `mul-each-prod` have a similar syntax to `sum-each` and `mul-each`. However, instead of iterating over sequences in parallel, they iterate over the Cartesian product of the elements from the sequences.

The macros `sum-each-prod*` and `mul-each-prod*` variants perform sequential variable binding when establishing the initial values of the variables, similarly to the `each*` operator.

The *`body-form`*s are surrounded by an implicit anonymous block. If these forms execute a return from this block, then these macros terminate with the specified return value.

When no iterations are specified, including in the case when an empty list of variables is specified, the summing macros terminate, yielding `0`, and the multiplicative macros terminate with `1`. Note that this behavior differs from `each`, and its closely-related operators, which loop infinitely when no variables are specified.

Examples:

```
;; Inefficiently calculate (+ (* 1 2 3) (* 4 3 2)).
;; Every value from (1 2 3) is paired with every value
;; from (4 3 2) to form a partial products, and
;; sum-each-prod adds these together implicitly:

(sum-each-prod ((x '(1 2 3))
                (y '(4 3 2)))
  (* x y))
-> 54
```

9.5.30 Operators `block` and `block*`

Syntax:

```
(block name body-form*)
(block* name-form body-form*)
```

Description:

The `block` operator introduces a named block around the execution of some forms. The *name* argument may be any object, though block names are usually symbols. Two block *name* objects are considered to be the same name according to `eq` equality. Since a block name is not a variable binding, keyword symbols are permitted, and so are the symbols `t` and `nil`. A block named by the symbol nil is slightly special: it is understood to be an anonymous block.

The `block*` operator differs from `block` in that it evaluates *name-form*, which is expected to produce a symbol. The resulting symbol is used for the name of the block.

A named or anonymous block establishes an exit point for the `return-from` or `return` operator, respectively. These operators can be invoked within a block to cause its immediate termination with a specified return value.

A block also establishes a prompt for a *delimited continuation*. Anywhere in a block, a continuation can be captured using the `sys:capture-cont` function. Delimited continuations are described in the section Delimited Continuations. A delimited continuation allows an apparently abandoned block to be restarted at the capture point, with the entire call chain and dynamic environment between the prompt and the capture point intact.

Blocks in **TXR Lisp** have dynamic scope. This means that the following situation is allowed:

```
(defun func () (return-from foo 42))
(block foo (func))
```

The function can return from the `foo` block even though the `foo` block does not lexically surround `foo`.

It is because blocks are dynamic that the `block*` variant exists; for lexically scoped blocks, it would make little sense to have support a dynamically computed name.

Thus blocks in **TXR Lisp** provide dynamic nonlocal returns, as well as returns out of lexical nesting.

It is permitted for blocks to be aggressively `progn`-converted by compilation. This means that a `block` form which meets certain criteria is converted to a `progn` form which surrounds the *body-form*s and thus no longer establishes an exit point.

A `block` form will be spared from `progn`-conversion by the compiler if it meets the following rules.

1. Any *body-form* references the block's *name* in a `return`, `return-from`, `sys:abscond-from` or `sys:capture-cont` expression.
2. The form contains at least one direct call to a function not in the standard **TXR Lisp** library.
3. The form contains at least one direct call to the functions `sys:capture-cont`, `return*`, `sys:abscond*`, `match-fun`, `eval`, `load`, `compile`, `compile-file` or `compile-toplevel`.

4. The form references any of the functions in rules 2 and 3 as a function binding via the `dwim` operator (or the DWIM brackets notation) or via the `fun` operator.
5. The form is a `block*` form; these are spared from the optimization.

Removal of blocks under the above rules means that some use of blocks which works in interpreted code will not work in compiled programs. Programs which adhere to the rules are not affected by such a difference.

Additionally, the compiler may `progn`-convert blocks in contravention of the above rules, but only if doing so makes no difference to visible program behavior.

Examples:

```
(defun helper ()
  (return-from top 42))

;; defun implicitly defines a block named top
(defun top ()
  (helper) ;; function returns 42
  (prinl 'notreached)) ;; never printed

(defun top2 ()
  (let ((h (fun helper)))
    (block top (call h))   ;; may progn-convert
    (block top (call 'helper)) ;; may progn-convert
    (block top (helper)))) ;; not removed
```

In the above examples, the block containing `(call h)` may be converted to `progn` because it doesn't express a **direct** call to the `helper` function. The block which calls `helper` using `(call 'helper)` is also not considered to be making a direct call.

Dialect Note:

In Common Lisp, blocks are lexical. A separate mechanism consisting of catch and throw operators performs nonlocal transfer based on symbols. The **TXR Lisp** example:

```
(defun func () (return-from foo 42))
(block foo (func))
```

is not allowed in Common Lisp, but can be transliterated to:

```
(defun func () (throw 'foo 42))
(catch 'foo (func))
```

Note that foo is quoted in CL. This underscores the dynamic nature of the construct. `throw` itself is a function and not an operator. Also note that the CL example, in turn, is even more closely transcribed back into **TXR Lisp** simply by replacing its `throw` and `catch` with `return*` and `block*`:

```
(defun func () (return* 'foo 42))
(block* 'foo (func))
```

Common Lisp blocks also do not support delimited continuations.

9.5.31 Operators `return` and `return-from`

Syntax:

```
(return [value])
(return-from name [value])
```

Description:

The `return` operator must be dynamically enclosed within an anonymous block (a block named by the symbol `nil`). It immediately terminates the evaluation of the innermost anonymous block which encloses it, causing it to return the specified value. If the value is omitted, the anonymous block returns `nil`.

The `return-from` operator must be dynamically enclosed within a named block whose name matches the *name* argument. It immediately terminates the evaluation of the innermost such block, causing it to return the specified value. If the value is omitted, that block returns `nil`.

Example:

```
(block foo
  (let ((a "abc\n")
        (b "def\n"))
    (pprint a *stdout*)
    (return-from foo 42)
    (pprint b *stdout*)))
```

Here, the output produced is `"abc"`. The value of `b` is not printed because. `return-from` terminates block `foo`, and so the second pprint form is not evaluated.

9.5.32 Function `return*`

Syntax:

```
(return* name [value])
```

Description:

The `return*` function is similar to the `return-from` operator, except that `name` is an ordinary function parameter, and so when `return*` is used, an argument expression must be specified which evaluates to a symbol. Thus `return*` allows the target block of a return to be dynamically computed.

The following equivalence holds between the operator and function:

```
(return-from a b)  <-->  (return* 'a b)
```

Expressions used as *name* arguments to `return*` which do not simply quote a symbol have no equivalent in `return-from`.

9.5.33 Macros `tagbody` and `go`

Syntax:

```
(tagbody {form | label}*)
(go label)
```

Description:

The `tagbody` macro provides a form of the "go to" control construct. The arguments of a

tagbody form are a mixture of zero or more forms and *go labels*. The latter consist of those arguments which are symbols, integers or characters. Labels are not considered by tagbody and go to be forms, and are not subject to macro expansion or evaluation.

The go macro is available inside tagbody. It is erroneous for a go form to occur outside of a tagbody. This situation is diagnosed by global macro called go, which unconditionally throws an error.

In the absence of invocations of go or other control transfers, the tagbody macro evaluates each *form* in left-to-right order. The go labels are ignored. After the last *form* is evaluated, the tagbody form terminates, and yields nil.

Any *form* itself, or else any of its subforms, may be the form (go *label*) where *label* matches one of the go labels of a surrounding tagbody. When this go form is evaluated, then the evaluation of *form* is immediately abandoned, and control transfers to the specified label. The forms are then evaluated in left-to-right order starting with the form immediately after that label. If the label is not followed by any forms, then the tagbody terminates. If *label* doesn't match to any label in any surrounding tagbody, the go form is erroneous.

The abandonment of a *form* by invocation of go is a dynamic transfer. All necessary unwinding inside *form* takes place.

The go labels are lexically scoped, but dynamically bound. Their scope being lexical means that the labels are not visible to forms which are not enclosed within the tagbody, even if their evaluation is invoked from that tagbody. The dynamic binding means that the labels of a tagbody form are established when it begins evaluating, and removed when that form terminates. Once a label is removed, it is not available to be the target of a go control transfer, even if that go form has the label in its lexical scope. Such an attempted transfer is erroneous.

It is permitted for tagbody forms to nest arbitrarily. The labels of an inner tagbody are not visible to an outer tagbody. However, the reverse is true: a go form in an inner tagbody may branch to a label in an outer tagbody, in which case the entire inner tagbody terminates.

In cases where the same objects are used as labels by an inner and outer tagbody, the inner labels shadow the outer labels.

There is no restriction on what kinds of symbols may be labels. Symbols in the keyword package as well as the symbols t and nil are valid tagbody labels.

Dialect Note:

ANSI Common Lisp tagbody supports only symbols and integers as labels (which are called "go tags"); characters are not supported.

Examples:

```
;; print the numbers 1 to 10
(let ((i 0))
  (tagbody
    (go skip) ;; forward goto skips 0
   again
    (prinl i)
   skip
    (when (<= (inc i) 10)
      (go again))))
```

```
;; Example of erroneous usage: by the time func is invoked
;; by (call func) the tagbody has already terminated. The
;; lambda body can still "see" the label, but it doesn't
;; have a binding.
(let (func)
  (tagbody
    (set func (lambda () (go label)))
    (go out)
   label
    (prinl 'never-reached)
    out)
  (call func))

;; Example of unwinding when the unwind-protect
;; form is abandoned by (go out). Output is:
;;   reached
;;   cleanup
;;   out
(tagbody
   (unwind-protect
     (progn
       (prinl 'reached)
       (go out)
       (prinl 'notreached))
     (prinl 'cleanup))
 out
   (prinl 'out))
```

9.5.34 Macros `prog` and `prog*`

Syntax:

```
(prog ({sym | (sym init-form)}*)
   {body-form | label}*)
(prog* ({sym | (sym init-form)}*)
   {body-form | label}*)
```

Description:

The `prog` and `progn*` macros combine the features of `let` and `let*`, respectively, anonymous block and `tagbody`.

The `prog` macro treats the *sym* and `init-form` expressions similarly to `let`, establishing variable bindings in parallel. The `prog*` macro treats these expressions in a similar way to `let*`.

The forms enclosed are treated like the argument forms of the `tagbody` macro: labels are permitted, along with use of `go`.

Finally, an anonymous block is established around all of the enclosed forms (both the *init-form*s and *body-forms*s) allowing the use of `return` to terminate evaluation with a value.

The `prog` macro may be understood according to the following equivalence:

```
(prog vars forms ...)  <-->  (block nil
                               (let vars
                                 (tagbody forms ...)))
```

Likewise, the `prog*` macro follows an analogous equivalence, with `let` replaced by `let*`.

9.6 Evaluation

9.6.1 Function `eval`

Syntax:

```
(eval form [env])
```

Description:

The `eval` function treats the *form* object as a Lisp expression, which is expanded and evaluated. The side effects implied by the form are performed, and the value which it produces is returned. The optional *env* object specifies an environment for resolving the function and variable references encountered in the expression. If this argument is omitted, then evaluation takes place in the global environment.

The *form* is not expanded all at once. Rather, it is treated by the following algorithm:

1. First, if *form* is a macro, it is macro-expanded as if by an application of the function `macroexpand`.
2. If the resulting expanded form is a `progn`, `compile-only`, or `eval-only` form, then `eval` iterates over that form's argument expressions, passing each expression to a recursive call to `eval` using the same *env*.
3. Otherwise, if the expanded form isn't one of the above three kinds of expressions, it is subject to a full expansion and evaluation.

This algorithm allows a sequence of top-level forms to be combined into a single top-level form, even when the expansion of forms occurring later in the sequence depends on the evaluation effects of forms earlier in the sequence.

For instance, a form like `(progn (defmacro foo ()) (foo))` may be processed with `eval`, because the above algorithm ensures that the `(defmacro foo ())` expression is fully evaluated first, thereby providing the macro definition required by `(foo)`.

This expansion and evaluation order is important because the semantics of `eval` forms the reference model for how the `load` function processes top-level forms. Moreover, file compilation perform a similar treatment of top-level forms and incremental macro compilation. The result is that the behavior is consistent between source files and compiled files. See the sections Top-Level Forms and File Compilation Model.

Note that, according to these rules, the constituent body forms of a `macrolet` or `symacrolet` top-level form are not individual top-level forms, even if the expansion of the construct combines the expanded versions of those forms with `progn`.

The form `(macrolet () (defmacro foo ()) (foo))` will therefore not work correctly. However, the specific problem in this situation can be be resolved by rewriting `foo` as a `macrolet` macro: `(macrolet ((foo ())) (foo))`.

See also: the `make-env` function.

9.6.2 Function `constantp`

Syntax:

```
(constantp form [env])
```

Description:

The `constantp` function determines whether *form* is a constant form, with respect to environment *env*.

If *env* is absent, the global environment is used. The *env* argument is used for fully expanding *form* prior to analyzing.

Currently, `constantp` returns true for any form which, after macro-expansion, is any of the following: a compound form with the symbol `quote` in its first position; a non-symbolic atom; or one of the symbols which evaluate to themselves and cannot be bound as variables. These symbols are the keyword symbols, and the symbols `t` and `nil`.

Additionally, `constantp` returns true for a compound form, or a DWIM form, whose symbol is the member of a set a large number of constant-foldable library functions, and whose arguments are, recursively, `constantp` expressions for the same environment. The arithmetic functions are members of this set.

For all other inputs, `constantp` returns `nil`.

Note: some uses of `constantp` require manual expansion.

Examples:

```
(constantp nil) -> t
(constantp t) -> t
(constantp :key) -> t
(constantp :) -> t
(constantp 'a) -> nil
(constantp 42) -> t

(constantp '(+ 2 2 [* 3 (/ 4 4)])) -> t

;; symacrolet form expands to 42, which is constant
(constantp '(symacrolet ((a 42)) a))

(defmacro cp (:env e arg)
  (constantp arg e))

;; macro call (cp 'a) is replaced by t because
;; the symbol a expands to (+ 2 2) in the given environment,
;; and so (* a a) expands to (* (+ 2 2) (+ 2 2)) which is constantp.
(symacrolet ((a (+ 2 2)))
  (cp '(* a a))) -> t
```

9.6.3 Function `make-env`

Syntax:

```
(make-env [var-bindings [fun-bindings [next-env]]])
```

Description:

The `make-env` function creates an environment object suitable as the `env` parameter.

The *var-bindings* and *fun-bindings* parameters, if specified, should be association lists, mapping symbols to objects. The objects in *fun-bindings* should be functions, or objects

callable as functions.

The *next-env* argument, if specified, should be an environment.

Note: bindings can also be added to an environment using the `env-vbind` and `env-fbind` functions.

9.6.4 Functions `env-vbind` and `env-fbind`

Syntax:

```
(env-vbind env symbol value)
(env-fbind env symbol value)
```

Description:

These functions bind a symbol to a value in either the function or variable space of environment `env`.

Values established in the function space should be functions or objects that can be used as functions such as lists, strings, arrays or hashes.

If *symbol* already exists in the environment, in the given space, then its value is updated with `value`.

If *env* is specified as `nil`, then the binding takes place in the global environment.

9.6.5 Functions `env-vbindings`, `env-fbindings` and `env-next`

Syntax:

```
(env-vbindings env)
(env-fbindings env)
(env-next env)
```

Description:

These function retrieve the components of *env*, which must be an environment. The `env-vbindings` function retrieves the association list representing variable bindings. Similarly, the `env-fbindings` retrieves the association list of function bindings. The `env-next` function retrieves the next environment, if *env* has one, otherwise `nil`.

If `e` is an environment constructed by the expression `(make-env v f n)`, then `(env-vbindings e)` retrieves `v`, `(env-fbindings e)` retrieves `f` and `(env-next e)` returns `n`.

9.7 Global Environment

9.7.1 Accessors `symbol-function`, `symbol-macro` and `symbol-value`

Syntax:

```
(symbol-function {symbol | method-name | lambda-expr})
(symbol-macro symbol)
(symbol-value symbol)
(set (symbol-function {symbol | method-name}) new-value)
(set (symbol-macro symbol) new-value)
(set (symbol-value symbol) new-value)
```

Description:

If given a *symbol* argument, the symbol-function function retrieves the value of the global function binding of the given *symbol* if it has one: that is, the function object bound to the *symbol*. If *symbol* has no global function binding, then nil is returned.

The symbol-function function supports method names of the form (meth *struct slot*) where *struct* names a struct type, and *slot* is either a static slot or one of the keyword symbols :init or :postinit which refer to special functions associated with a structure type. Names in this format are returned by the *func-get-name* function. The symbol-function function also supports names of the form (macro *name*) which denote macros. Thus, symbol-function provides unified access to functions, methods and macros.

If a lambda expression is passed to symbol-function, then the expression is macro-expanded and if that is successful, the function implied by that expression is returned. It is unspecified whether this function is interpreted or compiled.

The symbol-macro function retrieves the value of the global macro binding of *symbol* if it has one.

Note: the name of this function has nothing to do with symbol macros; it is named for consistency with symbol-function and symbol-value, referring to the "macro-expander binding of the symbol cell".

The value of a macro binding is a function object. Intrinsic macros are C functions in the **TXR** kernel, which receive the entire macro call form and macro environment, performing their own destructuring. Currently, macros written in **TXR Lisp** are represented as curried C functions which carry the following list object in their environment cell:

```
(#<environment object> macro-parameter-list body-form*)
```

Local macros created by macrolet have nil in place of the environment object.

This representation is likely to change or expand to include other forms in future **TXR** versions.

The symbol-value function retrieves the value stored in the dynamic binding of *symbol* that is apparent in the current context. If the variable has no dynamic binding, then symbol-value retrieves its value in the global environment. If *symbol* has no variable binding, but is defined as a global symbol macro, then the value of that symbol macro binding is retrieved. The value of a symbol macro binding is simply the replacement form.

Rather than throwing an exception, each of these functions returns nil if the argument symbol doesn't have the binding in the respective namespace or namespaces which that function searches.

A symbol-function, symbol-macro, or symbol-value form denotes a place, if *symbol* has a binding of the respective kind. This place may be assigned to or deleted. Assignment to the place causes the denoted binding to have a new value. Deleting a place with the del macro removes the binding, and returns the previous contents of that binding. A binding denoted by a symbol-function form is removed using fmakunbound, one denoted by by symbol-macro is removed using mmakunbound and a binding denoted by symbol-value is removed using makunbound.

Deleting a method via symbol-function is not possible; an attempt to do so has no effect.

Storing a value, using any one of these three accessors, to a nonexistent variable, function or

macro binding, is not erroneous. It has has the effect of creating that binding.

Using `symbol-function` accessor to assign to a lambda expression is erroneous.

Deleting a binding, using any of these three accessors, when the binding does not exist, also isn't erroneous. There is no effect and the `del` operator yields `nil` as the prior value, consistent with the behavior when accessors are used to retrieve a nonexistent value.

Dialect Note:

In ANSI Common Lisp, the `symbol-function` function retrieves a function, macro or special operator binding of a symbol. These are all in one space and may not coexist. In **TXR Lisp**, it retrieves a symbol's function binding only. Common Lisp has an accessor named `macro-function` similar to `symbol-macro`.

9.7.2 Functions `boundp`, `fboundp` and `mboundp`

Syntax:

```
(boundp symbol)
(fboundp {symbol | method-name | lambda-expr})
(mboundp symbol)
```

Description:

`boundp` returns `t` if the *`symbol`* is bound as a variable or symbol macro in the global environment, otherwise `nil`.

`fboundp` returns `t` if the *`symbol`* has a function binding in the global environment, the method specified by *`method-name`* exists, or a lambda expression argument is given. Otherwise it returns `nil`.

`mboundp` returns `t` if the symbol has an operator macro binding in the global environment, otherwise `nil`.

Dialect Notes:

The `boundp` function in ANSI Common Lisp doesn't report that global symbol macros have a binding. They are not considered bindings. In **TXR Lisp**, they are considered bindings.

The ANSI Common Lisp `fboundp` yields true if its argument has a function, macro or operator binding, whereas the **TXR Lisp** `fboundp` does not consider operators or macros. The ANSI CL `fboundp` does not yield true for lambda expressions. Behavior similar to the Common Lisp expression `(fboundp x)` in Common Lisp can be obtained in **TXR Lisp** using the

```
(or (fboundp x) (mboundp x) (special-operator-p x))
```

expression, except that this will also yield true when `x` is a lambda expression.

The `mboundp` function doesn't exist in ANSI Common Lisp.

9.7.3 Function `makunbound`

Syntax:

```
(makunbound symbol)
```

Description:

The function `makunbound` the binding of *`symbol`* from either the dynamic environment or the global symbol macro environment. After the call to `makunbound`, *`symbol`* appears to be unbound.

If the `makunbound` call takes place in a scope in which there exists a dynamic rebinding of *`symbol`*, the information for restoring the previous binding is not affected by `makunbound`. When that scope terminates, the previous binding will be restored.

If the `makunbound` call takes place in a scope in which the dynamic binding for `symbol` is the global binding, then the global binding is removed. When the global binding is removed, then if *`symbol`* was previously marked as special (for instance by `defvar`) this marking is removed.

Otherwise if *`symbol`* has a global symbol macro binding, that binding is removed.

If *`symbol`* has no apparent dynamic binding, and no global symbol macro binding, `makunbound` does nothing.

In all cases, `makunbound` returns *`symbol`*.

Dialect Note:

The behavior of `makunbound` differs from its counterpart in ANSI Common Lisp.

The `makunbound` function in Common Lisp only removes a value from a dynamic variable. The dynamic variable does not cease to exist, it only ceases to have a value (because a binding is a value). In **TXR Lisp**, the variable ceases to exist. The binding of a variable isn't its value, it is the variable itself: the association between a name and an abstract storage location, in some environment. If the binding is undone, the variable disappears.

The `makunbound` function in Common Lisp does not remove global symbol macros, which are not considered to be bindings in the variable namespace. That is to say, the Common Lisp `boundp` does not report true for symbol macros.

The Common Lisp `makunbound` also doesn't remove the special attribute from a symbol. If a variable is introduced with `defvar` and then removed with `makunbound`, the symbol continues to exhibit dynamic binding rather than lexical in subsequent scopes. In **TXR Lisp**, if a global binding is removed, so is the special attribute.

9.7.4 Functions `fmakunbound` and `mmakunbound`

Syntax:

```
(fmakunbound symbol)
(mmakunbound symbol)
```

Description:

The function `fmakunbound` removes any binding for *`symbol`* from the function namespace of the global environment. If *`symbol`* has no such binding, it does nothing. In either case, it returns *`symbol`*.

The function `mmakunbound` removes any binding for *`symbol`* from the operator macro namespace of the global environment. If *`symbol`* has no such binding, it does nothing. In either case, it returns *`symbol`*.

Dialect Note:

The behavior of `fmakunbound` differs from its counterpart in ANSI Common Lisp. The `fmakunbound` function in Common Lisp removes a function or macro binding, which do not coexist.

The `mmakunbound` function doesn't exist in Common Lisp.

9.7.5 Function `func-get-form`

Syntax:

```
(func-get-form func)
```

Description:

The `func-get-form` function retrieves a source code form of *`func`*, which must be an interpreted function. The source code form has the syntax `(name arglist body-form*)` .

9.7.6 Function `func-get-name`

Syntax:

```
(func-get-name func [env])
```

Description:

The `func-get-name` tries to resolve the function object *`func`* to a name. If that is not possible, it returns `nil`.

The resolution is performed by an exhaustive search through up to three spaces.

If an environment is specified by *`env`*, then this is searched first. If a binding is found in that environment which resolves to the function, then the search terminates and the binding's symbol is returned as the function's name.

If the search through environment *`env`* fails, or if that argument is not specified, then the global environment is searched for a function binding which resolves to *`func`*. If such a binding is found, then the search terminates, and the binding's symbol is returned. If two or more symbols in the global environment resolve to the function, it is not specified which one is returned.

If the global function environment search fails, then the function is considered as a possible macro. The global macro environment is searched for a macro binding whose expander function is *`func`*, similarly to the way the function environment was searched. If a binding is found, then the syntax `(macro name)` is returned, where *`name`* is the name of the global macro binding that was found which resolves to *`func`*. If two or more global macro bindings share *`func`*, it is not specified which of those bindings provides *`name`*.

If the global macro search fails, then *`func`* is considered as a possible method. The static slot space of all struct types is searched for a slot which contains *`func`*. If such a slot is found, then the method name is returned, consisting of the syntax `(meth type name)` where *`type`* is a symbol denoting the struct type and *`name`* is the static slot of the struct type which holds *`func`*.

A check is also performed whether *`func`* might be equal to one of the two special functions of a structure type: its *`initfun`* or *`postinitfun`*, in which case it is returned as either the `(meth type :init)` or the `(meth type :postinit)` syntax.

If *`func`* is an interpreted function not found under any name, then a lambda expression denoting

that function is returned in the syntax `(lambda *args form**)`

If *func* cannot be identified as a function, then `nil` is returned.

9.7.7 Function `func-get-env`

Syntax:

```
(func-get-env func)
```

Description:

The `func-get-env` function retrieves the environment object associated with function *func*. The environment object holds the captured bindings of a lexical closure.

9.7.8 Functions `fun-fixparam-count` and `fun-optparam-count`

Syntax:

```
(fun-fixparam-count func)
(fun-optparam-count func)
```

Description:

The `fun-fixparam-count` reports *func*'s number of fixed parameters. The fixed parameters consist of the required parameters and the optional parameters. Variadic functions have a parameter which captures the remaining arguments which are in excess of the fixed parameters. That parameter is not considered a fixed parameter and therefore doesn't contribute to this count.

The `fun-optparam-count` reports *func*'s number of optional parameters.

The *func* argument must be a function.

Note: if a function isn't variadic (see the *fun-variadic* function) then the value reported by `fun-fixparam-count` represents the maximum number of arguments which can be passed to the function. The minimum number of required arguments can be calculated for any function by subtracting the value from `fun-optparam-count` from the value from `fun-fixparam-count`.

9.7.9 Function `fun-variadic`

Syntax:

```
(fun-variadic func)
```

Description:

The `fun-variadic` function returns `t` if *func* is a variadic function, otherwise `nil`.

The *func* argument must be a function.

9.7.10 Function `interp-fun-p`

Syntax:

```
(interp-fun-p obj)
```

Description:

The `interp-fun-p` function returns `t` if *obj* is an interpreted function, otherwise it returns `nil`.

9.7.11 Function `vm-fun-p`

Syntax:

```
(vm-fun-p obj)
```

Description:

The `vm-fun-p` function returns `t` if *obj* a function compiled for the virtual machine: a function representation produced by means of the functions `compile-file`, `compile-toplevel` or `compile`. If *obj* is of any other type, the function returns `nil`.

9.7.12 Function `special-var-p`

Syntax:

```
(special-var-p obj)
```

Description:

The `special-var-p` function returns `t` if *obj* is a symbol marked for special variable binding, otherwise it returns `nil`. Symbols are marked special by `defvar` and `defparm`.

9.7.13 Function `special-operator-p`

Syntax:

```
(special-operator-p obj)
```

Description:

The `special-operator-p` function returns `t` if *obj* is a symbol which names a special operator, otherwise it returns `nil`.

9.7.14 Symbol macro `%fun%`

Description:

The symbol macro `%fun%` indicates the current function name, There is a global `%fun%` symbol macro which expands to `nil`. Around certain kinds of named functions, a local binding for `%fun%` is established which provides the function name. The purpose of this name is for use in diagnostic messages; therefore it is an abbreviated name.

The `%fun%` macro is established for `defun`, `defmacro` and `defmeth` forms. It is also established for methods defined inside a `defstruct` form including the methods `:init`, `:postinit`, `:fini` and `:postfini`.

The `%fun%` macro is visible not only to the its function's body, but also to the expressions inside the parameter list which compute the default values for optional parameters.

The name provided by `%fun%` is intended for use in diagnostic messages and is therefore an informal name, and not the formal name which can be passed to `symbol-function` to retrieve the function.

In the case of a `defun` function named `x`, the `%fun%` name is that symbol, `x`. Thus, in this case, the name is the same as the formal name. In the case of a `defmacro` named `x`, `%fun%` also expands to the symbol `x` `x`, but that is the formal name of the macro, which is `(macro x)`. In the case of a method `x` of a structure type `s`, `%fun%` is the two-element list `(s x)`, rather than the formal name `(meth s x)`.

Example:

```
;; log a message naming the function
(defun connect-to-host (addr)
  (format t "~s: connecting to host ~s" %fun% addr))
```

9.8 Object Type

In **TXR Lisp**, objects obey the following type hierarchy. In this type hierarchy, the internal nodes denote abstract types: no object is an instance of an abstract type. Nodes in square brackets indicate an internal structure in the type graph, invisible to programs, and angle brackets indicate a plurality of types which are not listed by name:

```
t ----+--- [cobj types] ---+--- hash
      |                    |
      |                    +--- hash-iter
      |                    |
      |                    +--- stream
      |                    |
      |                    +--- random-state
      |                    |
      |                    +--- regex
      |                    |
      |                    +--- buf
      |                    |
      |                    +--- tree
      |                    |
      |                    +--- tree-iter
      |                    |
      |                    +--- seq-iter
      |                    |
      |                    +--- cptr
      |                    |
      |                    +--- dir
      |                    |
      |                    +--- struct-type
      |                    |
      |                    +--- <all structures>
      |                    |
      |                    +--- ... others
      |
      |
      +--- sequence ---+--- string ---+--- str
      |                |              |
      |                |              +--- lstr
      |                |              |
      |                |              +--- lit
      |                |
      |                +--- list ---+--- null
      |                |            |
      |                |            +--- cons
      |                |            |
      |                |            +--- lcons
      |                |
      |                +--- vec
```

```
|                |
|                +--- <structures with car or length methods>
|
+--- number ---+--- float
|              |
|              +--- integer ---+--- fixnum
|                              |
|                              +--- bignum
|
+--- chr
|
+--- sym
|
+--- env
|
+--- range
|
+--- tnode
|
+--- pkg
|
+--- fun
|
+--- args
```

In addition to the above hierarchy, the following relationships also exist:

```
t ---+--- atom --- <any type other than cons> --- nil
     |
     +--- cons ---+--- lcons --- nil
                  |
                  +--- nil

sym --- null

struct ---- <all structures>
```

That is to say, the types are exhaustively partitioned into atoms and conses; an object is either a `cons` or else it isn't, in which case it is the abstract type `atom`.

The `cons` type is odd in that it is both an abstract type, serving as a supertype for the type `lcons` and it is also a concrete type in that regular conses are of this type.

The type `nil` is an abstract type which is empty. That is to say, no object is of type `nil`. This type is considered the abstract subtype of every other type, including itself.

The type `nil` is not to be confused with the type `null` which is the type of the `nil` symbol.

Because the type of `nil` is the type `null` and `nil` is also a symbol, the `null` type is a subtype of `sym`.

Lastly, the symbol `struct` serves as the supertype of all structures.

9.8.1 Function `typeof`

Syntax:

```
(typeof value)
```

Description:

The `typeof` function returns a symbol representing the type of *`value`*.

The core types are identified by the following symbols:

`cons` Cons cell.

`str` String.

`lit` Literal string embedded in the **TXR** executable image.

`chr` Character.

`fixnum`
: Fixnum integer: an integer that fits into the value word, not having to be heap-allocated.

`bignum`
: A bignum integer: arbitrary precision integer that is heap-allocated.

`float` Floating-point number.

`sym` Symbol.

`pkg` Symbol package.

`fun` Function.

`vec` Vector.

`lcons` Lazy cons.

`range` Range object.

`lstr` Lazy string.

`env` Function/variable binding environment.

`hash` Hash table.

`stream`
: I/O stream of any kind.

`regex` Regular-expression object.

`struct-type`
A structure type: the type of any one of the values which represents a structure type.

`tnode` Binary search tree node.

`tree` Binary search tree.

`args` Function argument list represented as an object.

There are more kinds of objects, such as user-defined structures.

9.8.2 Function `subtypep`

Syntax:

```
(subtypep left-type right-type)
```

Description:

The `subtypep` function tests whether *`left-type`* and *`right-type`* name a pair of types, such that the left type is a subtype of the right type.

The arguments are either type symbols, or structure type objects, as returned by the `find-struct-type` function. Thus, the symbol `time`, which is the name of a predefined struct type, and the object returned by `(find-struct-type 'time)` are considered equivalent argument values.

If either argument doesn't name a type, the behavior is unspecified.

Each type is a subtype of itself. Most other type relationships can be inferred from the type hierarchy diagrams given in the introduction to this section.

In addition, there are inheritance relationships among structures. If *`left-type`* and *`right-type`* are both structure types, then `subtypep` yields true if the types are the same struct type, or if the right type is a direct or indirect supertype of the left.

The type symbol `struct` is a supertype of all structure types.

9.8.3 Function `typep`

Syntax:

```
(typep object type-symbol)
```

Description:

The `typep` function tests whether the type of *`object`* is a subtype of the type named by *`type-symbol`*.

The following equivalence holds:

```
(typep a b) --> (subtypep (typeof a) b)
```

9.8.4 Macro `typecase`

Syntax:

```
(typecase test-form {(type-sym clause-form*)}*)
```

Description:

The `typecase` macro evaluates *`test-form`* and then successively tests its type against each clause.

Each clause consists of a type symbol *`type-sym`* and zero or more *`clause-form`*s.

The first clause whose *`type-sym`* is a supertype of the type of *`test-form`*'s value is considered to be the matching clause. That clause's *`clause-form`*s are evaluated, and the value of the last form is returned.

If there is no matching clause, or there are no clauses present, or the matching clause has no *`clause-form`*s, then `nil` is returned.

Note: since `t` is the supertype of every type, a clause whose *`type-sym`* is the symbol `t` always matches. If such a clause is placed as the last clause of a `typecase`, it provides a fallback case, whose forms are evaluated if none of the previous clauses match.

9.8.5 Macro `etypecase`

Syntax:

```
(etypecase test-form {(type-sym clause-form*)}*)
```

Description:

The `etypecase` macro is the error-catching variant of `typecase`, similar to the relationship between the `ecaseq` and `caseq` families of macros.

If one of the clauses has a *`type-sym`* which is the symbol `t`, then `etypecase` is precisely equivalent to `typecase`. Otherwise, a clause with a *`type-sym`* of `t` and which throws an exception of type `case-error`, derived from `error`, is appended to the existing clauses, after which the semantics follows that of `typecase`.

9.8.6 Function `built-in-type-p`

Syntax:

```
(built-in-type-p object)
```

Description:

The `built-in-type-p` function returns `t` if *`object`* is a symbol which is the name of a built-in type. For all other objects it returns `nil`.

9.9 Object Equivalence

9.9.1 Functions `identity`, `identity*` and `use`

Syntax:

```
(identity value)
(identity* value*)
(use value)
```

Description:

The `identity` function returns its argument.

If the `identity*` function is given at least one argument, then it returns its leftmost argument, otherwise it returns `nil`.

The `use` function is a synonym of `identity`.

Notes:

The `identity` function is useful as a functional argument, when a transformation function is required, but no transformation is actually desired. In this role, the `use` synonym leads to readable code. For instance:

```
;; construct a function which returns its integer argument
;; if it is odd, otherwise it returns its successor.
;; "If it's odd, use it, otherwise take its successor".

[iff oddp use succ]

;; Applications of the function:

[[iff oddp use succ] 3] -> 3  ;; use applied to 3

[[iff oddp use succ] 2] -> 3  ;; succ applied to 2
```

9.9.2 Functions `null`, `not` and `false`

Syntax:

```
(null value)
(not value)
(false value)
```

Description:

The `null`, `not` and `false` functions are synonyms. They tests whether *value* is the object `nil`. They return `t` if this is the case, `nil` otherwise.

Examples:

```
(null '()) -> t
(null nil) -> t
(null ()) -> t
(false t) -> nil

(if (null x) (format t "x is nil!"))

(let ((list '(b c d)))
  (if (not (memq 'a list))
    (format t "list ~s does not contain the symbol a\n")))
```

9.9.3 Functions `true` and `have`

Syntax:

```
(true value)
```

```
(have value)
```

Description:

The `true` function is the complement of the `null`, `not` and `false` functions. The `have` function is a synonym for `true`.

It return `t` if the *value* is any object other than `nil`. If *value* is `nil`, it returns `nil`.

Note: programs should avoid explicitly testing values with true. For instance `(if x ...)` should be favored over `(if (true x) ...)`. However, the latter is useful with the `ifa` macro because `(ifa (true expr) ...)` binds the `it` variable to the value of *expr*, no matter what kind of form *expr* is, which is not true in the `(ifa expr ...)` form.

Example:

```
;; Compute indices where the list '(1 nil 2 nil 3)
;; has true values:
[where '(1 nil 2 nil 3) true] -> (1 3)
```

9.9.4 Functions `eq`, `eql` and `equal`

Syntax:

```
(eq left-obj right-obj)
(eql left-obj right-obj)
(equal left-obj right-obj)
```

Description:

The principal equality test functions `eq`, `eql` and `equal` test whether two objects are equivalent, using different criteria. They return `t` if the objects are equivalent, and `nil` otherwise.

The `eq` function uses the strictest equivalence test, called implementation equality. The eq function returns `t` if and only if, *left-obj* and *right-obj* are actually the same object. The `eq` test is implemented by comparing the raw bit pattern of the value, whether or not it is an immediate value or a pointer to a heaped object. Two character values are `eq` if they are the same character, and two fixnum integers are `eq` if they have the same value. All other object representations are actually pointers, and are `eq` if and only if they point to the same object in memory. So, for instance, two bignum integers might not be `eq` even if they have the same numeric value, two lists might not be `eq` even if all their corresponding elements are `eq` and two strings might not be eq even if they hold identical text.

The `eql` function is slightly less strict than `eq`. The difference between `eql` and `eq` is that if *left-obj* and *right-obj* are numbers which are of the same kind and have the same numeric value, `eql` returns *t*, even if they are different objects. Note that an integers and a floating-point number are not `eql` even if one has a value which converts to the other: thus, `(eql 0.0 0)` yields `nil`; a comparison expression which finds these numbers equal is `(= 0.0 0)`. The `eql` function also specially treats range objects. Two distinct range objects are `eql` if their corresponding *from* and *to* fields are `eql`. For all other object types, `eql` behaves like `eq`.

The `equal` function is less strict still than `eql`. In general, it recurses into some kinds of aggregate objects to perform a structural equivalence check. For struct types, it also supports customization via equality substitution. See the Equality Substitution section under Structures.

Firstly, if *left-obj* and *right-obj* are `eql` then they are also `equal`, though the converse isn't necessarily the case.

If two objects are both cons cells, then they are equal if their car fields are equal and their cdr fields are equal.

If two objects are vectors, they are equal if they have the same length, and their corresponding elements are equal.

If two objects are strings, they are equal if they are textually identical.

If two objects are functions, they are equal if they have equal environments, and if they have the same code. Two compiled functions are considered to have the same code if and only if they are pointers to the same function. Two interpreted functions are considered to have the same code if their list structure is equal.

Two hashes are equal if they use the same equality (both are :equal-based, or both are :eql-based or else both are :eq-based), if their associated user data elements are equal (see the function hash-userdata), if their sets of keys are identical, and if the data items associated with corresponding keys from each respective hash are equal objects.

Two ranges are equal if their corresponding *to* and *from* fields are equal.

For some aggregate objects, there is no special semantics. Two arguments which are symbols, packages, or streams are equal if and only if they are the same object.

Certain object types have a custom equal function.

9.9.5 Functions neq, neql and nequal

Syntax:

```
(neq left-obj right-obj)
(neql left-obj right-obj)
(nequal left-obj right-obj)
```

Description:

The functions neq, neql and nequal are logically negated counterparts of, respectively, eq, eql and equal.

If eq returns t for a given pair of arguments *left-obj* and *right-obj*, then neq returns nil. Vice versa, if eq returns nil, neq returns t.

The same relationship exits between eql and neql, and between equal and nequal.

9.9.6 Functions meq, meql and mequal

Syntax:

```
(meq left-obj right-obj*)
(meql left-obj right-obj*)
(mequal left-obj right-obj*)
```

Description:

The functions meq, meql and mequal ("member equal" or "multi-equal") provide a particular kind of a generalization of the binary equality functions eq, eql and equal to multiple arguments.

The *left-obj* value is compared to each *right-obj* value using the corresponding binary

equality function. If a match occurs, then `t` is returned, otherwise `nil`.

The traversal of the *right-obj* argument values proceeds from left to right, and stops when a match is found.

9.9.7 Function `less`

Syntax:

```
(less left-obj right-obj)
(less obj obj*)
```

Description:

The `less` function, when called with two arguments, determines whether *left-obj* compares less than *right-obj* in a generic way which handles arguments of various types.

The argument syntax of `less` is generalized. It can accept one argument, in which case it unconditionally returns `t` regardless of that argument's value. If more than two arguments are given, then `less` generalizes in a way which can be described by the following equivalence pattern, with the understanding that each argument expression is evaluated exactly once:

```
(less a b c) <--> (and (less a b) (less b c))
(less a b c d) <--> (and (less a b) (less b c) (less c d))
```

The `less` function is used as the default for the *lessfun* argument of the functions `sort` and `merge`, as well as the *testfun* argument of the `pos-min` and `find-min`.

The `less` function is capable of comparing numbers, characters, symbols, strings, as well as lists and vectors of these. It can also compare buffers.

If both arguments are the same object so that `(eq left-obj right-obj)` holds true, then the function returns `nil` regardless of the type of *left-obj*, even if the function doesn't handle comparing different instances of that type. In other words, no object is less than itself, no matter what it is.

The `less` function pairs with the `equal` function. If values `a` and `b` are objects which are of suitable types to the `less` function, then exactly one of the following three expressions must be true: `(equal a b)`, `(less a b)` or `(less b a)`.

The `less` relation is: antisymmetric, such that if `(less a b)` is true, then then `(less b a)` is false; irreflexive, such that `(less a a)` is false; and transitive, such that `(less a b)` and `(less b c)` imply `(less a c)`.

The following are detailed criteria that `less` applies to arguments of different types and combinations thereof.

If both arguments are numbers or characters, they are compared as if using the < function.

If both arguments are strings, they are compared as if using the `string-lt` function.

If both arguments are symbols, the following rules apply. If the symbols have names which are different, then the result is that of their names being compared by the `string-lt` function. If `less` is passed symbols which have the same name, and neither of these symbols has a home package, then the raw bit patterns of their values are compared as integers: effectively, the object with the lower machine address is considered lesser than the other. If only one of the two same-

named symbols has no home package, then if that symbol is the left argument, `less` returns `t`, otherwise `nil`. If both same-named symbols have home packages, then the result of `less` is that of `string-lt` applied to the names of their respective packages. Thus `a:foo` is less than `z:foo`.

If both arguments are conses, then they are compared as follows:

1. The `less` function is recursively applied to the `car` fields of both arguments. If it yields true, then *`left-obj`* is deemed to be less than *`right-obj`*.
2. Otherwise, if the `car` fields are unequal under the `equal` function, `less` returns `nil`.
3. If the `car` fields are `equal` then `less` is recursively applied to the `cdr` fields of the arguments, and the result of that comparison is returned.

This logic performs a lexicographic comparison on ordinary lists such that for instance `(1 1)` is less than `(1 1 1)` but not less than `(1 0)` or `(1)`.

Note that the empty `nil` list nil compared to a cons is handled by type-based precedence, described below.

Two vectors are compared by `less` lexicographically, similarly to strings. Corresponding elements, starting with element 0, of the vectors are compared until an index position is found where corresponding elements of the two vectors are not *`equal`*. If this differing position is beyond the end of one of the two vectors, then the shorter vector is considered to be lesser. Otherwise, the result of `less` is the outcome of comparing those differing elements themselves with `less`.

Two buffers are also compared by `less` lexicographically, as if they were vectors of integer byte values.

Two ranges are compared by `less` using lexicographic logic similar to conses and vectors. The `from` fields of the ranges are first compared. If they are not `equal`, equal then `less` is applied to those fields and the result is returned. If the `from` fields are `equal`, then `less` is applied to the `to` fields and that result is returned.

If the two arguments are of the above types, but of different types from each other, then `less` resolves the situation based on the following precedence: numbers and characters are less than ranges, which are less than strings, which are less than symbols, which are less than conses, which are less than vectors, which are less than buffers.

Note that since `nil` is a symbol, it is ranked lower than a cons. This interpretation ensures correct behavior when `nil` is regarded as an empty list, since the empty list is lexicographically prior to a nonempty list.

If either argument is a structure for which the `equal` method is defined, the method is invoked on that argument, and the value returned is used in place of that argument for performing the comparison. Structures with no `equal` method cannot participate in a comparison, resulting in an error. See the Equality Substitution section under Structures.

Finally, if either of the arguments has a type other than the above types, the situation is an error.

9.9.8 Function `greater`

Syntax:

```
(greater left-obj right-obj)
```

```
(greater obj obj*)
```

Description:

The `greater` function is equivalent to `less` with the arguments reversed. That is to say, the following equivalences hold:

```
(greater a) <--> (less a) <--> t
(greater a b) <--> (less b a)
(greater a b c ...) <--> (less ... c b a)
```

The `greater` function is used as the default for the *`testfun`* argument of the `pos-max` and `find-max` functions.

9.9.9 Functions `lequal` and `gequal`

Syntax:

```
(lequal obj obj*)
(gequal obj obj*)
```

Description:

The functions `lequal` and `gequal` are similar to `less` and `greater` respectively, but differ in the following respect: when called with two arguments which compare true under the `equal` function, the `lequal` and `gequal` functions return `t`.

When called with only one argument, both functions return `t` and both functions generalize to three or more arguments in the same way as do `less` and `greater`.

9.9.10 Function `copy`

Syntax:

```
(copy object)
```

Description:

The `copy` function duplicates objects of various supported types: sequences, hashes, structures and random states. If *`object`* is `nil`, it returns `nil`. Otherwise, `copy` is equivalent to invoking a more specific copying function according to the type of the argument, as follows:

`cons` `(copy-list object)`

`str` `(copy-str object)`

`vec` `(copy-vec object)`

`hash` `(copy-hash object)`

struct type
`(copy-struct object)`

`fun` `(copy-fun object)`

`buf` `(copy-buf object)`

`carray`
`(copy-carray object)`

`random-state`
`(make-random-state object)`

`tnode` `(copy-tnode object)`

```
tree  (copy-search-tree object)
tree-iter
      (copy-tree-iter object)
cptr  (copy-cptr object)
```

For all other types of *object*, the invocation is erroneous.

Except in the case when *sequence* is `nil`, `copy` returns a value that is distinct from (not `eq` to) *sequence*. This is different from the behavior of `[sequence 0..t]` or `(sub sequence 0 t)` which recognize that they need not make a copy of *sequence*, and just return it.

Note however, that the elements of the returned sequence may be eq to elements of the original sequence. In other words, copy is a deeper copy than just duplicating the `sequence` value itself, but it is not a deep copy.

9.10 List Manipulation

9.10.1 Function `cons`

Syntax:

```
(cons car-value cdr-value)
```

Description:

The `cons` function allocates, initializes and returns a single cons cell. A cons cell has two fields called `car` and `cdr`, which are accessed by functions of the same name, or by the functions `first` and `rest`, which are synonyms for these.

Lists are made up of conses. A (proper) list is either the symbol `nil` denoting an empty list, or a cons cell which holds the first item of the list in its `car`, and the list of the remaining items in `cdr`. The expression `(cons 1 nil)` allocates and returns a single cons cell which denotes the one-element list `(1)`. The `cdr` is `nil`, so there are no additional items.

A cons cell whose `cdr` is an atom other than `nil` is printed with the dotted pair notation. For example the cell produced by `(cons 1 2)` is denoted `(1 . 2)`. The notation `(1 . nil)` is perfectly valid as input, but the cell which it denotes will print back as `(1)`. The notations are equivalent.

The dotted pair notation can be used regardless of what type of object is the cons cell's `cdr`. so that for instance `(a . (b c))` denotes the cons cell whose `car` is the symbol a `a` and whose `cdr` is the list `(b c)`. This is exactly the same thing as `(a b c)`. In other words `(a b ... l m . (n o ... w . (x y z)))` is exactly the same as `(a b ... l m n o ... w x y z)`.

Every list, and more generally cons-cell tree structure, can be written in a "fully dotted" notation, such that there are as many dots as there are cells. For instance the cons structure of the nested list `(1 (2) (3 4 (5)))` can be made more explicit using `(1 . ((2 . nil) . ((3 . (4 . ((5 . nil) . nil))) . nil))))`. The structure contains eight conses, and so there are eight dots in the fully dotted notation.

The number of conses in a linear list like `(1 2 3)` is simply the number of items, so that list in particular is made of three conses. Additional nestings require additional conses, so for instance `(1 2 (3))` requires four conses. A visual way to count the conses from the printed representation is to count the atoms, then add the count of open parentheses, and finally subtract one.

A list terminated by an atom other than `nil` is called an improper list, and the dot notation is extended to cover improper lists. For instance `(1 2 . 3)` is an improper list of two elements, terminated by `3`, and can be constructed using `(cons 1 (cons 2 3))`. The fully dotted notation for this list is `(1 . (2 . 3))`.

9.10.2 Function `atom`

Syntax:

```
(atom value)
```

Description:

The `atom` function tests whether *`value`* is an atom. It returns `t` if this is the case, `nil` otherwise. All values which are not cons cells are atoms.

`(atom x)` is equivalent to `(not (consp x))`.

Examples:

```
(atom 3) -> t
(atom (cons 1 2)) -> nil
(atom "abc") -> t
(atom '(3)) -> nil
```

9.10.3 Function `consp`

Syntax:

```
(consp value)
```

Description:

The `consp` function tests whether *`value`* is a cons. It returns `t` if this is the case, `nil` otherwise.

`(consp x)` is equivalent to `(not (atom x))`.

Nonempty lists test positive under `consp` because a list is represented as a reference to the first cons in a chain of one or more conses.

Note that a lazy cons is a cons and satisfies the `consp` test. See the function `make-lazy-cons` and the macro `lcons`.

Examples:

```
(consp 3) -> nil
(consp (cons 1 2)) -> t
(consp "abc") -> nil
(consp '(3)) -> t
```

9.10.4 Accessors `car` and `first`

Syntax:

```
(car object)
(first object)
(set (car object) new-value)
(set (first object) new-value)
```

Description:

The functions car and first are synonyms.

If *object* is a cons cell, these functions retrieve the car field of that cons cell. (car (cons 1 2)) yields 1.

For programming convenience, *object* may be of several other kinds in addition to conses.

(car nil) is allowed, and returns nil.

object may also be a vector or a string. If it is an empty vector or string, then nil is returned. Otherwise the first character of the string or first element of the vector is returned.

object may be a structure. The car operation is possible if the object has a car method. If so, car invokes that method and returns whatever the method returns. If the structure has no car method, but has a lambda method, then the car function calls that method with one argument, that being the integer zero. Whatever the method returns, car returns. If neither method is defined, an error exception is thrown.

A car form denotes a valid place whenever *object* is a valid argument for the rplaca function. Modifying the place denoted by the form is equivalent to invoking rplaca with *object* as the left argument, and the replacement value as the right argument. It takes place in the manner given under the description rplaca function, and obeys the same restrictions.

A car form supports deletion. The following equivalence then applies:

```
(del (car place)) <--> (pop place)
```

This implies that deletion requires the argument of the car form to be a place, rather than the whole form itself. In this situation, the argument place may have a value which is nil, because pop is defined on an empty list.

The abstract concept behind deleting a car is that physically deleting this field from a cons, thereby breaking it in half, would result in just the cdr remaining. Though fragmenting a cons in this manner is impossible, deletion simulates it by replacing the place which previously held the cons, with that cons' cdr field. This semantics happens to coincide with deleting the first element of a list by a pop operation.

9.10.5 Accessors cdr and rest

Syntax:

```
(cdr object)
(rest object)
(set (cdr object) new-value)
(set (rest object) new-value)
```

Description:

The functions cdr and rest are synonyms.

If *object* is a cons cell, these functions retrieve the cdr field of that cons cell. (cdr (cons 1 2)) yields 2.

For programming convenience, *object* may be of several other kinds in addition to conses.

`(cdr nil)` is allowed, and returns `nil`.

`object` may also be a vector or a string. If it is a nonempty string or vector containing at least two items, then the remaining part of the object is returned, with the first element removed. For example `(cdr "abc")` yields `"bc"`. If *`object`* is a one-element vector or string, or an empty vector or string, then `nil` is returned. Thus `(cdr "a")` and `(cdr "")` both result in `nil`.

If *`object`* is a structure, then `cdr` requires it to support either the `cdr` method or the `lambda` method. If both are present, `cdr` is used. When the `cdr` function uses the `cdr` method, it invokes it with no arguments. Whatever value the method returns becomes the return value of `cdr`. When `cdr` invokes a structure's `lambda` method, it passes as the argument the range object `#R(1 t)`. Whatever the `lambda` method returns becomes the return value of `cdr`.

The invocation syntax of a `cdr` or `rest` form is a syntactic place. The place is semantically correct if *`object`* is a valid argument for the `rplacd` function. Modifying the place denoted by the form is equivalent to invoking `rplacd` with *`object`* as the left argument, and the replacement value as the right argument. It takes place in the manner given under the description `rplacd` function, and obeys the same restrictions.

A `cdr` place supports deletion, according to the following near equivalence:

```
(del (cdr place)) <--> (prog1 (cdr place)
                              (set place (car place)))
```

The `place` expression is evaluated only once.

Note that this is symmetric with the delete semantics of `car` in that the cons stored in `place` goes away, as does the `cdr` field, leaving just the `car`, which takes the place of the original cons.

Example:

Walk every element of the list `(1 2 3)` using a `for` loop:

```
(for ((i '(1 2 3))) (i) ((set i (cdr i)))
  (print (car i) *stdout*)
  (print #\newline *stdout*))
```

The variable `i` marches over the cons cells which make up the "backbone" of the list. The elements are retrieved using the `car` function. Advancing to the next cell is achieved using `(cdr i)`. If `i` is the last cell in a (proper) list, `(cdr i)` yields `nil` and so `i` becomes `nil`, the loop guard expression `i` fails and the loop terminates.

9.10.6 Functions `rplaca` and `rplacd`

Syntax:

```
(rplaca object new-car-value)
(rplacd object new-cdr-value)
```

Description:

If `object` is a cons cell or lazy cons cell, then `rplaca` and `rplacd` functions assign new values into the `car` and `cdr` fields of the *`object`*. In addition, these functions are meaningful for other kinds of objects also.

Note that, except for the difference in return value, `(rplaca x y)` is the same as the more generic `(set (car x) y)`, and likewise `(rplacd x y)` can be written as `(set (cdr x) y)`.

The `rplaca` and `rplacd` functions return *cons*. Note: In **TXR** versions 89 and earlier, these functions returned the new value. The behavior was undocumented.

The *cons* argument does not have to be a cons cell. Both functions support meaningful semantics for vectors and strings. If *cons* is a string, it must be modifiable.

The `rplaca` function replaces the first element of a vector or first character of a string. The vector or string must be at least one element long.

The `rplacd` function replaces the suffix of a vector or string after the first element with a new suffix. The *new-cdr-value* must be a sequence, and if the suffix of a string is being replaced, it must be a sequence of characters. The suffix here refers to the portion of the vector or string after the first element.

It is permissible to use `rplacd` on an empty string or vector. In this case, *new-cdr-value* specifies the contents of the entire string or vector, as if the operation were done on a nonempty vector or string, followed by the deletion of the first element.

The *object* argument may be a structure. In the case of `rplaca`, the structure must have a defined `rplaca` method or else, failing that, a `lambda-set` method. The first of these methods which is available, in the given order, is used to perform the operation. Whatever the respective method returns, If the `lambda-set` method is used, it is called with two arguments (in addition to `object`): the integer zero, and *new-car-value*.

In the case of `rplacd`, the structure must have a defined `rplacd` method or else, failing that, a `lambda-set` method. The first of these methods which is available, in the given order, is used to perform the operation. Whatever the respective method returns, If the `lambda-set` method is used, it is called with two arguments (in addition to `object`): the range value `#R(1 t)` and *new-car-value*.

9.10.7 Accessors `second`, `third`, `fourth`, `fifth`, `sixth`, `seventh`, `eighth`, `ninth` and `tenth`

Syntax:

```
(first object)
(second object)
(third object)
(fourth object)
(fifth object)
(sixth object)
(seventh object)
(eighth object)
(ninth object)
(tenth object)
(set (first object) new-value)
(set (second object) new-value)
...
(set (tenth object) new-value)
```

Description:

Used as functions, these accessors retrieve the elements of a sequence by position. If the sequence

is shorter than implied by the position, these functions return `nil`.

When used as syntactic places, these accessors denote the storage locations by position. The location must exist, otherwise an error exception results. The places support deletion.

Examples:

```
(third '(1 2)) -> nil
(second "ab") -> #\b
(third '(1 2 . 3)) -> **error, improper list*

(let ((x (copy "abcd")))
  (inc (third x))
  x) -> "abce"
```

9.10.8 Functions `append` and `nconc`

Syntax:

```
(append [sequence*])
(nconc [sequence*])
```

Description:

The `append` function creates a new object which is a catenation of the *`list`* arguments. All arguments are optional; `append` produces the empty list, and if a single argument is specified, that argument is returned.

If two or more arguments are present, then the situation is identified as one or more *`sequence`* arguments followed by *`last-arg`*. The *`sequence`* arguments must be sequences; *`last-arg`* may be a sequence or atom.

The `append` operation over three or more arguments is left-associative, such that `(append x y z)` is equivalent to both `(append (append x y) z)` and `(append x (append z y))`.

This allows the catenation of an arbitrary number of arguments to be understood in terms of a repeated application of the two-argument case, whose semantics is given by these rules:

1. `nil` catenates with `nil` to produce `nil`:
   ```
   (append nil nil) -> nil
   ```
2. `nil` catenates with a proper or improper list, producing that list itself:
   ```
   (append nil '(1 2)) -> (1 2)
   (append nil '(1 2 . 3)) -> (1 2 . 3)
   ```
3. A proper list catenates with `nil`, producing that list itself:
   ```
   (append '(1 2) nil) -> (1 2)
   ```
4. A proper list catenates with an atom, producing an improper list terminated by that atom, whether or not that atom is a sequence:
   ```
   (append '(1 2) #(3)) -> (1 2 . #(3))
   (append '(1 2) 3) -> (1 2 . 3)
   ```
5. A non-list sequence catenates with another sequence into a sequence, producing a sequence which contains the elements of both, of the same kind as the left sequence. The elements must be compatible; a string can only catenate with a sequence of characters.
   ```
   (append #(1 2) #(3 4)) -> #(1 2 3 4)
   (append "ab" "cd") -> "abcd"
   ```

```
(append "ab" #(#\c #\d)) -> "abcd"
(append "ab" #(3 4)) -> ;; error
```

6. A non-list sequence catenates with an atom if it is a suitable element type for that kind of sequence. The resulting sequence is of the same kind, and includes that atom:
```
(append #(1 2) 3) -> #(1 2 3)
(append "ab" #  (append "ab" 3) -> ;; error
```
7. If an improper list is catenated with any object, the catenation takes place between the terminating atom of that list and that object. This requires the terminating atom to be a sequence. If the catenation is possible, then the result is a new improper list which is a copy of the original, but with the terminating atom replaced by a catenation of that atom and the object:
```
(append '(1 2 . "ab") "c") -> (1 2 . "abc")
(append '(1 2 . "ab") '(2 3)) -> ;; error
```
8. A non-sequence atom doesn't catenate; the situation is erroneous:
```
(append 1 2) -> ;; error
(append '(1 . 2) 3) -> ;; error
```

If N arguments are specified, where N > 1, then the first N-1 arguments must be proper lists. Copies of these lists are catenated together. The last argument N, shown in the above syntax as *last-arg*, may be any kind of object. It is installed into the `cdr` field of the last cons cell of the resulting list. Thus, if argument N is also a list, it is catenated onto the resulting list, but without being copied. Argument N may be an atom other than `nil`; in that case `append` produces an improper list.

The `nconc` function works like `append`, but may destructively manipulate any of the input objects.

Examples:

```
;; An atom is returned.
(append 3) -> 3

;; A list is also just returned: no copying takes place.
;; The eq function can verify that the same object emerges
;; from append that went in.
(let ((list '(1 2 3)))
  (eq (append list) list)) -> t

(append '(1 2 3) '(4 5 6) 7) -> '(1 2 3 4 5 6 . 7))

;; the (4 5 6) tail of the resulting list is the original
;; (4 5 6) object, shared with that list.

(append '(1 2 3) '(4 5 6)) -> '(1 2 3 4 5 6)

(append nil) -> nil

;; (1 2 3) is copied: it is not the last argument
(append '(1 2 3) nil) -> (1 2 3)

;; empty lists disappear
(append nil '(1 2 3) nil '(4 5 6)) -> (1 2 3 4 5 6)
(append nil nil nil) -> nil
```

```
;; atoms and improper lists other than in the last position
;; are erroneous
(append '(a . b) 3 '(1 2 3)) -> **error**

;; sequences other than lists can be catenated.
(append "abc" "def" "g" #\h) -> "abcdefgh"

;; lists followed by non-list sequences end with non-list
;; sequences catenated in the terminating atom:
(append '(1 2) '(3 4) "abc" "def") -> (1 2 3 4 . "abcdef")
```

9.10.9 Function `append*`

Syntax:

```
(append* [list*])
```

Description:

The `append*` function lazily catenates lists.

If invoked with no arguments, it returns `nil`. If invoked with a single argument, it returns that argument.

Otherwise, it returns a lazy list consisting of the elements of every *`list`* argument from left to right.

Arguments other than the last are treated as lists, and traversed using `car` and `cdr` functions to visit their elements.

The last argument isn't traversed: rather, that object itself becomes the `cdr` field of the last cons cell of the lazy list constructed from the previous arguments.

9.10.10 Functions `revappend` and `nreconc`

Syntax:

```
(revappend list1 list2)
(nreconc list1 list2)
```

Description:

The `revappend` function returns a list consisting of `list2` appended to a reversed copy of *`list1`*. The returned object shares structure with *`list2`*, which is unmodified.

The `nreconc` function behaves similarly, except that the returned object may share structure with not only *`list2`* but also *`list1`*, which is modified.

9.10.11 Function `list`

Syntax:

```
(list value*)
```

Description:

The `list` function creates a new list, whose elements are the argument values.

Examples:

```
(list) -> nil
(list 1) -> (1)
(list 'a 'b) -> (a b)
```

9.10.12 Function `list*`

Syntax:

```
(list* value*)
```

Description:

The `list*` function is a generalization of cons. If called with exactly two arguments, it behaves exactly like cons: `(list* x y)` is identical to `(cons x y)`. If three or more arguments are specified, the leading arguments specify additional atoms to be consed to the front of the list. So for instance `(list* 1 2 3)` is the same as `(cons 1 (cons 2 3))` and produces the improper list `(1 2 . 3)`. Generalizing in the other direction, `list*` can be called with just one argument, in which case it returns that argument, and can also be called with no arguments in which case it returns `nil`.

Examples:

```
(list*) -> nil
(list* 1) -> 1
(list* 'a 'b) -> (a . b)
(list* 'a 'b 'c) -> (a b . c)
```

Dialect Note:

Note that unlike in some other Lisp dialects, the effect of `(list* 1 2 x)` can also be obtained using `(list 1 2 . x)`. However, `(list* 1 2 (func 3))` cannot be rewritten as `(list 1 2 . (func 3))` because the latter is equivalent to `(list 1 2 func 3)`.

9.10.13 Accessor `sub-list`

Syntax:

```
(sub-list list [from [to]])
(set (sub-list list [from [to]]) new-value)
```

Description:

The `sub-list` function has the same parameters and semantics as the `sub` function, except that it operates on its *`list`* argument using list operations, and assumes that *`list`* is terminated by `nil`.

If a `sub-list` form is used as a place, then the *`list`* argument form must also be a place.

The `sub-list` place denotes a subrange of *`list`* as if it were a storage location. The previous value of this location, if needed, is fetched by a call to `sub-list`. Storing *`new-value`* to the place is performed by a call to `replace-list`. The return value of *`replace-list`* is stored into *`list`*. In an update operation which accesses the prior value and stores a new value, the arguments *`list`*, *`from`*, *`to`* and *`new-value`* are evaluated once.

9.10.14 Function `replace-list`

Syntax:

```
(replace-list list item-sequence [from [to]])
```

Description:

The `replace-list` function is like the `replace` function, except that it operates on its *`list`* argument using list operations. It assumes that *`list`* is terminated by `nil`, and that it is made of cells which can be mutated using `rplaca`.

9.10.15 Functions `listp` and `proper-list-p`

Syntax:

```
(listp value)
(proper-list-p value)
```

Description:

The `listp` and `proper-list-p` functions test, respectively, whether *`value`* is a list, or a proper list, and return `t` or `nil` accordingly.

The `listp` test is weaker, and executes without having to traverse the object. The value produced by the expression `(listp x)` is the same as that of `(or (null x) (consp x))`, except that `x` is evaluated only once. The empty list `nil` is a list, and a cons cell is a list.

The `proper-list-p` function returns `t` only for proper lists. A proper list is either `nil`, or a cons whose `cdr` is a proper list. `proper-list-p` traverses the list, and its execution will not terminate if the list is circular.

These functions return `nil` for list-like sequences that are not made of actual `cons` cells.

Dialect Note: in **TXR** 137 and older, `proper-list-p` is called `proper-listp`. The name was changed for adherence to conventions and compatibility with other Lisp dialects, like Common Lisp. However, the function continues to be available under the old name. Code that must run on **TXR** 137 and older installations should use `proper-listp`, but its use going forward is deprecated.

9.10.16 Function `endp`

Syntax:

```
(endp object)
```

Description:

The `endp` function returns `t` if *`object`* is the object `nil`.

If *`object`* is a cons cell, then `endp` returns `t`.

Otherwise, `endp` function throws an exception.

9.10.17 Function `length-list`

Syntax:

```
(length-list list)
```

Description:

The `length-list` function returns the length of *`list`*, which may be a proper or improper list. The length of a list is the number of conses in that list.

9.10.18 Function `copy-list`

Syntax:

```
(copy-list list)
```

Description:

The `copy-list` function which returns a list similar to *`list`*, but with a newly allocated cons-cell structure.

If *`list`* is an atom, it is simply returned.

Otherwise, *`list`* is a cons cell, and `copy-list` returns the same object as the expression `(cons (car list) (copy-list (cdr list)))`.

Note that the object `(car list)` is not deeply copied, but only propagated by reference into the new list. `copy-list` produces a new list structure out of the same items that are in *`list`*.

Dialect Note:

Common Lisp does not allow the argument to be an atom, except for the empty list `nil`.

9.10.19 Function `copy-cons`

Syntax:

```
(copy-cons cons)
```

Description:

The `copy-cons` function creates and returns a new object that is a replica of *`cons`*.

The *`cons`* argument must be either a `cons` cell, or else a lazy cons: an object of type `lcons`.

A new cell of the same type as *`cons`* is created, and all of its fields are initialized by copying the corresponding fields from *`cons`*.

If *`cons`* is lazy, the newly created object is in the same state as the original. If the original has not yet been updated and thus has an update function, the copy also has not yet been updated and has the same update function.

9.10.20 Function `copy-tree`

Syntax:

```
(copy-tree obj)
```

Description:

The `copy-tree` function returns a copy of *`obj`* which represents an arbitrary `cons`-cell-based structure.

The cell structure of *`obj`* is traversed and a similar structure is constructed, but without regard for substructure sharing or circularity.

More precisely, if *obj* is an atom, then it is returned. If it is an ordinary `cons` cell, then `copy-tree` is recursively applied to the `car` and `cdr` fields to produce their individual replicas. A new `cons` cell is then produced from the replicated `car` and `cdr`. If *obj* is a lazy `cons`, then just like in the ordinary `cons` case, the `car` and `cdr` fields are duplicated with a recursive call to `copy-tree`. Then, a lazy `cons` is created from these replicated fields. If *cell* has an update function, then the newly created lazy `cons` has the same update function; the function isn't copied.

Like `copy-cons`, the `copy-tree` function doesn't trigger the update of lazy conses. The copies of lazy conses which have not been updated are also conses which have not been updated.

9.10.21 Functions `reverse` and `nreverse`

Syntax:

```
(reverse list)
(nreverse list)
```

Description:

Description:

The functions `reverse` and `nreverse` produce an object which contains the same items as proper list *list*, but in reverse order. If *list* is `nil`, then both functions return `nil`.

The `reverse` function is non-destructive: it creates a new list.

The `nreverse` function creates the structure of the reversed list out of the cons cells of the input list, thereby destructively altering it (if it contains more than one element). How `nreverse` uses the material from the original list is unspecified. It may rearrange the cons cells into a reverse order, or it may keep the structure intact, but transfer the `car` values among cons cells into reverse order. Other approaches are possible.

9.10.22 Accessor `nthlast`

Syntax:

```
(nthlast index list)
(set (nthlast index list) new-value)
```

Description:

The `nthlast` function retrieves the n-th last cons cell of a list, indexed from one. The *index* parameter must be a an integer. If *index* is positive and so large that it specifies a nonexistent cons beyond the beginning of the list, `nthlast` returns *list*. Effectively, values of *index* larger than the length of the list are clamped to the length. If *index* is negative, then `nthlast` yields nil. An *index* value of zero retrieves the terminating atom of *list* or else the value *list* itself, if *list* is an atom.

The following equivalences hold:

```
(nthlast 1 list) <--> (last list)
```

An `nthlast` place designates the storage location which holds the n-th cell, as indicated by the value of *index*.

A negative *index* doesn't denote a place.

A positive *index* greater than the length of the list is treated as if it were equal to the length of the list.

If *list* is itself a syntactic place, then the *index* value *n* is permitted for a list of length *n*. This index value denotes the *list* place itself. Storing to this value overwrites *list*. If *list* isn't a syntactic place, then storing to position *n* isn't permitted.

If *list* is of length zero, or an atom (in which case its length is considered to be zero) then the above remarks about position *n* apply to an *index* value of zero: if *list* is a syntactic place, then the position denotes *list* itself, otherwise the position doesn't exist as a place.

If *list* contains one or more elements, then *index* value of zero denotes the `cdr` field of its last cons cell. Storing a value to this place overwrites the terminating atom.

9.10.23 Accessor `butlastn`

Syntax:

```
(butlastn num list)
(set (butlastn num list) new-value )
```

Description:

The `butlastn` function calculates that initial portion of *list* which excludes the last *num* elements.

Note: the `butlastn` function doesn't support non-list sequences as sequences; it treats them as the terminating atom of a zero-length improper list. The `butlast` sequence function supports non-list sequences. If `x` is a list, then the following equivalence holds:

```
(butlastn n x)  <-->  (butlast x n)
```

If *num* is zero, or negative, then `butlastn` returns *list*.

If *num* is positive, and meets or exceeds the length of *list*, then `butlastn` returns `nil`.

If a `butlastn` form is used as a syntactic place, then *list* must be a place. Assigning to the form causes *list* to be replaced with a new list which is a catenation of the new value and the last *num* elements of the original list, according to the following equivalence:

```
(set (butlastn n x) v)

<-->

(progn (set x (append v (nthlast n x))) v)
```

except that `n`, `x` and `v` are evaluated only once, in left-to-right order.

9.10.24 Accessor `nth`

Syntax:

```
(nth index object)
(set (nth index object) new-value)
```

Description:

The `nth` function performs random access on a list, retrieving the n-th element indicated by the zero-based index value given by *index*. The *index* argument must be a nonnegative integer.

If *index* indicates an element beyond the end of the list, then the function returns `nil`.

The following equivalences hold:

```
(nth 0 list) <--> (car 0) <--> (first list)
(nth 1 list) <--> (cadr list) <--> (second list)
(nth 2 list) <--> (caddr list) <--> (third list)

(nth x y) <--> (car (nthcdr x y))
```

9.10.25 Accessor `nthcdr`

Syntax:

```
(nthcdr index list)
(set (nthcdr index list) new-value)
```

Description:

The `nthcdr` function retrieves the n-th cons cell of a list, indexed from zero. The *index* parameter must be a nonnegative integer. If *index* specifies a nonexistent cons beyond the end of the list, then `nthcdr` yields nil.

The following equivalences hold:

```
(nthcdr 0 list) <--> list
(nthcdr 1 list) <--> (cdr list)
(nthcdr 2 list) <--> (cddr list)

(car (nthcdr x y)) <--> (nth x y)
```

An `nthcdr` place designates the storage location which holds the n-th cell, as indicated by the value of *index*. Indices beyond the last cell of *list* do not designate a valid place. If *list* is itself a place, then the zeroth index is permitted and the resulting place denotes *list*. Storing a value to `(nthcdr 0 list)` overwrites *list*. Otherwise if *list* isn't a syntactic place, then the zeroth index does not designate a valid place; *index* must have a positive value. A `nthcdr` place does not support deletion.

Dialect Note:

In Common Lisp, `nthcdr` is only a function, not an accessor; `nthcdr` forms do not denote places.

9.10.26 Function `tailp`

Syntax:

```
(tailp object list)
```

Description:

The `tailp` function tests whether *object* is a tail of *list*. This means that *object* is either *list* itself, or else one of the `cons` cells of *list* or else the terminating atom of *list*.

More formally, a recursive definition follows. If *object* and *list* are the same object (thus equal under the `eq` function) then `tailp` returns `t`. If *list* is an atom, and is not *object*, then the function returns `nil`. Otherwise, *list* is a `cons` that is not *object* and `tailp` yields the same value as the `(tailp object (cdr list))` expression.

9.10.27 Accessors `caar`, `cadr`, `cdar`, `cddr`, ..., `cddddddr`

Syntax:

```
(caar object)
(cadr object)
(cdar object)
(cddr object)
...
(cdddr object)
(set (caar object) new-value)
(set (cadr object) new-value)
...
```

Description:

The *a-d accessors* provide a shorthand notation for accessing two to five levels deep into a cons-cell-based tree structure. For instance, the the equivalent of the nested function call expression `(car (car (cdr object)))` can be achieved using the single function call `(caadr object)`. The symbol names of the a-d accessors are a generalization of the words "car" and "cdr". They encode the pattern of `car` and `cdr` traversal of the structure using a sequence of the letters `a` and `d` placed between `c` and `r`. The traversal is encoded in right-to-left order, so that `cadr` indicates a traversal of the `cdr` link, followed by the `car`. This order corresponds to the nested function call notation, which also encodes the traversal right-to-left. The following diagram illustrates the straightforward relationship:

```
(cdr (car (cdr x)))
  ^    ^    ^
  |   /     |
  |  /     /
  | / ____/
  || /
(cdadr x)
```

TXR Lisp provides all possible a-d accessors up to five levels deep, from `caar` all the way through `cddddddr`.

Expressions involving a-d accessors are places. For example, `(caddr x)` denotes the same place as `(car (cddr x))`, and `(cdadr x)` denotes the same place as `(cdr (cadr x))`.

The a-d accessor places support deletion, with semantics derived from the deletion semantics of the `car` and `cdr` places. For example, `(del (caddr x))` means the same as `(del (car (cddr x)))`.

9.10.28 Functions `cyr` and `cxr`

Syntax:

```
(cyr address object)
(cxr address object)
```

Description:

The `cyr` and `cxr` functions provide `car`/`cdr` navigation of tree structure driven by numeric address given by the *`address`* argument.

The *`address`* argument can express any combination of the application of `car` and `cdr` functions, including none at all.

The difference between `cyr` and `cxr` is the bit order of the encoding. Under `cyr`, the most significant bit of the encoding given in *`address`* indicates the initial `car`/`cdr` navigation, and the least significant bit gives the final one. Under `cxr`, it is opposite.

Both functions require *`address`* to be a positive integer. Any other argument raises an error.

Under both functions, the *`address`* value `1` encodes the `identity` operation: no `car`/`cdr`

9.10.29 Functions `flatten` and `flatten*`

Syntax:

```
(flatten list)
(flatten* list)
```

Description:

The `flatten` function produces a list whose elements are all of the non-`nil` atoms contained in the structure of *`list`*.

The `flatten*` function works like `flatten` except that it produces a lazy list. It can be used to lazily flatten an infinite lazy structure.

Examples:

```
(flatten '(1 2 () (3 4))) -> (1 2 3 4)

;; equivalent to previous, since
;; nil is the same thing as ()
(flatten '(1 2 nil (3 4))) -> (1 2 3 4)

(flatten nil) -> nil

(flatten '(((()) ()))) -> nil
```

9.10.30 Functions `flatcar` and `flatcar*`

Syntax:

```
(flatcar tree)
(flatcar* tree)
```

Description:

The `flatcar` function produces a list of all the atoms contained in the tree structure *`tree`*, in the order in which they appear, when the structure is traversed left to right.

This list includes those `nil` atoms which appear in `car` fields.

The list excludes `nil` atoms which appear in `cdr` fields.

The `flatcar*` function works like `flatcar` except that it produces a lazy list. It can be used to lazily flatten an infinite lazy structure.

Examples:

```
(flatcar '(1 2 () (3 4))) -> (1 2 nil 3 4)

(flatcar '(a (b . c) d (e) (((f)) . g) (nil . z) nil . h))

--> (a b c d e f g nil z nil h)
```

9.10.31 Function `tree-find`

Syntax:

```
(tree-find obj tree test-function)
```

Description:

The `tree-find` function searches *`tree`* for an occurrence of *`obj`*. Tree can be any atom, or a cons. If *`tree`* it is a cons, it is understood to be a proper list whose elements are also trees.

The equivalence test is performed by *`test-function`* which must take two arguments, and has conventions similar to `eq`, `eql` or `equal`.

`tree-find` works as follows. If *`tree`* is equivalent to *`obj`* under *`test-function`*, then `t` is returned to announce a successful finding. If this test fails, and *`tree`* is an atom, `nil` is returned immediately to indicate that the find failed. Otherwise, *`tree`* is taken to be a proper list, and `tree-find` is recursively applied to each element of the list in turn, using the same *`obj`* and *`test-function`* arguments, stopping at the first element which returns a non-`nil` value.

9.10.32 Functions `memq`, `memql` and `memqual`

Syntax:

```
(memq object list)
(memql object list)
(memqual object list)
```

Description:

The `memq`, `memql` and `memqual` functions search *`list`* for a member which is, respectively, `eq`, `eql` or `equal` to *`object`*. (See the `eq`, `eql` and `equal` functions above.)

If no such element found, `nil` is returned.

Otherwise, that suffix of *`list`* is returned whose first element is the matching object.

9.10.33 Functions `member` and `member-if`

Syntax:

```
(member key sequence [testfun [keyfun]])
(member-if predfun sequence [keyfun])
```

Description:

The `member` and `member-if` functions search through *`sequence`* for an item which matches a key, or satisfies a predicate function, respectively.

The *keyfun* argument specifies a function which is applied to the elements of the sequence to produce the comparison key. If this argument is omitted, then the untransformed elements of the sequence themselves are examined.

The `member` function's *testfun* argument specifies the test function which is used to compare the comparison keys taken from the sequence to the search key. If this argument is omitted, then the `equal` function is used. If `member` does not find a matching element, it returns `nil`. Otherwise it returns the suffix of *sequence* which begins with the matching element.

The `member-if` function's *predfun* argument specifies a predicate function which is applied to the successive comparison keys pulled from the sequence by applying the key function to successive elements. If no match is found, then `nil` is returned, otherwise what is returned is the suffix of *sequence* which begins with the matching element.

9.10.34 Functions `rmemq`, `rmemql`, `rmemqual`, `rmember` and `rmember-if`

Syntax:

```
(rmemq object list)
(rmemql object list)
(rmemqual object list)
(rmember key sequence [testfun [keyfun]])
(rmember-if predfun sequence [keyfun])
```

Description:

These functions are counterparts to `memq`, `memql`, `memqual`, `member` and `member-if` which look for the rightmost element which matches *object*, rather than for the leftmost element.

9.10.35 Functions `conses` and `conses*`

Syntax:

```
(conses list)
(conses* list)
```

Description:

These functions return a list whose elements are the conses which make up *list*. The `conses*` function does this in a lazy way, avoiding the computation of the entire list: it returns a lazy list of the conses of *list*. The `conses` function computes the entire list before returning.

The input *list* may be proper or improper.

The first cons of *list* is that *list* itself. The second cons is the rest of the list, or `(cdr list)`. The third cons is `(cdr (cdr list))` and so on.

Example:

```
(conses '(1 2 3)) -> ((1 2 3) (2 3) (3))
```

Dialect Note:

These functions are useful for simulating the `maplist` function found in other dialects like Common Lisp.

TXR Lisp's `(conses x)` can be expressed in Common Lisp as `(maplist #'identity x)`.

Conversely, the Common Lisp operation `(maplist function list)` can be computed in **TXR Lisp** as `(mapcar function (conses list))`.

More generally, the Common Lisp operation

```
(maplist function list0 list1 ... listn)
```

can be expressed as:

```
(mapcar function (conses list0)
                 (conses list1) ... (conses listn))
```

9.10.36 Function `delcons`

Syntax:

```
(delcons cons list)
```

Description:

The `delcons` function destructively removes a cons cell from a list. The *`list`* is searched to see whether one of its cons cells is the same object as *`cons`*. If so, that cell is removed from the list.

The *`list`* argument may be a proper or improper list, possibly empty. It may also be an atom other than `nil`, which is regarded as being, effectively, an empty improper list terminated by that atom.

The operation of `delcons` is divided into the following three cases. If *`cons`* is the first cons cell of *`list`*, then the `cdr` of *`list`* is returned. If *`cons`* is the second or subsequent cons of *`list`*, then *`list`* is destructively altered to remove *`cons`* and then returned. This means that the `cdr` field of the predecessor of *`cons`* is altered from referencing *`cons`* to referencing `(cdr cons)` instead. The returned value is the same cons cell as *`list`*. The third case occurs when *`cons`* is not found in *`list`*. In this situation, *`list`* is returned unchanged.

Examples:

```
(let ((x (list 1 2 3)))
  (delcons x x))
-> (2 3)

(let ((x (list 1 2 . 3)))
  (delcons (cdr x) x))
-> (1 . 3)
```

9.11 Association Lists

Association lists are ordinary lists formed according to a special convention. Firstly, any empty list is a valid association list. A nonempty association list contains only cons cells as the key elements. These cons cells are understood to represent key/value associations, hence the name "association list".

9.11.1 Function `assoc`

Syntax:

```
(assoc key alist)
```

Description:

The `assoc` function searches an association list *`alist`* for a cons cell whose `car` field is equivalent to *`key`* under the `equal` function. The first such cons is returned. If no such cons is found, `nil` is returned.

9.11.2 Functions `assq` and `assql`

Syntax:

```
(assq key alist)
(assql key alist)
```

Description:

The `assq` and `assql` functions are very similar to `assoc`, with the only difference being that they determine equality using, respectively, the `eq` and `eql` functions rather than `equal`.

9.11.3 Functions `rassq`, `rassql` and `rassoc`

Syntax:

```
(rassq value alist)
(rassql value alist)
(rassoc value alist)
```

Description:

The `rassq`, `rassql` and `rassoc` functions are reverse lookup counterparts to `assql` and `assoc`. When searching, they examine the `cdr` field of the pairs of *`alist`* rather than the `car` field.

The `rassoc` function searches association list *`alist`* for a cons whose `cdr` field equivalent to *`value`* according to the `equal` function. If such a cons is found, it is returned. Otherwise `nil` is returned.

The `rassq` and `rassql` functions search in the same way as `rassoc` but compares values using, respectively, `eq` and `eql`.

9.11.4 Function `acons`

Syntax:

```
(acons car cdr alist)
```

Description:

The `acons` function constructs a new alist by consing a new cons to the front of *`alist`*. The following equivalence holds:

```
(acons car cdr alist) <--> (cons (cons car cdr) alist)
```

9.11.5 Function `acons-new`

Syntax:

```
(acons-new car cdr alist)
```

Description:

The `acons-new` function searches *`alist`*, as if using the assoc function, for an existing cell which matches the key provided by the car argument. If such a cell exists, then its cdr field is

overwritten with the *cdr* argument, and then the *alist* is returned. If no such cell exists, then a new list is returned by adding a new cell to the input list consisting of the *car* and *cdr* values, as if by the `acons` function.

9.11.6 Function `aconsql-new`

Syntax:

```
(aconsql-new car cdr alist)
```

Description:

The `aconsql-new` function has similar same parameters and semantics as `acons-new`, except that the `eql` function is used for equality testing. Thus, the list is searched for an existing cell as if using the `assql` function rather than `assoc`.

9.11.7 Function `alist-remove`

Syntax:

```
(alist-remove alist key*)
```

Description:

The `alist-remove` function takes association list *alist* and produces a duplicate from which cells matching any of the specified *key*s have been removed.

9.11.8 Function `alist-nremove`

Syntax:

```
(alist-nremove alist key*)
```

Description:

The `alist-nremove` function is like `alist-remove`, but potentially destructive. The input list *alist* may be destroyed and its structural material reused to form the output list. The application should not retain references to the input list.

9.11.9 Function `copy-alist`

Syntax:

```
(copy-alist alist)
```

Description:

The `copy-alist` function duplicates `alist`. Unlike `copy-list`, which only duplicates list structure, `copy-alist` also duplicates each cons cell of the input alist. That is to say, each element of the output list is produced as if by the `copy-cons` function applied to the corresponding element of the input list.

9.11.10 Function `pairlis`

Syntax:

```
(pairlis keys values [alist])
```

Description:

The `pairlis` function returns an association list consisting of pairs formed from the elements of *keys* and *values* prepended to the existing *alist*.

If an *alist* argument is omitted, it defaults to `nil`.

Pairs of elements are formed by taking successive elements from the *keys* and *values* sequences in parallel.

If the sequences are not of equal length, the excess elements from the longer sequence are ignored.

The pairs appear in the resulting list in the original order in which their constituents appeared in *keys* and *values*.

Dialect Note:

The ANSI CL `pairlis` requires *key* and *data* to be lists, not sequences. The behavior of the ANSI CL `pairlis` is undefined of those lists are of different lengths. Finally, the elements are permitted to appear in either the original order or reverse order.

Examples:

```
(pairlis nil nil) -> nil
(pairlis "abc" #(1 2 3 4)) -> ((#\a . 1) (#\b . 2) (#\c . 3))

(pairlis '(1 2 3) '(a b c) '((x . y) (z . w)))
-> ((1 . a) (2 . b) (3 . c) (x . y) (z . w))
```

9.12 Property Lists

A *property list*, also referred to as a *plist*, is a flat list of even length consisting of interleaved pairs of property names (usually symbols) and their values (arbitrary objects). An example property list is (:a 1 :b "two") which contains two properties, :a having value 1, and :b having value "two".

An *improper plist* represents Boolean properties in a condensed way, as property indicators which are not followed by a value. Such properties only indicate their presence or absence, which is useful for encoding a Boolean value. If it is absent, then the property is false. Correctly using an improper plist requires that the exact set of Boolean keys is established by convention.

In this document, the unqualified terms *property list* and *plist* refer strictly to an ordinary plist, not to an improper plist.

Dialect Note:

Unlike in some other Lisp dialects, including ANSI Common Lisp, symbols do not have property lists in **TXR Lisp**. Improper plists aren't a concept in ANSI CL.

9.12.1 Function `prop`

Syntax:

```
(prop plist key)
```

Description:

The `prop` function searches property list *plist* for key *key*. If the key is found, then the value next to it is returned. Otherwise `nil` is returned.

It is ambiguous whether `nil` is returned due to the property not being found, or due to the property being present with a `nil` value.

The indicators in *plist* are compared with *key* using eq equality, allowing them to be symbols, characters or fixnum integers.

9.12.2 Function memp

Syntax:

```
(memp key plist)
```

Description:

The memp function searches property list *plist* for key *key*, using eq equality.

If the key is found, then the entire suffix of *plist* beginning with the indicator is returned, such that the first element of the returned list is *key* and the second element is the property value.

Note the reversed argument convention relative to the prop function, harmonizing with functions in the member family.

9.12.3 Functions plist-to-alist and improper-plist-to-alist

Syntax:

```
(plist-to-alist plist)
(improper-plist-to-alist imp-plist bool-keys)
```

Description:

The functions plist-to-alist and improper-plist-to-alist convert, respectively, a property list and improper property list to an association list.

The plist-to-alist function scans *plist* and returns the indicator-property pairs as a list of cons cells, such that each car is the indicator, and each cdr is the value.

The improper-plist-to-alist is similar, except that it handles the Boolean properties which, by convention, aren't followed by a value. The list of all such indicators is specified by the bool-keys argument.

Examples:

```
(plist-to-alist '(a 1 b 2))  -->  ((a . 1) (b . 2))

(improper-plist-to-alist '(:x 1 :blue :y 2) '(:blue))
-->  ((:x . 1) (:blue) (:y . 2))
```

9.13 List Sorting

Note: these functions operate on lists. The principal sorting function in **TXR Lisp** is sort, described under Sequence Manipulation.

The merge function described here provides access to an elementary step of the algorithm used internally by sort when operating on lists.

The multi-sort operation sorts multiple lists in parallel. It is implemented using sort.

9.13.1 Function `merge`

Syntax:

```
(merge seq1 seq2 [lessfun [keyfun]])
```

Description:

The `merge` function merges two sorted sequences *seq1* and *seq2* into a single sorted sequence. The semantics and defaulting behavior of the *lessfun* and *keyfun* arguments are the same as those of the sort function.

The sequence which is returned is of the same kind as *seq1*.

This function is destructive of any inputs that are lists. If the output is a list, it is formed out of the structure of the input lists.

9.13.2 Function `multi-sort`

Syntax:

```
(multi-sort columns less-funcs [key-funcs])
```

Description:

The `multi-sort` function regards a list of lists to be the columns of a database. The corresponding elements from each list constitute a record. These records are to be sorted, producing a new list of lists.

The *columns* argument supplies the list of lists which comprise the columns of the database. The lists should ideally be of the same length. If the lists are of different lengths, then the shortest list is taken to be the length of the database. Excess elements in the longer lists are ignored, and do not appear in the sorted output.

The *less-funcs* argument supplies a list of comparison functions which are applied to the columns. Successive functions correspond to successive columns. If *less-funcs* is an empty list, then the sorted database will emerge in the original order. If *less-funcs* contains exactly one function, then the rows of the database are sorted according to the first column. The remaining columns simply follow their row. If *less-funcs* contains more than one function, then additional columns are taken into consideration if the items in the previous columns compare `equal`. For instance if two elements from column one compare `equal`, then the corresponding second column elements are compared using the second column comparison function. The *less-funcs* argument may be a function object, in which case it is treated as if it were a one-element list containing that function object.

The optional *key-funcs* argument supplies transformation functions through which column entries are converted to comparison keys, similarly to the single key function used in the sort function and others. If there are more key functions than less functions, the excess key functions are ignored.

9.14 Lazy Lists and Lazy Evaluation

9.14.1 Function `make-lazy-cons`

Syntax:

```
(make-lazy-cons function [car [cdr]])
```

Description:

The function `make-lazy-cons` makes a special kind of cons cell called a lazy cons, whose type is `lcons`. Lazy conses are useful for implementing lazy lists.

Lazy lists are lists which are not allocated all at once. Rather, the elements of its structure materialize just before they are accessed.

A lazy cons has `car` and `cdr` fields like a regular cons, and those fields are initialized to the values of the *car* and *cdr* arguments of `make-lazy-cons` when the lazy cons is created. These arguments default to *nil* if omitted. A lazy cons also has an update function, which is specified by the *function* argument to `make-lazy-cons`.

The *function* argument must be a function that may be called with exactly one parameter.

When either the `car` and `cdr` fields of a cons are accessed for the first time to retrieve their value, *function* is automatically invoked first, and is given the lazy cons as a parameter. That function has the opportunity to store new values into the `car` and `cdr` fields. Once the function is called, it is removed from the lazy cons: the lazy cons no longer has an update function. If the update function itself attempts to retrieve the value of the lazy cons cell's `car` or `cdr` field, it will be recursively invoked.

The functions `lcons-car` and `lcons-cdr` may be used to access the fields of a lazy cons without triggering the update function.

Storing a value into either the `car` or `cdr` field does not have the effect of invoking the update function.

If the function terminates by returning normally, the access to the value of the field then proceeds in the ordinary manner, retrieving whatever value has most recently been stored.

The return value of the function is ignored.

To perpetuate the growth of a lazy list, the function can make another call to `make-lazy-cons` and install the resulting cons as the `cdr` of the lazy cons.

Example:

```
;;; lazy list of integers between min and max
(defun integer-range (min max)
  (let ((counter min))
    ;; min is greater than max; just return empty list,
    ;; otherwise return a lazy list
    (if (> min max)
      nil
      (make-lazy-cons
        (lambda (lcons)
          ;; install next number into car
          (rplaca lcons counter)
          ;; now deal wit cdr field
          (cond
            ;; max reached, terminate list with nil!
            ((eql counter max)
             (rplacd lcons nil))
            ;; max not reached: increment counter
```

```
;; and extend with another lazy cons
(t
  (inc counter)
  (rplacd lcons
          (make-lazy-cons
            (lcons-fun lcons))))))))))
```

9.14.2 Function `lconsp`

Syntax:

```
(lconsp value)
```

Description:

The `lconsp` function returns `t` if *`value`* is a lazy cons cell. Otherwise it returns `nil`, even if *`value`* is an ordinary cons cell.

9.14.3 Function `lcons-fun`

Syntax:

```
(lcons-fun lazy-cons)
```

Description:

The `lcons-fun` function retrieves the update function of a lazy cons. Once a lazy cons has been accessed, it no longer has an update function and `lcons-fun` returns `nil`. While the update function of a lazy cons is executing, it is still accessible. This allows the update function to retrieve a reference to itself and propagate itself into another lazy cons (as in the example under `make-lazy-cons`).

9.14.4 Functions `lcons-car` and `lcons-cdr`

Syntax:

```
(lcons-car lazy-cons)
(lcons-cdr lazy-cons)
```

Description:

The functions `lcons-car` and `lcons-cdr` retrieve the `car` and `cdr` fields of *`lazy-cons`*, without triggering the invocation of its associated update function.

The *`lazy-cons`* argument must be an object of type `lcons`. Unlike the functions `car` and `cdr`, These functions cannot be applied to any other type of object.

Note: these functions may be used by the update function to retrieve the values which were stored into *`lazy-cons`* by the `make-lazy-cons` constructor, without triggering recursion. The function may then overwrite either or both of these values. This allows the fields of the lazy cons to store state information necessary for the propagation of a lazy list. If that state information consists of no more than two values, then no additional context object need be allocated.

9.14.5 Macro `lcons`

Syntax:

```
(lcons car-expression cdr-expression)
```

Description:

The `lcons` macro simplifies the construction of structures based on lazy conses. Syntactically, it resembles the `cons` function. However, the arguments are expressions rather than values. The macro generates code which, when evaluated, immediately produces a lazy cons. The expressions *car-expression* and *cdr-expression* are not immediately evaluated. Rather, when either the `car` or `cdr` field of the lazy cons cell is accessed, these expressions are both evaluated at that time, in the order that they appear in the `lcons` expression, and in the original lexical scope in which that expression was evaluated. The return values of these expressions are used, respectively, to initialize the corresponding fields of the lazy cons.

Note: the `lcons` macro may be understood in terms of the following reference implementation, as a syntactic sugar combining the `make-lazy-cons` constructor with a lexical closure provided by a `lambda` function:

```
(defmacro lcons (car-form cdr-form)
  (let ((lc (gensym)))
     ^(make-lazy-cons (lambda (,lc)
                        (rplaca ,lc ,car-form)
                        (rplacd ,lc ,cdr-form)))))
```

Example:

```
;; Given the following function ...

(defun fib-generator (a b)
  (lcons a (fib-generator b (+ a b))))

;; ... the following function call generates the Fibonacci
;; sequence as an infinite lazy list.

(fib-generator 1 1) -> (1 1 2 3 5 8 13 ...)
```

9.14.6 Functions `lazy-stream-cons` and `get-lines`

Syntax:

```
(lazy-stream-cons stream [no-throw-close-p])
(get-lines [stream [no-throw-close-p]])
```

Description:

The `lazy-stream-cons` and `get-lines` functions are synonyms, except that the *stream* argument is optional in `get-lines` and defaults to `*stdin*`. Thus, the following description of `lazy-stream-cons` also applies to `get-lines`.

The `lazy-stream-cons` returns a lazy cons which generates a lazy list based on reading lines of text from input stream *stream*, which form the elements of the list. The `get-line` function is called on demand to add elements to the list.

The `lazy-stream-cons` function itself makes the first call to `get-line` on the stream. If this returns `nil`, then the stream is closed and `nil` is returned. Otherwise, a lazy cons is returned whose update function will install that line into the `car` field of the lazy cons, and continue the lazy list by making another call to `lazy-stream-cons`, installing the result into the `cdr` field. When this lazy list obtains an end-of-file indication from the stream, it closes the stream.

`lazy-stream-cons` inspects the real-time property of a stream as if by the `real-time-`

stream-p function. This determines which of two styles of lazy list are returned. For an ordinary (non-real-time) stream, the lazy list treats the end-of-file condition accurately: an empty file turns into the empty list nil, a one line file into a one-element list which contains that line and so on. This accuracy requires one line of lookahead which is not acceptable in real-time streams, and so a different type of lazy list is used, which generates an extra nil item after the last line. Under this type of lazy list, an empty input stream translates to the list (nil); a one-line stream translates to ("line" nil) and so forth.

If and when *stream* is closed by the function directly, or else by the returned lazy list, the *no-throw-close-p* Boolean argument, defaulting to nil, controls the *throw-on-error-p* argument of the call to the close-stream function. These arguments have opposite polarity: if *no-throw-close-p* is true, then *throw-on-error-p* shall be false, and vice versa.

9.14.7 Macro close-lazy-streams

Syntax:

```
(close-lazy-streams body-form*)
```

Description:

The close-lazy-streams macro establishes a dynamic environment in which zero or more *body-form*s are evaluated, yielding the value of the last *body-form*, or else nil if there are no *body-form* arguments. In this regard, the macro operator resembles progn.

The environment established by close-lazy-streams sets up special monitoring of the the functions lazy-stream-cons and get-lines. Whenever these functions register an I/O stream with a lazy list, that stream is recorded in a hidden. When the close-lazy-streams form terminates, it invokes the close-stream on each stream in the hidden list.

Note: the close-lazy-streams macro provides a possible solution for situations in which a body of code, possibly consisting of nested functions, manipulates lazy lists of lines coming from from I/O streams, such that these lists are not completely forced. Incompletely processed lazy lists will not close their associated streams until they are reclaimed by garbage collection, which could cause the application to run out of file descriptors. The close-lazy-streams macro allows the application to delineate a dynamic contour of code upon whose termination all such stream associations generated within that contour will be duly cleaned up.

Example:

Collect list of names of .tl files which contain the string "(cons ":

```
;; Incorrect version: could run out of open files if there are many
;; files which contain a match processed, because find-if will stop
;; traversing the list of lines when it finds a match:
(build
  (each ((file (glob "*.tl")))
    (if (find-if #/\(cons / (file-get-lines file))
      (add file))))

;; Addressed with close-lazy-streams: after each iteration, the
;; stream created by file-get-lines is closed.
(build
  (each ((file (glob "*.tl")))
    (close-lazy-streams
      (if (find-if #/\(cons / (file-get-lines file))
```

```
                (add file)))))
```

9.14.8 Macro `delay`

Syntax:

```
(delay expression)
```

Description:

The delay operator arranges for the delayed (or "lazy") evaluation of *expression*. This means that the expression is not evaluated immediately. Rather, the delay expression produces a promise object.

The promise object can later be passed to the `force` function (described later in this document). The force function will trigger the evaluation of the expression and retrieve the value.

The expression is evaluated in the original scope, no matter where the `force` takes place.

The expression is evaluated at most once, by the first call to `force`. Additional calls to `force` only retrieve a cached value.

Example:

```
;; list is popped only once: the value is computed
;; just once when force is called on a given promise
;; for the first time.

(defun get-it (promise)
  (format t "*list* is ~s\n" *list*)
  (format t "item is ~s\n" (force promise))
  (format t "item is ~s\n" (force promise))
  (format t "*list* is ~s\n" *list*))

(defvar *list* '(1 2 3))

(get-it (delay (pop *list*)))

Output:

*list* is (1 2 3)
item is 1
item is 1
*list* is (2 3)
```

9.14.9 Accessor `force`

Syntax:

```
(force promise)
(set (force promise) new-value)
```

Description:

The `force` function accepts a promise object produced by the `delay` macro. The first time `force` is invoked, the *expression* which was wrapped inside *promise* by the `delay` macro is evaluated (in its original lexical environment, regardless of where in the program the `force` call takes place). The value of *expression* is cached inside *promise* and returned, becoming

the return value of the `force` function call. If the `force` function is invoked additional times on the same promise, the cached value is retrieved.

A `force` form is a syntactic place, denoting the value cache location within *promise*.

Storing a value in a `force` place causes future accesses to the *promise* to return that value.

If the promise had not yet been forced, then storing a value into it prevents that from ever happening. The delayed *expression* will never be evaluated.

If, while a promise is being forced, the evaluation of *expression* itself causes an assignment to the promise, it is not specified whether the promise will take on the value of *expression* or the assigned value.

9.14.10 Function `promisep`

Syntax:

```
(promisep object)
```

Description:

The `promisep` function returns `t` if *object* is a promise object: an object created by the `delay` macro. Otherwise it returns `nil`.

Note: promise objects are conses. The `typeof` function applied to a promise returns `cons`.

9.14.11 Macro `mlet`

Syntax:

```
(mlet ({sym | (sym init-form)}*) body-form*)
```

Description:

The `mlet` macro ("magic let" or "mutual let") implements a variable binding construct similar to `let` and `let*`.

Under `mlet`, the scope of the bindings of the *sym* variables extends over the *init-form*s, as well as the *body-form*s.

Unlike the `let*` construct, each *init-form* has each *sym* in scope. That is to say, an *init-form* can refer not only to previous variables, but also to later variables as well as to its own variable.

The variables are not initialized until their values are accessed for the first time. Any *sym* whose value is not accessed is not initialized.

Furthermore, the evaluation of each *init-form* does not take place until the time when its value is needed to initialize the associated *sym*. This evaluation takes place once. If a given *sym* is not accessed during the evaluation of the `mlet` construct, then its *init-form* is never evaluated.

The bound variables may be assigned. If, before initialization, a variable is updated in such a way that its prior value is not needed, it is unspecified whether initialization takes place, and thus whether its *init-form* is evaluated.

Direct circular references are erroneous and are diagnosed. This takes place when the macro-expanded form is evaluated, not during the expansion of `mlet`.

Examples:

```
;; Dependent calculations in arbitrary order
(mlet ((x (+ y 3))
       (z (+ x 1))
       (y 4))
  (+ z 4))  -->  12

;; Error: circular reference:
;; x depends on y, y on z, but z on x again.
(mlet ((x (+ y 1))
       (y (+ z 1))
       (z (+ x 1)))
  z)

;; Okay: lazy circular reference because lcons is used
(mlet ((list (lcons 1 list)))
  list)  -->  (1 1 1 1 1 ...) ;; circular list
```

In the last example, the `list` variable is accessed for the first time in the body of the `mlet` form. This causes the evaluation of the `lcons` form. This form evaluates its arguments lazily, which means that it is not a problem that `list` is not yet initialized. The form produces a lazy cons, which is then used to initialize `list`. When the `car` or `cdr` fields of the lazy cons are accessed, the `list` expression in the `lcons` argument is accessed. By that time, the variable is initialized and holds the lazy cons itself, which creates the circular reference, and a circular list.

9.14.12 Functions `generate`, `giterate` and `ginterate`

Syntax:

```
(generate while-fun gen-fun)
(giterate while-fun gen-fun [value])
(ginterate while-fun gen-fun [value])
```

Description:

The `generate` function produces a lazy list which dynamically produces items according to the following logic.

The arguments to `generate` are functions which do not take any arguments. The return value of generate is a lazy list.

When the lazy list is accessed, for instance with the functions car and cdr, it produces items on demand. Prior to producing each item, *`while-fun`* is called. If it returns a true Boolean value (any value other than `nil`), then the *`gen-fun`* function is called, and its return value is incorporated as the next item of the lazy list. But if *`while-fun`* yields `nil`, then the lazy list immediately terminates.

Prior to returning the lazy list, generate invokes the *`while-fun`* one time. If `while-fun` yields `nil`, then `generate` returns the empty list `nil` instead of a lazy list. Otherwise, it instantiates a lazy list, and invokes the `gen-fun` to populate it with the first item.

The `giterate` function is similar to `generate`, except that *`while-fun`* and *`gen-fun`* are functions of one argument rather than functions of no arguments. The optional *`value`* argument defaults to `nil` and is threaded through the function calls. That is to say, the lazy list returned is `(`*`value`* `[`*`gen-fun value`*`] [`*`gen-fun`* `[`*`gen-fun value`*`]] ...)`.

The lazy list terminates when a value fails to satisfy *while-fun*. That is to say, prior to generating each value, the lazy list tests the value using *while-fun*. If that function returns `nil`, then the item is not added, and the sequence terminates.

Note: `giterate` could be written in terms of `generate` like this:

```
(defun giterate (w g v)
   (generate (lambda () [w v])
             (lambda () (prog1 v (set v [g v])))))
```

The `ginterate` function is a variant of `giterate` which includes the test-failing item in the generated sequence. That is to say `ginterate` generates the next value and adds it to the lazy list. The value is then tested using *while-fun*. If that function returns `nil`, then the list is terminated, and no more items are produced.

Example:

```
(giterate (op > 5) (op + 1) 0) -> (0 1 2 3 4)
(ginterate (op > 5) (op + 1) 0) -> (0 1 2 3 4 5)
```

9.14.13 Function `expand-right`

Syntax:

```
(expand-right gen-fun value)
```

Description:

The `expand-right` function is a complement to `reduce-right`, with lazy semantics.

The *gen-fun* parameter is a function, which must accept a single argument, and return either a cons pair or `nil`.

The *value* parameter is any value.

The first call to *gen-fun* receives *value*.

The return value is interpreted as follows. If *gen-fun* returns a cons-cell pair `(elem . next)` then *elem* specifies the element to be added to the lazy list, and *next* specifies the value to be passed to the next call to *gen-fun*. If *gen-fun* returns `nil` then the lazy list ends.

Examples:

```
;; Count down from 5 to 1 using explicit lambda
;; for gen-fun:

(expand-right
  (lambda (item)
    (if (zerop item) nil
      (cons item (pred item))))
  5)
--> (5 4 3 2 1)

;; Using functional combinators:
[expand-right [iff zerop nilf [callf cons identity pred]] 5]
--> (5 4 3 2 1)
```

```
;; Include zero:
[expand-right
  [iff null
     nilf
     [callf cons identity [iff zerop nilf pred]]] 5]
--> (5 4 3 2 1 0)
```

9.14.14 Functions `expand-left` and `nexpand-left`

Syntax:

```
(expand-left gen-fun value)
(nexpand-left gen-fun value)
```

Description:

The `expand-left` function is a companion to `expand-right`.

Unlike `expand-right`, it has eager semantics: it calls `gen-fun` repeatedly and accumulates an output list, not returning until `gen-fun` returns `nil`.

The semantics is as follows. `expand-left` initializes an empty accumulation list. Then *gen-fun* is called, with *value* as its argument.

If *gen-fun* it returns a cons cell, then the `car` of that cons cell is pushed onto the accumulation list, and the procedure is repeated: *gen-fun* is called again, with `cdr` taking the place of *value*.

If *gen-fun* returns `nil`, then the accumulation list is returned.

If the expression `(expand-right f v)` produces a terminating list, then the following equivalence holds:

```
(expand-left f v) <--> (reverse (expand-right f v))
```

The equivalence cannot hold for arguments to `expand-left` which produce an infinite list.

The `nexpand-left` function is a destructive version of `expand-left`.

The list returned by `nexpand-left` is composed of the cons cells returned by `gen-fun` whereas the list returned by `expand-left` is composed of freshly allocated cons cells.

9.14.15 Function `repeat`

Syntax:

```
(repeat list [count])
```

Description:

If *list* is empty, then repeat returns an empty list.

If *count* is omitted, the `repeat` function produces an infinite lazy list formed by catenating together copies of *list*.

If *count* is specified and is zero or negative, then an empty list is returned.

Otherwise a list is returned consisting of *count* repetitions of *list* catenated together.

9.14.16 Function `pad`

Syntax:

```
(pad sequence [object [count]])
```

Description:

The `pad` function produces a lazy list which consists of all of the elements of *`sequence`* followed by repetitions of *`object`*.

If *`object`* is omitted, it defaults to `nil`.

If *`count`* is omitted, then the repetition of *`object`* is infinite. Otherwise the specified number of repetitions occur.

Note that *`sequence`* may be a lazy list which is infinite. In that case, the repetitions of *`object`* will never occur.

9.14.17 Function `weave`

Syntax:

```
(weave {sequence}*)
```

Description:

The `weave` function interleaves elements from the sequences given as arguments.

If called with no arguments, it returns the empty list.

If called with a single sequence, it returns the elements of that sequence as a new lazy list.

When called with two or more sequences, `weave` returns a lazy list which draws elements from the sequences in a round-robin fashion, repeatedly scanning the sequences from left to right, and taking an item from each one, removing it from the sequence. Whenever a sequence runs out of items, it is deleted; the weaving then continues with the remaining sequences. The weaved sequence terminates when all sequences are eliminated. (If at least one of the sequences is an infinite lazy list, then the weaved sequence is infinite.)

Examples:

```
;; Weave negative integers with positive ones:
(weave (range 1) (range -1 : -1)) -> (1 -1 2 -2 3 -3 ...)

(weave "abcd" (range 1 3) '(x x x x x x x))
--> (#\a 1 x #\b 2 x #\c 3 x #\d x x x x)
```

9.14.18 Macros `gen` and `gun`

Syntax:

```
(gen while-expression produce-item-expression)
(gun produce-item-expression)
```

Description:

The `gen` macro operator produces a lazy list, in a manner similar to the `generate` function. Whereas the `generate` function takes functional arguments, the `gen` operator takes two expressions, which is often more convenient.

The return value of gen is a lazy list. When the lazy list is accessed, for instance with the functions car and cdr, it produces items on demand. Prior to producing each item, the *while-expression* is evaluated, in its original lexical scope. If the expression yields a non-nil value, then *produce-item-expression* is evaluated, and its return value is incorporated as the next item of the lazy list. If the expression yields nil, then the lazy list immediately terminates.

The gen operator itself immediately evaluates *while-expression* before producing the lazy list. If the expression yields nil, then the operator returns the empty list nil. Otherwise, it instantiates the lazy list and invokes the *produce-item-expression* to force the first item.

The gun macro similarly creates a lazy list according to the following rules. Each successive item of the lazy list is obtained as a result of evaluating *produce-item-expression*. However, when *produce-item-expression* yields nil, then the list terminates (without adding that nil as an item).

Note 1: the form gun can be implemented as a macro-expanding to an instance of the gen operator, like this:

```
(defmacro gun (expr)
  (let ((var (gensym)))
    ^(let (,var)
       (gen (set ,var ,expr)
            ,var))))
```

This exploits the fact that the set operator returns the value that is assigned, so the set expression is tested as a condition by gen, while having the side effect of storing the next item temporarily in a hidden variable.

In turn, gen can be implemented as a macro expanding to some lambda functions which are passed to the generate function:

```
(defmacro gen (while-expr produce-expr)
  ^(generate (lambda () ,while-expr)
             (lambda () ,produce-expr)))
```

Note 2: gen can be considered as an acronym for Generate, testing Expression before Next item, whereas gun stands for Generate Until Null.

Example:

```
;; Make a lazy list of integers up to 1000
;; access and print the first three.
(let* ((counter 0)
       (list (gen (< counter 1000) (inc counter))))
  (format t "~s ~s ~s\n" (pop list) (pop list) (pop list)))

Output:
1 2 3
```

9.14.19 Functions range and range*

Syntax:

```
(range [from [to [step]]])
(range* [from [to [step]]])
```

Description:

The `range` and `range*` functions generate a lazy, potentially infinite list, according to several disciplines.

There is a major division in behavior depending on whether or not the `from` argument, which specifies the initial item, is an arithmetic type according to the `arithp` function. The following remarks describe the arithmetic case. A description of the non-arithmetic behavior follows.

The difference between and `range*` is that `range*` excludes the endpoint. For instance `(range 0 3)` generates the list `(0 1 2 3)`, whereas `(range* 0 3)` generates `(0 1 2)`.

All arguments are optional. If the *`step`* argument is omitted, then it defaults to 1 if the *`to`* argument is omitted, or else if it is greater than or equal to *`from`* according to the > function. If *`to`* is given, and is less than *`from`*, then a missing `step` argument defaults to -1.

Each value in the list is obtained from the previous by adding the *`step`* value. Positive or negative *`step`* values are allowed. There is no check for a step size of zero, or for a step direction which cannot meet the endpoint.

The *`step`* argument may be a function. The function must accept one argument. That argument is an element of the list, from which the function calculates the next element.

The *`to`* argument specifies the endpoint value, which, if it occurs in the list, is excluded from it by the `range*` function, but included by the range function. If *`to`* is missing, or specified as `nil`, then there is no endpoint, and the list which is generated is infinite, regardless of *`step`*.

If *`from`* is omitted, then the list begins at zero, otherwise *`from`* must be an arithmetic object which specifies the initial value.

The list stops if it reaches the endpoint value (which is included in the case of `range`, and excluded in the case of `range*`). However, depending on the arguments, it is possible that the generated list doesn't contain the endpoint value, yet steps over it. This occurs when the previous value of the list is less than the endpoint value, but the next value is greater, or vice versa. In this situation, the list also stops, and the excess value which surpasses the endpoint is excluded from the list.

The rest of the description applies to the case when the `from` argument is a non-arithmetic type.

In the non-arithmetic case, the *`step`* argument unconditionally defaults to 1. If it is given, it must either be a function, or else a positive integer.

If *`step`* is a function, that function is used to determine each successive value from the previous similarly to the arithmetic case. If the *`to`* value is omitted, an infinite list is generated this way. If the *`to`* argument is present, the list stops if it attains the endpoint value. No provision is made for the endpoint value being skipped, like in the arithmetic case. When the endpoint value is reached, `range*` function omits that value from the list.

If *`step`* is a positive integer, then range iteration is used. A range value is constructed from the *`from`* and *`to`* arguments as if by the `(rcons* from to)` expression. Here, the `to` argument defaults to `nil` if it is missing. An iterator is created for the resulting range object as if by `iter-begin` and this iterator is then used to obtain values for the lazy list returned by `range` or `range*`. The list ends when the iterator indicates that no more items are available. In the case of the `range*` function, the last value produced by the iterator is omitted from the list. The *`step`* size is used to skip items from the iterator. For instance, if the value is 3, then the sequence begins

with the *from* value. The next two values from the sequence are omitted, The fourth item from the sequence is included in the list, (unless there either is no such item, or the function is `range*`, and that item is the last one).

Examples:

```
(range 1 1) -> (1)
(range 0 4) -> (0 1 2 3 4)
(range 4 0) -> (4 3 2 1 0)
(range 0.0 2.0 0.5) (0.0 0.5 1.0 1.5 2.0)
(range #R(0 1) #R(3 4)) (#R(0 1) #R(1 2) #R(3 4))
(range 0 4 2) -> (0 2 4)
(range #\a #\e 2) (#\a #\c #\e)
(range 1 32 (op * 2)) -> (1 2 4 8 16 32))

(range* 1 1) -> nil
(range* 0 4) -> (0 1 2 3)

(range* 4 0 -2) -> (4 2)

(range 0 1.25 0.5) -> (0 0.5 1.0)
(range* 0 1.25 0.5) -> (0 0.5 1.0))

(range "A" "A") -> nil
(range "AA" "BC") -> ("AA" "AB" "AC" "BA" "BB" "BC")
(range "AA" "BC" 2) -> ("AA" "AC" "BB")

[range* "ABCD" nil rest] -> ("ABCD" "BCD" "CD" "D")
```

9.14.20 Functions `rlist` and `rlist*`

Syntax:

```
(rlist item*)
(rlist* item*)
```

Description:

The `rlist` ("range list") function is useful for producing a list consisting of a mixture of discontinuous numeric or character ranges and individual items.

The function returns a lazy list of elements. The items are produced by converting the function's successive *item* arguments into lists, which are lazily catenated together to form the output list.

Each *item* is transformed into a list as follows. Any item which is **not** a range object is trivially turned into a one-element list as if by the `(list item*)` expression.

Any item which is a range object, whose `to` field **isn't** a range is turned into a lazy list as if by evaluating the `(range (from item)(to item))` expression. Thus for instance the argument `1..10` turns into the (lazy) list `(1 2 3 4 5 6 7 8 9 10)`.

Any item which is a range object such that its `to` field is also a range is turned into a lazy list as if by evaluating the `(range (from item)(from (to item))(to (to item)))` expression. Thus for instance the argument expression `1..10..2` produces an *item* which `rlist` turns into the lazy list `(1 3 5 7 9)` as if by the call `(range 1 10 2)`. Note that the expression `1..10..2` stands for the expression `(rcons 1 (rcons 10 2))` which

evaluates to `#R(1 #R(10 2))`.

The `#R(1 #R(10 2))` range literal syntax can be passed as an argument to `rlist` with the same result as `1..10..2`.

The `rlist*` function differs from `rlist` in one regard: under `rlist*`, the ranges denoted by the range notation exclude the endpoint. That is, the ranges are generated as if by the `range*` function rather than `range`.

Note: it is permissible for *`item`* objects to specify infinite ranges. It is also permissible to apply `rlist` to an infinite argument list.

Examples:

```
(rlist 1 "two" :three)  ->  (1 "two" :three)
(rlist 10 15..16 #\a..#\d 2) -> (10 15 16 #\a #\b #\c #\d 2)
(take 7 (rlist 1 2 5..:)) -> (1 2 5 6 7 8 9)
```

9.15 Ranges

Ranges are objects that aggregate two values, not unlike `cons` cells. However, they are atoms, and are primarily intended to hold numeric or character values in their two fields. These fields are called `from` and `to` which are the names of the functions which access them. These fields are not mutable; a new value cannot be stored into either field of a range.

The printed notation for a range object consists of the prefix `#R` (hash R) followed by the two values expressed as a two-element list. Ranges can be constructed using the `rcons` function. The notation `x..y` corresponds to `(rcons x y)`.

Ranges behave as a numeric type and support a subset of the numeric operations. Two ranges can be added or subtracted, which obeys these equivalences:

```
(+ a..b c..d)  <-->  (+ a c)..(+ b d)
(- a..b c..d)  <-->  (- a c)..(- b d)
```

A range `a..b` can be combined with a character or number `n` using addition or subtractions, which obeys these equivalences:

```
(+ a..b n)  <-->  (+ n a..b)  <-->  (+ a n)..(+ b n)
(- a..b n)  <-->  (- a n)..(- b n)
(- n a..b)  <-->  (- n a)..(- n b)
```

A range can be multiplied by a number:

```
(* a..b n)  <-->  (* n a..b)  <-->  (* a n)..(* b n)
```

A range can be divided by a number using the `/` or `trunc` functions, but a number cannot be divided by a range:

```
(trunc a..b n)  <-->  (trunc a n)..(trunc b n)
(/ a..b n)      <-->  (/ a n)..(/ b n)
```

Ranges can be compared using the equality and inequality functions =, <, >, <= and >=. Equality obeys this equivalence:

```
(= a..b c..d)  <-->  (and (= a c) (= b d))
```

Inequality comparisons treat the `from` component with precedence over `to` such that only if the `from` components of the two ranges are not equal under the = function, then the inequality is based solely on them. If they are equal, then the inequality is based on the `to` components. This gives rise to the following equivalences:

```
(< a..b c..d)   <-->  (if (= a c) (< b d) (< a c))
(> a..b c..d)   <-->  (if (= a c) (> b d) (> a c))
(>= a..b c..d)  <-->  (if (= a c) (>= b d) (> a c))
(<= a..b c..d)  <-->  (if (= a c) (<= b d) (< a c))
```

Ranges can be negated with the one-argument form of the − function, which is equivalent to subtraction from zero: the negation distributes over the two range components.

The `abs` function also applies to ranges and distributes into their components.

The `succ` and `pred` family of functions also operate on ranges.

The length of a range may be obtained with the `length` function;

The length of the range `a..b` is defined as `(- b a)`, and may be obtained using the `length` function. The `empty` function accepts ranges and tests them for zero length.

9.15.1 Function `rcons`

Syntax:

(rcons *from to*)

Description:

The `rcons` function constructs a range object which holds the values *from* and *to*.

Though range objects are effectively binary cells like conses, they are atoms. They also aren't considered sequences, nor are they structures.

Range objects are used for indicating numeric ranges, such as substrings of lists, arrays and strings. The dotdot notation serves as a syntactic sugar for `rcons`. The syntax `a..b` denotes the expression `(rcons a b)`.

Note that ranges are immutable, meaning that it is not possible to replace the values in a range.

9.15.2 Function `rangep`

Syntax:

(rangep *value*)

Description:

The `rangep` function returns `t` if *value* is a range. Otherwise it returns `nil`.

9.15.3 Functions `from` and `to`

Syntax:

(from *range*)

```
(to range)
```

Description:

The `from` and `to` functions retrieve, respectively, the from and to fields of a range.

Note that these functions are not accessors, which is because ranges are immutable.

9.15.4 Functions `in-range` and `in-range*`

Syntax:

```
(in-range range value)
(in-range* range value)
```

Description:

The `in-range` and `in-range*` functions test whether the *value* argument lies in the range represented by the *range* argument, indicating the Boolean result using one of the values `t` or `nil`.

The *range* argument must be a range object.

It is expected that the range object's `from` value does not exceed the `to` value; a reversed range is considered empty.

The `in-range*` function differs from `in-range` in that it excludes the upper endpoint.

The implicit comparison against the range endpoints is performed using the `less` and `lequal` functions, as appropriate.

The following equivalences hold:

```
(in-range r x)  <-->  (and (lequal (from r) x)
                           (lequal x (to r)))

(in-range* r x)  <-->  (and (lequal (from r) x)
                            (less x (to r)))
```

9.16 Characters and Strings

9.16.1 Functions `mkstring` and `str`

Syntax:

```
(mkstring length [char])
(str length [char | string])
```

Description:

The `mkstring` function constructs a string object of a length specified by the *length* parameter. The *length* parameter must be non-negative. Every position in the string is initialized with *char*, which must be a character value.

If the optional argument *char* is not specified, it defaults to the space character.

The `str` function resembles `mkstring`, and behaves the same way when the second argument is omitted, and when it is a character value. The second argument of `str` may be a *string*, in which case the newly created string is filled by taking successive characters from *string*. If *string* is longer than *length*, its excess characters are ignored. If *string* is shorter, then

characters are taken from the beginning again; *string* is effectively taken as a fill pattern to be repeated as many times as necessary to provide the required number of characters. If *string* is empty, `str` fills the new string with spaces.

9.16.2 Function `copy-str`

Syntax:

```
(copy-str string)
```

Description:

The `copy-str` function constructs a new string whose contents are identical to *string*.

If *string* is a lazy string, then a lazy string is constructed with the same attributes as *string*. The new lazy string has its own copy of the prefix portion of *string* which has been forced so far. The unforced list and separator string are shared between *string* and the newly constructed lazy string.

9.16.3 Function `upcase-str`

Syntax:

```
(upcase-str string)
```

Description:

The `upcase-str` function produces a copy of *string* such that all lowercase characters of the English alphabet are mapped to their uppercase counterparts.

9.16.4 Function `downcase-str`

Syntax:

```
(downcase-str string)
```

Description:

The `downcase-str` function produces a copy of *string* such that all uppercase characters of the English alphabet are mapped to their lowercase counterparts.

9.16.5 Function `string-extend`

Syntax:

```
(string-extend string tail [final])
```

Description:

The `string-extend` function destructively increases the length of *string*, which must be an ordinary dynamic string. It is an error to invoke this function on a literal string or a lazy string.

The *tail* argument can be a character, string or integer. If it is a string or character, it specifies material which is to be added to the end of the string: either a single character or a sequence of characters. If it is an integer, it specifies the number of characters to be added to the string.

If *tail* is an integer, the newly added characters have indeterminate contents. The string appears to be the original one because of an internal terminating null character remains in place, but the characters beyond the terminating zero are indeterminate.

The optional Boolean argument *final*, defaulting to `nil`, is a hint which indicates whether this

`string-extend` call is expected to be the last time that the function is invoked on the given *`string`*. If *`final`* is true, then the *`string`* object's underlying memory allocation is trimmed to fit the actual string data. If the argument is false, the object may be given a larger allocation intended to improves the performance of subsequent `string-extend` calls.

9.16.6 Function `string-finish`

Syntax:

```
(string-finish string)
```

Description:

The `string-finish` function removes excess allocation from *`string`* that may have been produced by previous calls to `string-extend`.

Note: if the most recent call to string `string-extend` specified a true value for the *`final`* parameter, then calling `string-finish` is unnecessary and does nothing.

9.16.7 Function `stringp`

Syntax:

```
(stringp obj)
```

Description:

The `stringp` function returns `t` if *`obj`* is one of the several kinds of strings. Otherwise it returns `nil`.

9.16.8 Function `length-str`

Syntax:

```
(length-str string)
```

Description:

The `length-str` function returns the length *`string`* in characters. The argument must be a string.

9.16.9 Function `coded-length`

Syntax:

```
(coded-length string)
```

Description:

The `coded-length` function returns the number of bytes required to encode *`string`* in UTF-8.

The argument must be a character string.

If the string contains only characters in the ASCII range U+0001 to U+007F range, then the value returned shall be the same as that returned by the `length-str` function.

9.16.10 Function `search-str`

Syntax:

```
(search-str haystack needle [start [from-end]])
```

Description:

The `search-str` function finds an occurrence of the string *needle* inside the *haystack* string and returns its position. If no such occurrence exists, it returns `nil`.

If a *start* argument is not specified, it defaults to zero. If it is a nonnegative integer, it specifies the starting character position for the search. Negative values of *start* indicate positions from the end of the string, such that `-1` is the last character of the string.

If the *from-end* argument is specified and is not `nil`, it means that the search is conducted right-to-left. If multiple matches are possible, it will find the rightmost one rather than the leftmost one.

9.16.11 Function `search-str-tree`

Syntax:

```
(search-str-tree haystack tree [start [from-end]])
```

Description:

The `search-str-tree` function is similar to `search-str`, except that instead of searching *haystack* for the occurrence of a single needle string, it searches for the occurrence of numerous strings at the same time. These search strings are specified, via the *tree* argument, as an arbitrarily structured tree whose leaves are strings.

The function finds the earliest possible match, in the given search direction, from among all of the needle strings.

If *tree* is a single string, the semantics is equivalent to `search-str`.

9.16.12 Function `match-str`

Syntax:

```
(match-str bigstring littlestring [start])
```

Description:

Without the *start* argument, the `match-str` function determines whether *littlestring* is a prefix of *bigstring*.

If the *start* argument is specified, and is a nonnegative integer, then the function tests whether *littlestring* matches a prefix of that portion of *bigstring* which starts at the given position.

If the *start* argument is a negative integer, then `match-str` determines whether *littlestring* is a suffix of *bigstring*, ending on that position of bigstring, where `-1` denotes the last character of *bigstring*, `-2` the second last one and so on.

If *start* is `-1`, then this corresponds to testing whether *littlestring* is a suffix of *bigstring*.

The `match-str` function returns `nil` if there is no match.

If a prefix match is successful, then an integer value is returned indicating the position, inside *bigstring*, one character past the matching prefix. If the entire string is matched, then this value corresponds to the length of *bigstring*.

If a suffix match is successful, the return value is the position within *bigstring* where the left-most character of *littlestring* matched.

9.16.13 Function `match-str-tree`

Syntax:

```
(match-str-tree bigstring tree [start])
```

Description:

The `match-str-tree` function is a generalization of `match-str` which matches multiple test strings against *bigstring* at the same time. The value reported is the longest match from among any of the strings.

The strings are specified as an arbitrarily shaped tree structure which has strings at the leaves.

If *tree* is a single string atom, then the function behaves exactly like `match-str`.

9.16.14 Accessor `sub-str`

Syntax:

```
(sub-str str [from [to]])
(set (sub-str str [from [to]]) new-value)
```

Description:

The `sub-str` function has the same parameters and semantics as the `sub` function, function, except that the first argument is operated upon using string operations.

If a `sub-str` form is used as a place, it denotes a subrange of *list* as if it were a storage location. The previous value of this location, if needed, is fetched by a call to `sub-str`. Storing *new-value* to the place is performed by a call to `replace-str`. In an update operation which accesses the prior value and stores a new value, the arguments *str*, *from*, *to* and *new-value* are evaluated once.

The *str* argument is not itself required to be a place; it is not updated when a value is written to the `sub-str` storage location.

9.16.15 Function `replace-str`

Syntax:

```
(replace-str string item-sequence [from [to]])
```

Description:

The `replace-str` function has the same parameters and semantics as the `replace` function, except that the first argument is operated upon using string operations.

9.16.16 Functions `cat-str`, `join-with` and `join`

Syntax:

```
(cat-str item-seq [sep])
(join-with sep item*)
(join item*)
```

Description:

The `cat-str`, `join-with` and `join` functions combine items into a single string, which is returned.

Every *item* argument must be a character, string or else a possibly empty sequence of items. This rule applies recursively.

If a *sep* argument is present, it must be a character or string.

The *item-seq* argument must be a sequence of any mixture of items which are characters, strings or sequences of items. Note that this means that if *item-seq* is a character string, it is a valid argument, since it is a sequence of characters.

If *item-seq* is empty, or no *item* arguments are present, then all three functions return an empty string.

All three functions operate on an abstract sequence of character and string items, produced by a left-to-right recursive traversal of their *item-seq* or *item* arguments.

Under the `join-with` function, as well as the `cat-str` function a *sep* argument is given to it, the items are catenated together such that *sep* is interposed between them. If there are *n* character or string items, then *n - 1* copies of *sep* occur in the resulting string, which is returned.

Under the `join` function, or `cat-str` function invoked without a *sep* argument, the items are catenated together directly, without any separator. The resulting string is returned.

9.16.17 Function `split-str`

Syntax:

```
(split-str string sep [keep-between [count]])
```

Description:

The `split-str` function breaks the *string* into pieces, returning a list thereof. The *sep* argument must be one of three types: a string, a character or a regular expression. It determines the separator character sequences within *string*.

The following describes the behavior of `split-str` in the case when the integer parameter *count* is omitted. The semantics of *count* are then given.

All non-overlapping matches for *sep* within *string* are identified in left-to-right order, and are removed from *string*. The string is broken into pieces according to the gaps left behind by the removed separators, and a list of the remaining pieces is returned.

If *sep* is the empty string, then the separator pieces removed from the string are considered to be the empty strings between its characters. In this case, if *string* is of length one or zero, then it is considered to have no such pieces, and a list of one element is returned containing the original string. These remarks also apply to the situation when *sep* is a regular expression which matches only an empty substring of *string*.

If a match for *sep* is not found in the string at all (not even an empty match), then the string is not split at all: a list of one element is returned containing the original string.

If *sep* matches the entire string, then a list of two empty strings is returned, except in the case that the original string is empty, in which case a list of one element is returned, containing the empty

string.

Whenever two adjacent matches for *sep* occur, they are considered separate cuts with an empty piece between them.

This operation is nondestructive: *string* is not modified in any way.

If the optional *keep-between* argument is specified and is not `nil`, If an argument is given and is true, then *split-str* incorporates the matching separating pieces of *string* into the resulting list, such that if the resulting list is catenated, a string equivalent to the original string will be produced.

Note: to split a string into pieces of length one such that an empty string produces `nil` rather than `("")`, use the `(tok-str` *string* `#/./)` pattern.

Note: the function call `(split-str s r t)` produces a resulting list identical to `(tok-str s r t)`, for all values of `r` and `s`, provided that `r` does not match empty strings. If `r` matches empty strings, then the `tok-str` call returns extra elements compared to `split-str`, because `tok-str` allows empty matches to take place and extract empty tokens before the first character of the string, and after the last character, whereas `split-str` does not recognize empty separators at these outer limits of the string.

If the *count* parameter is present, it must be a non-negative integer. This value specifies the maximum number of pieces of the input *string* which are extracted by the splitting process. The returned list consists of these pieces, followed by the remainder of the string, if the remainder is nonempty. If *keep-sep* is true, then separators appear between the pieces, and if the remainder piece is present, the separator between the last piece and the remainder is included. If *count* is zero, then `split-str` returns a list of one element, which is *string*.

9.16.18 Functions `spl` and `spln`

Syntax:

```
(spl sep [keep-between] string)
(spln count sep [keep-between] string)
```

Description:

The `spl` function performs the same computation as `split-str`. The same-named parameters of `spl` and `split-str` have the same semantics. The difference is the argument order. The `spl` function takes the *sep* argument first. The last argument is always *string* whether or not there are two arguments or three. If there are three arguments, then *keep-between* is the middle one.

Note: the argument conventions of `spl` facilitate less verbose partial application, such as with macros in the `op` family, in the common situation when *string* is the unbound argument.

The `spln` function is similar to `spl`, taking a required argument *count*, which behaves exactly like the same-named argument of `spl-str`.

9.16.19 Functions `split-str-set` and `sspl`

Syntax:

```
(split-str-set string set)
(sspl set string)
```

Description:

The `split-str-set` function breaks the *string* into pieces, returning a list thereof. The *set* argument must be a string. It specifies a set of characters. All occurrences of any of these characters within *string* are identified, and are removed from *string*. The string is broken into pieces according to the gaps left behind by the removed separators.

Adjacent occurrences of characters from *set* within *string* are considered to be separate gaps which come between empty strings.

This operation is nondestructive: *string* is not modified in any way.

The `sspl` function performs the same operation; the only difference between `sspl` and `split-str-set` is argument order.

9.16.20 Functions `tok-str` and `tok-where`

Syntax:

```
(tok-str string regex [keep-between [count]])
(tok-where string regex)
```

Description:

The `tok-str` function searches *string* for tokens, which are defined as substrings of *string* which match the regular expression *regex* in the longest possible way, and do not overlap. These tokens are extracted from the string and returned as a list.

Whenever *regex* matches an empty string, then an empty token is returned, and the search for another token within *string* resumes after advancing by one character position. However, if an empty match occurs immediately after a nonempty token, that empty match is not turned into a token.

So for instance, `(tok-str "abc" #/a?/)` returns `("a" "" "")`. After the token `"a"` is extracted from a nonempty match for the regex, an empty match for the regex occurs just before the character `b`. This match is discarded because it is an empty match which immediately follows the nonempty match. The character `b` is skipped. The next match is an empty match between the `b` and `c` characters. This match causes an empty token to be extracted. The character `c` is skipped, and one more empty match occurs after that character and is extracted.

If the *keep-between* argument is true, then the behavior of `tok-str` changes in the following way. The pieces of *string* which are skipped by the search for tokens are included in the output. If no token is found in *string*, then a list of one element is returned, containing *string*. Generally, if N tokens are found, then the returned list consists of 2N + 1 elements. The first element of the list is the (possibly empty) substring which had to be skipped to find the first token. Then the token follows. The next element is the next skipped substring and so on. The last element is the substring of *string* between the last token and the end.

If *count* is specified, it must be a nonnegative integer. The value limits the number of tokens which are extracted. The returned list then includes one more item: the remainder of the string after the last extracted token. This item is omitted if the rest of the string is empty, unless *keep-between* is true.

The `tok-where` function works similarly to `tok-str`, but instead of returning the extracted tokens themselves, it returns a list of the character position ranges within *string* where matches for *regex* occur. The ranges are pairs of numbers, represented as cons cells, where the first number of the pair gives the starting character position, and the second number is one position past the

end of the match. If a match is empty, then the two numbers are equal.

The `tok-where` function does not support the *`keep-between`* parameter.

9.16.21 Function `tok`

Syntax:

```
(tok regex [keep-between] string)
(tokn count regex [keep-between] string)
```

Description:

The `tok` function performs the same computation as `tok-str`. The same-named parameters of `tok` and `tok-str` have the same semantics. The difference is the argument order. The `tok` function takes the *`regex`* argument first. The last argument is always *`string`* whether or not there are two arguments or three. If there are three arguments, then *`keep-between`* is the middle one.

Note: the argument conventions of `tok` facilitate less verbose partial application, such as with macros in the `op` family, in the common situation when *`string`* is the unbound argument.

The `tokn` function is similar to `tok`, taking a required argument *`count`*, which behaves exactly like the same-named argument of `tok-str`.

9.16.22 Function `list-str`

Syntax:

```
(list-str string)
```

Description:

The `list-str` function converts a string into a list of characters.

9.16.23 Function `trim-str`

Syntax:

```
(trim-str string)
```

Description:

The `trim-str` function produces a copy of *`string`* from which leading and trailing tabs, spaces and newlines are removed.

9.16.24 Functions `string-set-code` and `string-get-code`

Syntax:

```
(string-set-code string value)
(string-get-code string)
```

Description:

The `string-set-code` and `string-get-code` functions provide a mechanism for associating an integer code with a string.

Note: this mechanism is the basis for associating system error messages passed in exceptions with the `errno` values of the failed system library calls which precipitated these error exceptions.

Not all string types can have an integer code: lazy strings and literal strings do not have this capability. The *string* argument must be of type `str`.

The *value* argument must be an integer or character. It is recommended that its value be confined to the non-negative range of the platform's `int` C type. Otherwise it is unspecified whether the same value shall be observed by `string-get-code` as what was stored with `string-set-code`.

The `string-set-code` function associates the integer *value* with the given `string`, and returns `string`. Any previously associated value is overwritten.

The `string-get-code` function retrieves the value most recently associated with *string*. If *string* has no associated value, then `nil` is returned.

If the `string-extend` is invoked on a *string* then it is unspecified whether or not *string* has an associated value and, if so, what value that is, except in the following case: if `string-extend` is invoked with a *final* argument which is true, then *string* is caused not to have an associated value.

If the `string-finish` function is invoked on a *string*, that string is caused not to have an associated value.

9.16.25 Function `chrp`

Syntax:

```
(chrp obj)
```

Description:

Returns `t` if *obj* is a character, otherwise nil.

9.16.26 Function `chr-isalnum`

Syntax:

```
(chr-isalnum char)
```

Description:

Returns `t` if *char* is an alphanumeric character, otherwise nil. Alphanumeric means one of the uppercase or lowercase letters of the English alphabet found in ASCII, or an ASCII digit. This function is not affected by locale.

9.16.27 Function `chr-isalpha`

Syntax:

```
(chr-isalpha char)
```

Description:

Returns `t` if *char* is an alphabetic character, otherwise `nil`. Alphabetic means one of the uppercase or lowercase letters of the English alphabet found in ASCII. This function is not affected by locale.

9.16.28 Function `chr-isascii`

Syntax:

```
(chr-isascii char)
```

Description:

The `chr-isascii` function returns `t` if the code of character *`char`* is in the range 0 to 127 inclusive. For characters outside of this range, it returns `nil`.

9.16.29 Function `chr-iscntrl`

Syntax:

```
(chr-iscntrl char)
```

Description:

The `chr-iscntrl` function returns `t` if the character *`char`* is a control character. For all other character, it returns `nil`.

A control character is one which belongs to the Unicode C0 or C1 block. C0 consists of the characters U+0000 through U+001F, plus the character U+007F. These are the original ASCII control characters. Block C1 consists of U+0080 through U+009F.

9.16.30 Functions `chr-isdigit` and `chr-digit`

Syntax:

```
(chr-isdigit char)
(chr-digit char)
```

Description:

If *`char`* is an ASCII decimal digit character, `chr-isdigit` returns the value `t` and `chr-digit` returns the integer value corresponding to that digit character, a value in the range 0 to 9. Otherwise, both functions return `nil`.

9.16.31 Function `chr-isgraph`

Syntax:

```
(chr-isgraph char)
```

Description:

The `chr-isgraph` function returns `t` if *`char`* is a non-space printable ASCII character. It returns `nil` if it is a space or control character.

It also returns `nil` for non-ASCII characters: Unicode characters with a code above 127.

9.16.32 Function `chr-islower`

Syntax:

```
(chr-islower char)
```

Description:

The `chr-islower` function returns `t` if *`char`* is an ASCII lowercase letter. Otherwise it returns `nil`.

9.16.33 Function `chr-isprint`

Syntax:

```
(chr-isprint char)
```

Description:

The `chr-isprint` function returns `t` if *`char`* is an ASCII character which is not a control character. It also returns `nil` for all non-ASCII characters: Unicode characters with a code above 127.

9.16.34 Function `chr-ispunct`

Syntax:

```
(chr-ispunct char)
```

Description:

The `chr-ispunct` function returns `t` if *`char`* is an ASCII character which is not a control character. It also returns `nil` for all non-ASCII characters: Unicode characters with a code above 127.

9.16.35 Function `chr-isspace`

Syntax:

```
(chr-isspace char)
```

Description:

The `chr-isspace` function returns `t` if *`char`* is an ASCII whitespace character: any of the characters in the set `#\space`, `#\tab`, `#\linefeed`, `#\newline`, `#\return`, `#\vtab` and `#\page`. For all other characters, it returns `nil`.

9.16.36 Function `chr-isblank`

Syntax:

```
(chr-isblank char)
```

Description:

The `chr-isblank` function returns `t` if *`char`* is a space or tab: the character `#\space` or `#\tab`. For all other characters, it returns `nil`.

9.16.37 Function `chr-isunisp`

Syntax:

```
(chr-isunisp char)
```

Description:

The `chr-isunisp` function returns `t` if *`char`* is a Unicode whitespace character. This the case for all the characters for which `chr-isspace` returns `t`. It also returns `t` for these additional characters: `#\xa0`, `#\x1680`, `#\x180e`, `#\x2000`, `#\x2001`, `#\x2002`, `#\x2003`, `#\x2004`, `#\x2005`, `#\x2006`, `#\x2007`, `#\x2008`, `#\x2009`, `#\x200a`, `#\x2028`, `#\x2029`, `#\x205f`, and `#\x3000`. For all other characters, it returns `nil`.

9.16.38 Function `chr-isupper`

Syntax:

```
(chr-isupper char)
```

Description:

The `chr-isupper` function returns `t` if *`char`* is an ASCII uppercase letter. Otherwise it returns `nil`.

9.16.39 Functions `chr-isxdigit` and `chr-xdigit`

Syntax:

```
(chr-isxdigit char)
(chr-xdigit char)
```

Description:

If *`char`* is a hexadecimal digit character, `chr-isxdigit` returns the value `t` and `chr-xdigit` returns the integer value corresponding to that digit character, a value in the range 0 to 15. Otherwise, both functions returns `nil`.

A hexadecimal digit is one of the ASCII digit characters `0` through `9`, or else one of the letters `A` through `F` or their lowercase equivalents `a` through `f` denoting the values 10 to 15.

9.16.40 Function `chr-toupper`

Syntax:

```
(chr-toupper char)
```

Description:

If character *`char`* is a lowercase ASCII letter character, this function returns the uppercase equivalent character. If it is some other character, then it just returns *`char`*.

9.16.41 Function `chr-tolower`

Syntax:

```
(chr-tolower char)
```

Description:

If character *`char`* is an uppercase ASCII letter character, this function returns the lowercase equivalent character. If it is some other character, then it just returns *`char`*.

9.16.42 Functions `int-chr` and `chr-int`

Syntax:

```
(int-chr char)
(chr-int num)
```

Description:

The *`char`* argument must be a character. The `int-chr` function returns that character's Unicode code point value as an integer.

The *`num`* argument must be a fixnum integer in the range `0` to `#\x10FFFF`. The `chr-int` function interprets *`num`* as a Unicode code point value and returns the corresponding character object.

Note: these functions are also known by the obsolescent names `num-chr` and `chr-num`.

9.16.43 Accessor `chr-str`

Syntax:

```
(chr-str str idx)
(set (chr-str str idx) new-value)
```

Description:

The `chr-str` function performs random access on string *`str`* to retrieve the character whose position is given by integer *`idx`*, which must be within range of the string.

The index value 0 corresponds to the first (leftmost) character of the string and so nonnegative values up to one less than the length are possible.

Negative index values are also allowed, such that -1 corresponds to the last (rightmost) character of the string, and so negative values down to the additive inverse of the string length are possible.

An empty string cannot be indexed. A string of length one supports index 0 and index -1. A string of length two is indexed left to right by the values 0 and 1, and from right to left by -1 and -2.

If the element *`idx`* of string *`str`* exists, and the string is modifiable, then the `chr-str` form denotes a place.

A `chr-str` place supports deletion. When a deletion takes place, then the character at *`idx`* is removed from the string. Any characters after that position move by one position to close the gap, and the length of the string decreases by one.

Notes:

Direct use of `chr-str` is equivalent to the DWIM bracket notation except that `str` must be a string. The following relation holds:

```
(chr-str s i) --> [s i]
```

since `[s i] <--> (ref s i)`, this also holds:

```
(chr-str s i) --> (ref s i)
```

However, note the following difference. When the expression `[s i]` is used as a place, then the subexpression `s` must be a place. When `(chr-str s i)` is used as a place, `s` need not be a place.

9.16.44 Function `chr-str-set`

Syntax:

```
(chr-str-set str idx char)
```

Description:

The `chr-str` function performs random access on string *`str`* to overwrite the character whose position is given by integer *`idx`*, which must be within range of the string. The character at *`idx`* is overwritten with character *`char`*.

The *`idx`* argument works exactly as in `chr-str`.

The *`str`* argument must be a modifiable string.

Notes:

Direct use of `chr-str` is equivalent to the DWIM bracket notation provided that *`str`* is a string and *`idx`* an integer. The following relation holds:

```
(chr-str-set s i c) --> (set [s i] c)
```

Since `(set [s i] c) <--> (refset s i c)` for an integer index `i`, this also holds:

```
(chr-str s i) --> (refset s i c)
```

9.16.45 Function `span-str`

Syntax:

```
(span-str str set)
```

Description:

The `span-str` function determines the longest prefix of string *`str`* which consists only of the characters in string *`set`*, in any combination.

If both arguments are strings, the function returns an integer between 0 and the length of *`str`*.

Examples:

```
(span-str "abcde" "ab") -> 2
(span-str "abcde" "z") -> 0
(span-str "abcde" "") -> 0
(span-str "abcde" "edcba") -> 5
```

9.16.46 Function `compl-span-str`

Syntax:

```
(compl-span-str str set)
```

Description:

The `compl-span-str` function determines the longest prefix of string *`str`* which consists only of the characters which do not appear in *`set`*, in any combination.

If both arguments are strings, the function returns an integer between 0 and the length of *`str`*.

Examples:

```
(compl-span-str "abc,def" ",") -> 3
(compl-span-str "abc," ",") -> 3
(compl-span-str "abc" ",") -> 3
(compl-span-str "abc3" "0123456789") -> 3
(compl-span-str "3" "0123456789") -> 0
```

9.16.47 Function `break-str`

Syntax:

```
(break-str str set)
```

Description:

The `break-str` function returns an integer which represents the position of the first character in string *str* which appears in string *set*.

If there is no such character, then `nil` is returned.

Examples:

```
(break-str "abc,def.ghi" ",.:") -> 3
(break-str "abc,def.ghi" ".:") -> 6
(break-str "abc,def.ghi" ":") -> nil
```

9.17 Lazy Strings

Lazy strings are objects that were developed for the **TXR** pattern-matching language, and are exposed via **TXR Lisp**. Lazy strings behave much like strings, and can be substituted for strings. However, unlike regular strings, which exist in their entirety, first to last character, from the moment they are created, lazy strings do not exist all at once, but are created on demand. If character at index N of a lazy string is accessed, then characters 0 through N of that string are forced into existence. However, characters at indices beyond N need not necessarily exist.

A lazy string dynamically grows by acquiring new text from a list of strings which is attached to that lazy string object. When the lazy string is accessed beyond the end of its hitherto materialized prefix, it takes enough strings from the list in order to materialize the index. If the list doesn't have enough material, then the access fails, just like an access beyond the end of a regular string. A lazy string always takes whole strings from the attached list.

Lazy string growth is achieved via the `lazy-str-force-upto` function which forces a string to exist up to a given character position. This function is used internally to handle various situations.

The `lazy-str-force` function forces the entire string to materialize. If the string is connected to an infinite lazy list, this will exhaust all memory.

Lazy strings are specially recognized in many of the regular string functions, which do the right thing with lazy strings. For instance when `sub-str` is invoked on a lazy string, a special version of the `sub-str` logic is used which handles various lazy string cases, and can potentially return another lazy string. Taking a `sub-str` of a lazy string from a given character position to the end does not force the entire lazy string to exist, and in fact the operation will work on a lazy string that is infinite.

Furthermore, special lazy string functions are provided which allow programs to be written carefully to take better advantage of lazy strings. What carefully means is code that avoids unnecessarily forcing the lazy string. For instance, in many situations it is necessary to obtain the length of a string, only to test it for equality or inequality with some number. But it is not necessary to compute the length of a string in order to know that it is greater than some value.

9.17.1 Function `lazy-str`

Syntax:

```
(lazy-str string-list [terminator [limit-count]])
```

Description:

The `lazy-str` function constructs a lazy string which draws material from *string-list* which is a list of strings.

If the optional *terminator* argument is given, then it specifies a string which is appended to every string from *string-list*, before that string is incorporated into the lazy string. If *terminator* is not given, then it defaults to the string `"\n"`, and so the strings from *string-list* are effectively treated as lines which get terminated by newlines as they accumulate into the growing prefix of the lazy string. To avoid the use of a terminator string, a null string *terminator* argument must be explicitly passed. In that case, the lazy string grows simply by catenating elements from *string-list*.

If the *limit-count* argument is specified, it must be a positive integer. It expresses a maximum limit on how many elements will be consumed from *string-list* in order to feed the lazy string. Once that many elements are drawn, the string ends, even if the list has not been exhausted. However, that remaining list, though not contributing to the string, is still incorporated into the value returned by `lazy-str-get-trailing-list`.

9.17.2 Function `lazy-stringp`

Syntax:

```
(lazy-stringp obj)
```

Description:

The `lazy-stringp` function returns `t` if *obj* is a lazy string. Otherwise it returns `nil`.

9.17.3 Function `lazy-str-force-upto`

Syntax:

```
(lazy-str-force-upto lazy-str index)
```

Description:

The `lazy-str-force-upto` function tries to instantiate the lazy string such that the position given by *index* materializes. The *index* is a character position, exactly as used in the `chr-str` function.

It is an error if the *lazy-str* argument isn't a lazy string.

Some positions beyond *index* may also materialize, as a side effect, because the operation takes only whole strings from the internal list, according to the algorithm described below.

If the string is already materialized through to at least *index*, or if it is possible to materialize the string that far, then the value `t` is returned to indicate success.

If there is sufficient material to force the lazy string through to the *index* position, then `t` is returned, otherwise `nil`.

The *lazy-str* object's *limit-count* is observed: a total of no more than *limit-count* elements are taken from the object's list.

The algorithm is as follows:

1. While the length of the materialized prefix of the string is less than or equal to *index* and while elements are available in the list, subject to observance of the *limit-count*, perform the following steps 2 and 3:
2. Remove the next available string from the list, and add it as a suffix to the materialized prefix.

3. Add the *terminator* string to the materialized prefix.
4. Return `t` if the length of the materialized prefix exceeds *index*, otherwise `nil`.

The algorithm does not take portions of strings from the list, and always adds the terminator after incorporating each piece into the materialized prefix.

9.17.4 Function `lazy-str-force`

Syntax:

```
(lazy-str-force lazy-str)
```

Description:

The *lazy-str* argument must be a lazy string. The lazy string is forced to fully materialize.

The return value is an ordinary, non-lazy string equivalent to the fully materialized lazy string.

The *lazy-str* object's *limit-count* is observed: a total of no more than *limit-count* elements are taken from the object's list.

The algorithm that is followed by `lazy-str-force` is similar to the one followed by `lazy-str-force-upto`, with only the following modification. The test in step 1 isn't concerned with the length of the materialized prefix, since the goal is to materialize all available characters. Steps 2 and 3 are performed while elements are available in the list, subject to observance of the *limit-count*.

9.17.5 Function `lazy-str-get-trailing-list`

Syntax:

```
(lazy-str-get-trailing-list string index)
```

Description:

The `lazy-str-get-trailing-list` function can be considered, in some way, an inverse operation to the production of the lazy string from its associated list.

Note: the behavior of this function changed in **TXR** 274. This is subject to a note in the COMPATIBILITY section.

First, the lazy string *string* is forced up through the position *index*, as if by a call to *lazy-str-force-upto*.

If *string* consists of *index* or more characters, then after the forcing operation, it is guaranteed that at least *index* characters of the string have been materialized into a single string, called the *materialized prefix* of the lazy string. If fewer than *index* characters are available, taking into account the contribution of the terminator string, then the number of characters in the materialized prefix fall short of *index*. The materialized prefix never takes fractional strings from the lazy string's list, and is always terminated by the terminator string.

Next, the materialized prefix is split into pieces on occurrences of *string*'s terminator string, as if by using `spl` function. If the terminator string is empty, it is split into individual characters, in accordance with the semantics of that function.

Then, if the last piece of the split prefix is an empty string, it is removed. This situation occurs in two cases: the materialized prefix is empty, or else it ends in the terminating string. For example, if

the terminating string is a single newline, and the prefix is `"foo\n"`. In this case, `(spl "\n" "foo\n")` produces `("foo" "")` from which the trailing empty string is removed, leaving `("foo")`.

Finally, a list is formed by appending the split piece of the materialized prefix, calculated as described above, with *string*'s remaining list of strings which have not been pulled into the materialized prefix. This list is returned.

9.17.6 Functions `length-str->`, `length-str->=`, `length-str-<` and `length-str-<=`

Syntax:

```
(length-str-> string len)
(length-str->= string len)
(length-str-< string len)
(length-str-<= string len)
```

Description:

These functions compare the lengths of two strings. The following equivalences hold, as far as the resulting value is concerned:

```
(length-str-> s l) <--> (> (length-str s) l)
(length-str->= s l) <--> (>= (length-str s) l)
(length-str-< s l) <--> (< (length-str s) l)
(length-str-<= s l) <--> (<= (length-str s) l)
```

The difference between the functions and the equivalent forms is that if the string is lazy, the `length-str` function will fully force it in order to calculate and return its length.

These functions only force a string up to position *len*, so they are not only more efficient, but on infinitely long lazy strings they are usable.

`length-str` cannot compute the length of a lazy string with an unbounded length; it will exhaust all memory trying to force the string.

These functions can be used to test such as string whether it is longer or shorter than a given length, without forcing the string beyond that length.

9.17.7 Function `cmp-str`

Syntax:

```
(cmp-str left-string right-string)
```

Description:

The `cmp-str` function returns -1 if *left-string* is lexicographically prior to *right-string*. If the reverse relationship holds, it returns 1. Otherwise the strings are equal and zero is returned.

If either or both of the strings are lazy, then they are only forced to the minimum extent necessary for the function to reach a conclusion and return the appropriate value, since there is no need to look beyond the first character position in which they differ.

The lexicographic ordering is naive, based on the character code point values in Unicode taken as integers, without regard for locale-specific collation orders.

Note: in **TXR** 232 and earlier versions, `cmp-str` conforms to a weaker requirements: any negative integer value may be returned rather than -1, and any positive integer value can be returned instead of 1.

9.17.8 Functions `str=`, `str<`, `str>`, `str>=` and `str<=`

Syntax:

```
(str= left-string right-string)
(str< left-string right-string)
(str> left-string right-string)
(str<= left-string right-string)
(str>= left-string right-string)
```

Description:

These functions compare *left-string* and *right-string* lexicographically, as if by the `cmp-str` function.

The `str=` function returns `t` if the two strings are exactly the same, character for character, otherwise it returns `nil`.

The `str<` function returns `t` if *left-string* is lexicographically before *right-string*, otherwise nil.

The `str>` function returns `t` if *left-string* is lexicographically after *right-string*, otherwise `nil`.

The `str<` function returns `t` if *left-string* is lexicographically before *right-string*, or if they are exactly the same, otherwise `nil`.

The `str<` function returns `t` if *left-string* is lexicographically after *right-string*, or if they are exactly the same, otherwise `nil`.

9.17.9 Function `string-lt`

Syntax:

```
(string-lt left-str right-str)
```

Description:

The `string-lt` is a deprecated alias for `str<`.

9.18 Vectors

9.18.1 Function `vector`

Syntax:

```
(vector length [initval])
```

Description:

The `vector` function creates and returns a vector object of the specified length. The elements of the vector are initialized to *initval*, or to nil if *initval* is omitted.

9.18.2 Function `vec`

Syntax:

```
(vec arg*)
```

Description:

The `vec` function creates a vector out of its arguments.

9.18.3 Function `vectorp`

Syntax:

```
(vectorp obj)
```

Description:

The `vectorp` function returns `t` if *`obj`* is a vector, otherwise it returns `nil`.

9.18.4 Function `vec-set-length`

Syntax:

```
(vec-set-length vec len)
```

Description:

The `vec-set-length` modifies the length of *`vec`*, making it longer or shorter. If the vector is made longer, then the newly added elements are initialized to nil. The *`len`* argument must be non-negative.

The return value is *`vec`*.

9.18.5 Accessor `vecref`

Syntax:

```
(vecref vec idx)
(set (vecref vec idx) new-value)
```

Description:

The `vecref` function performs indexing into a vector. It retrieves an element of *`vec`* at position *`idx`*, counted from zero. The *`idx`* value must range from 0 to one less than the length of the vector. The specified element is returned.

If the element *`idx`* of vector *`vec`* exists, then the `vecref` form denotes a place.

A `vecref` place supports deletion. When a deletion takes place, then if *`idx`* denotes the last element in the vector, the vector's length is decreased by one, so that the vector no longer has that element. Otherwise, if *`idx`* isn't the last element, then each elements values at a higher index than *`idx`* shifts by one one element position to the adjacent lower index. Then, the length of the vector is decreased by one, so that the last element position disappears.

9.18.6 Function `vec-push`

Syntax:

```
(vec-push vec elem)
```

Description:

The `vec-push` function extends the length of a vector *`vec`* by one element, and sets the new element to the value *`elem`*.

The previous length of the vector (which is also the position of *elem*) is returned.

9.18.7 Function `length-vec`

Syntax:

```
(length-vec vec)
```

Description:

The `length-vec` function returns the length of vector *vec*. It performs similarly to the generic `length` function, except that the argument must be a vector.

9.18.8 Function `size-vec`

Syntax:

```
(size-vec vec)
```

Description:

The `size-vec` function returns the number of elements for which storage is reserved in the vector *vec*.

Notes:

The `length` of the vector can be extended up to this size without any memory allocation operations having to be performed.

9.18.9 Function `vec-list`

Syntax:

```
(vec-list list)
```

Description:

The `vec-list` function returns a vector which contains all of the same elements and in the same order as list *list*.

Note: this function is also known by the obsolescent name `vector-list`.

9.18.10 Function `list-vec`

Syntax:

```
(list-vec vec)
```

Description:

The `list-vec` function returns a list of the elements of vector *vec*.

Note: this function is also known by the obsolescent name `list-vector`.

9.18.11 Function `copy-vec`

Syntax:

```
(copy-vec vec)
```

Description:

The `copy-vec` function returns a new vector object of the same length as *vec* and containing

the same elements in the same order.

9.18.12 Accessor `sub-vec`

Syntax:

```
(sub-vec vec [from [to]])
(set (sub-vec vec [from [to]]) new-value)
```

Description:

The `sub-vec` function has the same parameters and semantics as the function `sub`, except that the *vec* argument must be a vector.

If a `sub-vec` form is used as a place, it denotes a subrange of *list* as if it were a storage location. The previous value of this location, if needed, is fetched by a call to `sub-vec`. Storing *new-value* to the place is performed by a call to `replace-vec`. In an update operation which accesses the prior value and stores a new value, the arguments *vec*, *from*, *to* and *new-value* are evaluated once.

The *vec* argument is not itself required to be a place; it is not updated when a value is written to the `sub-vec` storage location.

9.18.13 Function `replace-vec`

Syntax:

```
(replace-vec vec item-sequence [from [to]])
```

Description:

The `replace-vec` is like the `replace` function except that the *vec* argument must be a vector.

9.18.14 Function `fill-vec`

Syntax:

```
(fill-vec vec elem [from [to]])
```

Description:

The `fill-vec` function overwrites a range of the vector with copies of the *elem* value.

The *from* and *to* index arguments follow the same range indexing conventions as the *replace* and *sub* functions. If *from* is omitted, it defaults to zero. If *to* is omitted, it defaults to the length of *vec*. Negative values of *from* and *to* are adjusted by adding the length of the vector to them, once.

If the adjusted value of either *from* or *to* is negative, or exceeds the length of *vec*, an error exception is thrown.

The adjusted values of *to* and *from* specify a range of vec starting at the *from* index, and ending at the *to* index, which is excluded from the range.

If the adjusted *to* is less than or equal to the adjusted *from*, then *vec* is unaltered.

Otherwise, copies of element are stored into *vec* starting at the *from* index, ending just before the *to* index is reached.

The `fill-vec` function returns *vec*.

Examples:

```
(defvarl v (vec 1 2 3))

v --> #(1 2 3)

(fill-vec v 0) --> #(0 0 0)

(fill-vec v 3 1) --> #(0 3 3)

(fill-vec v 4 -1) --> #(0 3 4)

(fill-vec v 5 -3 -1) --> #(5 5 4)
```

9.18.15 Function `cat-vec`

Syntax:

```
(cat-vec vec-list)
```

Description:

The *vec-list* argument is a list of vectors. The `cat-vec` function produces a catenation of the vectors listed in *vec-list*. It returns a single large vector formed by catenating those vectors together in order.

9.19 Buffers

9.19.1 The `buf` type

Object of the type `buf` are *buffers*: vector-like objects specialized for holding binary data represented as a sequence of 8-bit bytes. Buffers support operations specialized toward the encoding of Lisp values into machine-oriented data types, and decoding such data types into Lisp values.

Buffers are particularly useful in conjunction with the Foreign Function Interface (FFI), since they can be used to prepare arbitrary data which can be passed into and out of a function by pointer. They are also useful for binary I/O.

9.19.2 Conventions Used by the `buf-put-` Functions

Buffers support a number of similar functions for converting Lisp numeric values into common data types, which are placed into the buffer. These functions are named starting with the `buf-put-` prefix, followed by an abbreviated type name.

Each of these functions takes three arguments: *buf* specifies the buffer, *pos* specifies the byte offset position into the buffer which receives the low-order byte of the data transfer, and *val* indicates the value.

If *pos* has a value such that any portion of the data transfer would like outside of the buffer, the buffer is automatically extended in length to contain the data transfer. If this extension causes any padding bytes to appear between the previous length of the buffer and *pos*, those bytes are initialized to zero.

The argument *val* giving the value to be stored must be an integer or character, except in the case of the types *float* and *double*(the functions `buf-put-float` and `buf-put-double`) for which it is required to be of type `float`, and in case of the function `buf-put-cptr` which expects the *val* argument to be a `cptr` object.

The *val* argument must be in range for the data type, or an exception results.

Unless otherwise indicated, the stored datum is in the local format used by the machine with regard to byte order and other representational details.

9.19.3 Conventions Used by the `buf-get-` Functions

Buffers support a number of similar functions for extracting common data types, and converting them into Lisp values. These functions are named starting with the `buf-get-` prefix, followed by an abbreviated type name.

Each of these functions takes two arguments: *buf* specifies the buffer and *pos* specifies the byte offset position into the buffer which holds the low-order byte of the datum to be extracted.

If any portion of requested datum lies outside of the boundaries of the buffer, an error exception is thrown.

The extracted value is converted to a Lisp datum. For the majority of these functions, the returned value is of type integer. The `buf-get-float` and `buf-get-double` return a floating-point value. The `buf-get-cptr` function returns a value of type `cptr`.

9.19.4 Function `make-buf`

Syntax:

```
(make-buf len [init-val [alloc-size]])
```

Description:

The `make-buf` function creates a new buffer object which holds *len* bytes. This argument may be zero.

If *init-val* is present, it specifies the value with which the first *len* bytes of the buffer are initialized. If omitted, it defaults to zero. The value of *init-val* must lie in the range 0 to 255.

The *alloc-size* parameter indicates how much memory to actually allocate for the buffer. If an argument is not given, the parameter takes on the same value as *len*. If an argument is given, its value must not be less than *len*.

9.19.5 Function `bufp`

Syntax:

```
(bufp object)
```

Description:

The `bufp` function returns `t` if *object* is a `buf`, otherwise it returns `nil`.

9.19.6 Function `length-buf`

Syntax:

```
(length-buf buf)
```

Description:

The `length-buf` function retrieves the buffer length: how many bytes are stored in the buffer.

Note: the generic `length` function is also applicable to buffers.

9.19.7 Function `buf-alloc-size`

Syntax:

```
(buf-alloc-size buf)
```

Description:

The `buf-alloc-size` function retrieves the allocation size of the buffer.

9.19.8 Function `buf-trim`

Syntax:

```
(buf-trim buf)
```

Description:

The `buf-trim` function reduces the amount of memory allocated to the buffer to the minimum required to hold it contents, effectively setting the allocation size to the current length.

The previous allocation size is returned.

9.19.9 Function `buf-set-length`

Syntax:

```
(buf-set-length buf len [init-val])
```

Description:

The `buf-set-length` function changes the length of the buffer. If the buffer is made longer, the newly added bytes appear at the end, and are initialized to the value given by *`init-val`*. If *`init-val`* is specified, its value must be in the range 0 to 255. It defaults to zero.

9.19.10 Function `copy-buf`

Syntax:

```
(copy-buf buf)
```

Description:

The `copy-buf` function returns a duplicate of *`buf`*: an object distinct from *`buf`* which has the same length and contents, and compares `equal` to *`buf`*.

9.19.11 Accessor `sub-buf`

Syntax:

```
(sub-buf buf [from [to]])
(set (sub-buf buf [from [to]]) new-val)
```

Description:

The `sub-buf` function has the same semantics as the `sub` function, except that the first argument must be a buffer.

The extracted sub-range of a buffer is itself a buffer object.

If `sub-buf` is used as a syntactic place, the argument expressions *`buf`*, *`from`*, *`to`* and *`new-val`* are evaluated just once. The prior value, if required, is accessed by calling `sub-buf` and *`new-val`* is then stored via `replace-buf`.

9.19.12 Function `replace-buf`

Syntax:

```
(replace-buf buf item-sequence [from [to]])
```

Description:

The `replace-buf` function has the same semantics as the `replace` function, except that the first argument must be a buffer.

The elements of `item-sequence` are stored into *`buf`* as if using the `buf-put-u8` function and therefore must be suitable *`val`* arguments for that function.

The of the arguments, semantics and return value given for `replace` apply to `replace-buf`.

9.19.13 Function `buf-list`

Syntax:

```
(buf-list list)
```

Description:

The `buf-list` function creates and returns a new buffer, whose contents are derived from the elements of *`list`*, which may be any kind of sequence.

The elements of *`list`* must be integers whose values lie in the range 0 to 255, or else characters whose code point values lie in that range. These values are placed into the newly created buffer, which therefore has the same length as *`list`*.

9.19.14 Function `buf-put-buf`

Syntax:

```
(buf-put-buf dst-buf pos src-buf)
```

Description:

The `buf-put-buf` function stores a copy of buffer *`src-buf`* into *`dst-buf`* at the offset indicated by *`pos`*.

The source and destination memory regions may overlap.

The return value is *`src-buf`*.

Note: the effect of a `buf-put-buf` operation may also be performed by a suitable call to `replace-buf`; however, `buf-put-buf` is less general: it doesn't insert or delete by replacing destination ranges with data of differing length, and requires a source operand of buffer type.

9.19.15 Function `buf-put-i8`

Syntax:

```
(buf-put-i8 buf pos val)
```

Description:

The `buf-put-i8` converts *`val`* into an 8-bit signed integer, and stores it into the buffer at the offset indicated by *`pos`*.

The return value is *`val`*.

9.19.16 Function `buf-put-u8`

Syntax:

```
(buf-put-u8 buf pos val)
```

Description:

The `buf-put-u8` converts *val* into an 8-bit unsigned integer, and stores it into the buffer at the offset indicated by *pos*.

The return value is *val*.

9.19.17 Function `buf-put-i16`

Syntax:

```
(buf-put-i16 buf pos val)
```

Description:

The `buf-put-i16` converts *val* into a sixteen bit signed integer, and stores it into the buffer at the offset indicated by *pos*.

The return value is *val*.

9.19.18 Function `buf-put-u16`

Syntax:

```
(buf-put-u16 buf pos val)
```

Description:

The `buf-put-u16` converts *val* into a sixteen bit unsigned integer, and stores it into the buffer at the offset indicated by *pos*.

The return value is *val*.

9.19.19 Function `buf-put-i32`

Syntax:

```
(buf-put-i32 buf pos val)
```

Description:

The `buf-put-i32` converts *val* into a 32-bit signed integer, and stores it into the buffer at the offset indicated by *pos*.

The return value is *val*.

9.19.20 Function `buf-put-u32`

Syntax:

```
(buf-put-u32 buf pos val)
```

Description:

The `buf-put-u32` converts *val* into a 32-bit unsigned integer, and stores it into the buffer at the offset indicated by *pos*.

The return value is *val*.

9.19.21 Function `buf-put-i64`

Syntax:

```
(buf-put-i64 buf pos val)
```

Description:

The `buf-put-i64` converts *val* into a 64-bit signed integer, and stores it into the buffer at the offset indicated by *pos*.

The return value is *val*.

9.19.22 Function `buf-put-u64`

Syntax:

```
(buf-put-u64 buf pos val)
```

Description:

The `buf-put-u64` converts the value *val* into a 64-bit unsigned integer, and stores it into the buffer at the offset indicated by *pos*.

The return value is *val*.

9.19.23 Function `buf-put-char`

Syntax:

```
(buf-put-char buf pos val)
```

Description:

The `buf-put-char` converts *val* into a value of the C type `char` and stores it into the buffer at the offset indicated by *pos*.

The return value is *val*.

Note that the `char` type may be signed or unsigned.

9.19.24 Function `buf-put-uchar`

Syntax:

```
(buf-put-uchar buf pos val)
```

Description:

The `buf-put-uchar` converts *val* into a value of the C type `unsigned char` and stores it into the buffer at the offset indicated by *pos*.

9.19.25 Function `buf-put-short`

Syntax:

```
(buf-put-short buf pos val)
```

Description:

The `buf-put-short` converts *val* into a value of the C type `short` and stores it into the buffer at the offset indicated by *pos*.

9.19.26 Function `buf-put-ushort`

Syntax:

```
(buf-put-ushort buf pos val)
```

Description:

The `buf-put-ushort` converts *val* into a value of the C type `unsigned short` and stores it into the buffer at the offset indicated by *pos*.

9.19.27 Function `buf-put-int`

Syntax:

```
(buf-put-int buf pos val)
```

Description:

The `buf-put-int` converts *val* into a value of the C type `int` and stores it into the buffer at the offset indicated by *pos*.

9.19.28 Function `buf-put-uint`

Syntax:

```
(buf-put-uint buf pos val)
```

Description:

The `buf-put-uint` converts *val* into a value of the C type `unsigned int` and stores it into the buffer at the offset indicated by *pos*.

9.19.29 Function `buf-put-long`

Syntax:

```
(buf-put-long buf pos val)
```

Description:

The `buf-put-long` converts *val* into a value of the C type `long` and stores it into the buffer at the offset indicated by *pos*.

9.19.30 Function `buf-put-ulong`

Syntax:

```
(buf-put-ulong buf pos val)
```

Description:

The `buf-put-ulong` converts *val* into a value of the C type `unsigned long` and stores it into the buffer at the offset indicated by *pos*.

9.19.31 Function `buf-put-float`

Syntax:

```
(buf-put-float buf pos val)
```

Description:

The `buf-put-float` converts *val* into a value of the C type `float` and stores it into the buffer at the offset indicated by *pos*.

Note: the conversion of a **TXR Lisp** floating-point value to the C type float may be inexact, reducing the numeric precision.

9.19.32 Function `buf-put-double`

Syntax:

```
(buf-put-double buf pos val)
```

Description:

The `buf-put-double` converts *val* into a value of the C type `double` and stores it into the buffer at the offset indicated by *pos*.

9.19.33 Function `buf-put-cptr`

Syntax:

```
(buf-put-cptr buf pos val)
```

Description:

The `buf-put-cptr` expects *val* to be of type `cptr`. It stores the object's pointer value into the buffer at the offset indicated by *pos*.

9.19.34 Function `buf-get-i8`

Syntax:

```
(buf-get-i8 buf pos)
```

Description:

The `buf-get-i8` function extracts and returns signed 8-bit integer from *buf* at the offset given by *pos*.

9.19.35 Function `buf-get-u8`

Syntax:

```
(buf-get-u8 buf pos)
```

Description:

The `buf-get-u8` function extracts and returns an unsigned 8-bit integer from *buf* at the offset given by *pos*.

9.19.36 Function `buf-get-i16`

Syntax:

```
(buf-get-i16 buf pos)
```

Description:

The `buf-get-i16` function extracts and returns a signed 16-bit integer from *buf* at the offset given by *pos*.

9.19.37 Function `buf-get-u16`

Syntax:

```
(buf-get-u16 buf pos)
```

Description:

The `buf-get-u16` function extracts and returns an unsigned 16-bit integer from *buf* at the offset given by *pos*.

9.19.38 Function `buf-get-i32`

Syntax:

```
(buf-get-i32 buf pos)
```

Description:

The `buf-get-i32` function extracts and returns a signed 32-bit integer from *buf* at the offset given by *pos*.

9.19.39 Function `buf-get-u32`

Syntax:

```
(buf-get-u32 buf pos)
```

Description:

The `buf-get-u32` function extracts and returns an unsigned 32-bit integer from *buf* at the offset given by *pos*.

9.19.40 Function `buf-get-i64`

Syntax:

```
(buf-get-i64 buf pos)
```

Description:

The `buf-get-i64` function extracts and returns a signed 64-bit integer from *buf* at the offset given by *pos*.

9.19.41 Function `buf-get-u64`

Syntax:

```
(buf-get-u64 buf pos)
```

Description:

The `buf-get-u64` function extracts and returns an unsigned 64-bit integer from *buf* at the offset given by *pos*.

9.19.42 Function `buf-get-char`

Syntax:

```
(buf-get-char buf pos)
```

Description:

The `buf-get-char` function extracts and returns a value of the C type `char` from *buf* at the offset given by *pos*. Note that `char` may be signed or unsigned.

9.19.43 Function `buf-get-uchar`

Syntax:

```
(buf-get-uchar buf pos)
```

Description:

The `buf-get-uchar` function extracts and returns a value of the C type `unsigned char` from *`buf`* at the offset given by *`pos`*.

9.19.44 Function `buf-get-short`

Syntax:

```
(buf-get-short buf pos)
```

Description:

The `buf-get-short` function extracts and returns a value of the C type `short` from *`buf`* at the offset given by *`pos`*.

9.19.45 Function `buf-get-ushort`

Syntax:

```
(buf-get-ushort buf pos)
```

Description:

The `buf-get-ushort` function extracts and returns a value of the C type `unsigned short` from *`buf`* at the offset given by *`pos`*.

9.19.46 Function `buf-get-int`

Syntax:

```
(buf-get-int buf pos)
```

Description:

The `buf-get-int` function extracts and returns a value of the C type `int` from *`buf`* at the offset given by *`pos`*.

9.19.47 Function `buf-get-uint`

Syntax:

```
(buf-get-uint buf pos)
```

Description:

The `buf-get-uint` function extracts and returns a value of the C type `unsigned int` from *`buf`* at the offset given by *`pos`*.

9.19.48 Function `buf-get-long`

Syntax:

```
(buf-get-long buf pos)
```

Description:

The `buf-get-long` function extracts and returns a value of the C type `long` from *`buf`* at the offset given by *`pos`*.

9.19.49 Function `buf-get-ulong`

Syntax:

```
(buf-get-ulong buf pos)
```

Description:

The `buf-get-ulong` function extracts and returns a value of the C type `unsigned long` from *`buf`* at the offset given by *`pos`*.

9.19.50 Function `buf-get-float`

Syntax:

```
(buf-get-float buf pos)
```

Description:

The `buf-get-float` function extracts and returns a value of the C type `float` from *`buf`* at the offset given by *`pos`*, returning that value as a Lisp floating-point number.

9.19.51 Function `buf-get-double`

Syntax:

```
(buf-get-double buf pos)
```

Description:

The `buf-get-double` function extracts and returns a value of the C type `double` from *`buf`* at the offset given by *`pos`*, returning that value as a Lisp floating-point number.

9.19.52 Function `buf-get-cptr`

Syntax:

```
(buf-get-cptr buf pos)
```

Description:

The `buf-get-cptr` function extracts a C pointer from *`buf`* at the offset given by *`pos`*, returning that value as a Lisp object of type `cnum`.

9.19.53 Function `put-buf`

Syntax:

```
(put-buf buf [pos [stream]])
```

Description:

The `put-buf` function writes the contents of buffer *`buf`*, starting at position *`pos`* to a stream, through to the last byte, if possible. Successive bytes from the buffer are written to the stream as if by a `put-byte` operation.

If *`stream`* is omitted, it defaults to `*stdout*`.

If *`pos`* is omitted, it defaults to zero. It indicates the starting position within the buffer.

The stream must support the `put-byte` operation. Streams which support `put-byte` can be expected to support `put-buf` and, conversely, streams which do not support `put-byte` do not support `put-buf`.

The `put-buf` function returns the position of the last byte that was successfully written. If the buffer was written through to the end, then this value corresponds to the length of the buffer.

If an error occurs before any bytes are written, the function throws an error.

9.19.54 Functions `fill-buf` and `fill-buf-adjust`

Syntax:

```
(fill-buf buf [pos [stream]])
(fill-buf-adjust buf [pos [stream]])
```

Description:

The `fill-buf` reads bytes from *`stream`* and writes them into consecutive locations in buffer *`buf`* starting at position *`pos`*. The bytes are read as if using the `get-byte` function.

If the *`stream`* argument is omitted, it defaults to `*stdin*`.

If *`pos`* is omitted, it defaults to zero. It indicates the starting position within the buffer.

The stream must support the `get-byte` operation. Buffers which support `get-byte` can be expected to support `fill-buf` and, conversely, streams which do not support `get-byte` do not support `fill-buf`.

The `fill-buf` function returns the position that is one byte past the last byte that was successfully read. If an end-of-file or other error condition occurs before the buffer is filled through to the end, then the value returned is smaller than the buffer length. In this case, the area of the buffer beyond the read size retains its previous content.

If an error situation occurs other than a premature end-of-file before any bytes are read, then an exception is thrown.

If an end-of-file condition occurs before any bytes are read, then zero is returned.

The `fill-buf-adjust` differs usefully from `fill-buf` as follows. Whereas `fill-buf` doesn't manipulate the length of the buffer at any stage of the operation, the `fill-buf-adjust` begins by adjusting the length of the buffer to the underlying allocated size. Then it performs the fill operation in exactly the same manner as `fill-buf`. Finally, if the operation succeeds, then `fill-buf-adjust` adjusts the length of the buffer to match the position that is returned.

9.19.55 Function `get-line-as-buf`

Syntax:

```
(get-line-as-buf [stream])
```

Description:

The `get-line-as-buf` reads bytes from *`stream`* as if using the `get-byte` function, until either a the newline character is encountered, or else the end of input is encountered. The bytes which are read, exclusive of the newline character, are returned in a new buffer object. The newline character, if it occurs, is consumed.

If *`stream`* is omitted, it defaults to `*stdin*`.

The stream is required to support byte input.

9.19.56 Functions `file-get-buf` and `command-get-buf`

Syntax:

```
(file-get-buf name [max-bytes
                        [skip-bytes [mode-opts]]])
(command-get-buf cmd [max-byte> [skip-bytes]])
```

Description:

The `file-get-buf` function opens a binary stream over the file indicated by the string argument *name* for reading. By default, the entire file is read and its contents are returned as a buffer object. The buffer's length corresponds to the number of bytes read from the file.

The `command-get-buf` function opens a binary stream over an input command pipe created for the command string *cmd*, as if by the `open-command` function. It read bytes from the pipe until the indication that no more input is available. The bytes are returned aggregated into a buffer object.

If the *max-bytes* parameter is given an argument, it must be a nonnegative integer. That value specifies a limit on the number of bytes to read. A buffer no longer than *max-bytes* shall be returned.

If the *skip-bytes* parameter is given an argument, it must be a nonnegative integer. That value specifies how many initial bytes of the input should be discarded before accumulation of the buffer begins. If possible, the semantics of this parameter is achieved by performing a `seek-stream` operation, falling back on reading and discarding bytes if the stream doesn't support seeking.

If *max-bytes* is specified, then the stream is opened in unbuffered mode, so that bytes beyond the specified range shall not be requested from the underlying file, device or process.

The `file-get-buf` function opens the file as if using the `open-file` function, using a *mode-string* of `"r"`. If the *mode-opts* is present, it specifies *options* to be added to the string. These must be compatible with the implicit `"r"` mode.

9.19.57 Functions `file-put-buf`, `file-append-buf` and `command-put-buf`

Syntax:

```
(file-put-buf name buf skip-bytes [mode-opts])
(file-place-buf name buf skip-bytes [mode-opts])
(file-append-buf name buf [mode-opts])
(command-put-buf cmd buf)
```

Description:

The `file-put-buf` function opens a text stream over the file indicated by the string argument *name*, writes the contents of the buffer object *buf* into the file, and then closes the file. If the file doesn't exist, it is created. If it exists, it is truncated to zero length and overwritten. The default value of the optional *skip-bytes* parameter is zero. If an argument is given, it must be a nonnegative integer. If it is nonzero, then after opening the file, before writing the buffer, the function will seek to an offset of that many bytes from the start of the file. The contents of *buf* will be written at that offset.

The `file-place-buf` function does not truncate an existing file to zero length. In all other regards, it is equivalent to `file-put-buf`.

The `file-append-buf` function is similar to `file-put-buf` except that if the file exists, it

isn't overwritten. Rather, the buffer is appended to the file.

The `command-put-buf` function opens an output text stream over an output command pipe created for the command specified in the string argument *cmd*, as if by the `open-command` function. It then writes the contents of buffer *buf* into the stream and closes the stream.

The `file-put-buf`, `file-place-buf` and `file-append-buf` functions open a file as if using the `open-file` function using, respectively, *mode-string* values of `"wb"`, `"mb"`, and `"ab"`.

The *mode-opts* argument, if present, specifies additional *options* to be added to these modes.

The return value of all three functions is that of the `put-buf` operation which is implicitly performed.

9.19.58 Functions `buf-str` and `str-buf`

Syntax:

```
(buf-str str [null-term-p])
(str-buf buf [null-term-p])
```

Description:

The `buf-str` and `str-buf` functions perform UTF-8 conversion between the character string and buffer data types.

The `buf-str` function UTF-8-encodes *str* and returns a buffer containing the converted representation. If a true argument is given to the *null-term-p* parameter, then a null terminating byte is added to the buffer. This byte is added even if the previous byte is already a null byte from the conversion of a pseudo-null character occurring in *str*.

The `str-buf` function takes the contents of buffer *buf* to be UTF-8 data, which is converted to a character string and returned. Null bytes in the buffer are mapped to the pseudo-null character `#\xDC00`. If a true argument is given to the *null-term-p* parameter, then if the contents of *buf* end in a null byte, that byte is not included in the conversion.

9.19.59 Functions `buf-int` and `buf-uint`

Syntax:

```
(buf-int integer)
(buf-uint integer)
```

Description:

The `buf-int` and `buf-uint` functions convert a signed and unsigned integer, respectively, or else a character, into a binary representation, which is returned as a buffer object.

Under both functions, the representation uses big endian byte order: most significant byte first.

The `buf-uint` function requires a nonnegative *integer* argument, which may be a character. The representation stored in the buffer is a pure binary representation of the value using the smallest number of bytes required for the given *integer* value.

The `buf-int` function requires an integer or character argument. The representation stored in the buffer is a two's complement representation of *integer* using the smallest number of bytes which can represent that value. If *integer* is nonnegative, then the first byte of the buffer lies in

the range 0 to 127. If *integer* is negative, then the first byte of the buffer lies in the range 128 to 255. The integer 255 therefore doesn't convert to the buffer `#b'ff'` but rather `#b'00ff'`. The buffer `#b'ff'` represents -1.

If the *integer* argument is a character object, it is taken to be its Unicode code point value, as returned by the `int-chr` function.

9.19.60 Functions `int-buf` and `uint-buf`

Syntax:

```
(int-buf buf)
(uint-buf buf)
```

Description:

The `int-buf` and `uint-buf` functions recover an integer value from its binary form which appears inside *buf*, which must be a buffer object. These functions expect *buf* to contain the representation produced by, respectively, the functions `buf-int` and `buf-uint`.

If *buf* holds the representation of an integer value *n*, as produced by `(buf-int n)` then `(int-buf buf)` returns *n*.

The same relationship holds between `buf-uint` and `uint-buf`.

Thus, these equalities hold:

```
(= (int-buf (buf-int n)) n)
(= (uint-buf (buf-uint n)) n)
```

provided that *n* is of integer type and, in the case of `buf-uint`, nonnegative.

9.19.61 Functions `buf-compress` and `buf-decompress`

Syntax:

```
(buf-compress buf [level])
(buf-decompress buf)
```

Description:

The `buf-compress` and `buf-decompress` functions perform compression using the Deflate algorithm, via Zlib. These functions are only available if **TXR** is built with Zlib support. More specifically, `buf-compress` uses Zlib's `compress2` function; therefore it can be expected to interoperate with other software which uses the same function.

The `buf-compress` function compresses the entire contents of *buf* and returns new buffer with the compressed contents. The optional *level* argument specifies the compression level as an integer. Valid values range from 0 (no compression) to 9 (maximum compression). The value -1 selects a default compression determined internally by Zlib.

The `buf-decompress` function reverses the `buf-compress` operation: it takes a compressed *buf* and returns a buffer containing the original uncompressed data.

The `buf-compress` function throws an error exception if the *level* value is unacceptable to Zlib. The `buf-decompress` function throws an error exception if *buf* doesn't contain a compressed image.

9.20 Structures

TXR supports user-defined types in the form of structures. Structures are objects which hold multiple storage locations called slots, which are named by symbols. Structures can be related to each other by inheritance. Multiple inheritance is permitted.

The type of a structure is itself an object, of type `struct-type`.

When the program defines a new structure type, it does so by creating a new `struct-type` instance, with properties which describe the new structure type: its name, its list of slots, its initialization and "boa constructor" functions, and the structures type it inherits from (the *supertypes*).

The `struct-type` object is then used to generate instances.

Structures instances are not only containers which hold named slots, but they also indicate their struct type. Two structures which have the same number of slots having the same names are not necessarily of the same type.

Structure types and structures may be created and manipulated using a programming interface based on functions.

For more convenient and clutter-free expression of structure-based program code, macros are also provided.

Furthermore, concise and expressive slot access syntax is provided courtesy of the referencing dot and unbound referencing dot syntax, a syntactic sugar for the `qref` and `uref` macros.

Structure types have a name, which is a symbol. The `typeof` function, when applied to any struct type, returns the symbol `struct-type`. When `typeof` is applied to a struct instance, it returns the name of the struct type. Effectively, struct names are types.

The consequences are unspecified if an existing struct name is reused for a different struct type, or an existing type name is used for a struct type.

9.20.1 Static Slots

Structure slots can be of two kinds: they can be the ordinary instance slots or they can be static slots. The instances of a given structure type have their own instance of a given instance slot. However, they all share a single instance of a static slot.

Static slots are allocated in a global area associated with a structure type and are initialized when the structure type is created. They are useful for efficiently representing properties which have the same value for all instances of a struct. These properties don't have to occupy space in each instance, and time doesn't have to be wasted initializing them each time a new instance is created. Static slots are also useful for struct-specific global variables. Lastly, static slots are also useful for holding methods and functions. Although structures can have methods and functions in their instances, usually, all structures of the same type share the same functions. The `defstruct` macro supports a special syntax for defining methods and struct-specific functions at the same time when a new structure type is defined. The `defmeth` macro can be used for adding new methods and functions to an existing structure and its descendants.

Static slots may be assigned just like instance slots. Changing a static slot changes that slot in every structure of the same type.

Static slots are not listed in the `#S(...)` notation when a structure is printed. When the structure notation is read from a stream, if static slots are present, they will be processed and their values stored in the static locations they represent, thus changing their values for all instances.

Static slots are inherited just like instance slots. The following simplified discussion is restricted to single inheritance. A detailed description of multiple inheritance is given in the Multiple Inheritance section below. If a given structure *B* has some static slot *s*, and a new structure *D* is derived from *B*, using `def-struct`, and does not define a slot *s*, then *D* inherits *s*. This means that *D* shares the static slot with *B*: both types share a single instance of that slot.

On the other hand if `D` defines a static slot *s* then that slot will have its own instance in the *D* structure type; *D* will not inherit the *B* instance of slot *s*. Moreover, if the definition of `D` omits the *init-form* for slot *s*, then that slot will be initialized with a copy of the current value of slot *s* of the *B* base type, which allows derived types to obtain the value of base type's static slot, yet have that in their own instance.

The slot type can be overridden. A structure type deriving from another type can introduce slots which have the same names as the supertype, but are of a different kind: an instance slot in the supertype can be replaced by a static slot in the derived type or vice versa.

Note that, in light of the above type overriding possibility, the static slot value propagation happens only from the immediate supertype. If `D` is derived from `G` which has a static slot `s`, whereas `D` specifies `s` as an instance slot, but then `B` again specifies a static slot `s`, then `B`'s slot `s` will not inherit the value from `G`'s `s` slot. Simply, `B`'s supertype is `D` and that supertype is not considered to have a static slot `s`.

A structure type is associated with a static initialization function which may be used to store initial values into static slots. This function is invoked once in a type's life time, when the type is created. The function is also inherited by derived struct types and invoked when they are created.

9.20.2 Multiple Inheritance

When a structure type is defined, two or more supertypes may be specified. The new structure type then potentially inherits instance and static slots from all of the specified supertypes, and is considered to be a subtype of all of them. This situation with two or more supertypes is called *multiple inheritance*. The contrasting term is *single inheritance*, denoting the situation when a structure has exactly one supertype. **TXR Lisp**'s struct types initially permitted only single inheritance. Multiple inheritance support was introduced in version 229, as a straightforward extension of single inheritance semantics.

In the `make-struct-type` function and `defstruct` macro, a list of supertypes can be given instead of just one. The type then inherits slots from all of the specified types. If any conflicts arise among the supertypes due to slots having the same name, the leftmost supertype dominates: that type's slot will be inherited. If the leftmost slot is static, then that static slot will be inherited. Otherwise, the instance slot will be inherited.

Of course, any slot which is specified in the newly defined type itself dominates over any same-named slots among the supertypes.

The new structure type inherits all of the slot initializing expressions, as well as `:init` and `:postinit` methods of all of its supertypes.

Each time the structure is instantiated, the `:init` initializing expressions inherited from the supertypes, together with the slot initializing expressions, are all evaluated, in right-to-left order: the initializations contributed by each supertype are performed before considering the next supertype to the left. The `:postinit` methods are similarly invoked in right-to-left order, before the `:postinit` methods of the new type itself. Thus the order is: supertype inits, own inits, supertype post-inits, own post-inits.

9.20.3 Duplicate Supertypes

Multiple inheritance makes it possible for a type to inherit the same supertype more than once, either directly (by naming it more than once as a direct supertype) or indirectly (by inheriting two or more

different types, which have a common ancestor). The latter situation is sometimes referred to as the *diamond problem*.

Until **TXR** 242, the situation of duplicate supertypes was ignored for the purposes of object initialization. It was documented that if a supertype is referenced by inheritance, directly or indirectly, two or more times, then its initializing expressions are evaluated that many times.

Starting in **TXR** 243, duplicate supertypes no longer give rise to duplicate initialization. When an object is instantiated, only one initialization of a duplicated supertype occurs. The subsequent initializations that would take place in the absence of duplicate detection are suppressed.

Note also that the `:fini` mechanism is tied to initialization. Initialization of an object registers the finalizers, and so in **TXR** 242, `:fini` finalizers are also executed multiple times, if `:init` initializers are.

Examples:

Consider following program:

```
(defstruct base ()
  (:init (me) (put-line "base init"))
  (:fini (me) (put-line "base fini")))

(defstruct d1 (base)
  (:init (me) (put-line "d1 init"))
  (:fini (me) (put-line "d1 fini")))

(defstruct d2 (base)
  (:init (me) (put-line "d2 init"))
  (:fini (me) (put-line "d2 fini")))

(defstruct s (d1 d2))

(call-finalizers (new s))
```

Under **TXR** 242, and earlier versions that support multiple inheritance, it produces the output:

```
base init
d2 init
base init
d1 init
d1 fini
base fini
d2 fini
base fini
```

The supertypes are initialized in a right-to-left traversal of the type lattice, without regard for `base` being duplicated.

Starting with **TXR** 243, the output is:

```
base init
d2 init
d1 init
d1 fini
d2 fini
```

```
base fini
```

The rightmost duplicate of the base is initialized, so that the initialization is complete prior to the initializations of any dependent types. Likewise, the same rightmost duplicate of the base is finalized, so that finalization takes place after that of any dependent struct types.

Note, however, that the `derived` function function mechanism is not required to detect duplicated direct supertypes. If a supertype implements the `derived` function to detect situations when it is the target of inheritance, and some subtype inherits that type more than once, that function may be called more than once. The behavior is unspecified.

9.20.4 Dirty Flags

All structure instances contain a Boolean flag called the *dirty flag*. This flag is not a slot, but rather a metadata property that is exposed to program access. When the flag is set, an object is said to be dirty; otherwise it is clean.

Newly constructed objects come into existence dirty. The dirty flag state can be tested with the function `test-dirty`. An object can be marked as clean by clearing its dirty flag with `clear-dirty`. A combined operation `test-clear-dirty` is provided which clears the dirty flag, and returns its previous value.

The dirty flag is set whenever a new value is stored into the instance slot of an object.

Note: the dirty flag can be used to support support the caching of values derived from an object's slots. The derived values don't have to be recomputed while an object remains clean.

9.20.5 Equality Substitution

In object-based or object-oriented programming, sometimes it is necessary for a new data type to provide its own notion of equality: its own requirements for when two distinct instances of the type are considered equal. Furthermore, types sometimes have to implement their own notion, also, of inequality: the requirements for the manner in which one instance is considered lesser or greater than another.

TXR Lisp structures implement a concept called *equality substitution* which provides a simple, unified way for the implementor of an object to encode the requirements for both equality and inequality. Equality substitution allows for objects to be used as keys in a hash table according to the custom equality, without the programmer being burdened with the responsibility of developing a custom hashing function.

An object participates in equality substitution by implementing the `equal` method. The `equal` method takes no arguments other than the object itself. It returns a representative value which is used in place of that object for the purposes of `equal` comparison.

Whenever an object which supports equality substitution is used as an argument of any of the functions `equal`, `nequal`, `greater`, `less`, `gequal`, `lequal` or `hash-equal`, the `equal` method of that object is invoked, and the return value of that method is taken in place of that object.

The same is true if an object which supports equality substitution is used as a key in an `:equal-based` hash table.

The substitution is applied repeatedly: if the return value of the object's `equal` method is an object which itself supports equality substitution, than that returned object's method is invoked on that object to fetch its equality substitute. This repeats as many times as necessary until an object is determined which isn't a structure that supports equality substitution.

Once the equality substitute is determined, then the given function proceeds with the replacement object. Thus for example `equal` compares the replacement object in place of the original, and an `:equal-based` hash table uses the replacement object as the key for the purposes of hashing and comparison.

9.20.6 Custom Slot Expansion

The `defstruct` macro has a provision for for application-defined clauses, which may be defined using the `define-struct-clause` macro. This macro associates new clause keywords with custom expansion. The `:delegate` clause of `defstruct` is in fact implemented externally to `defstruct` using `define-struct-clause`.

9.20.7 Custom Preludes

The `defstruct` macro has a provision for implicit inclusion of application-defined clauses called preludes, which are previously defined via the `define-struct-prelude` macro. During macro-expansion, `defstruct` checks whether the structure being defined is the target of one or more preludes. If so, it includes the clauses from those preludes as if they were written directly in the `defstruct` syntax.

9.20.8 Macro `defstruct`

Syntax:

```
(defstruct {name | (name arg*)} super
   slot-specifier*)
```

Description:

The `defstruct` macro defines a new structure type and registers it under *name*, which must be a bindable symbol, according to the `bindable` function. Likewise, the name of every *slot* must also be a bindable symbol.

The *super* argument must either be `nil`, or a symbol which names an existing struct type, or else a list of such symbols. The newly defined struct type will inherit all slots, as well as initialization behaviors from the specified struct types.

The `defstruct` macro is implemented using the `make-struct-type` function, which is more general. The macro analyzes the `defstruct` argument syntax, and synthesizes arguments which are then used to call the function. Some remarks in the description of `defstruct` only apply to structure types defined using that macro.

Slots are specified using zero or more *slot-specifier* clauses.

Application-defined clauses are possible via `define-struct-clause`. The `defstruct` macro may bring in prelude clauses which are not specified in its syntax, but that have been specified using `define-struct-prelude`.

The following built-in clauses are supported:

name
: The simplest slot specifier is just a name, which must be a bindable symbol, as defined by the `bindable` function. This form is a short form for the `(:instance name)` syntax.

`(`*name init-form*`)`
: This syntax is a short form for the `(:instance name init-form)` syntax.

`(:instance` *name* `[`*init-form*`])`
: This syntax specifies an instance slot called *name* whose initial value is obtained by evaluating *init-form* whenever a new instance of the structure is created. This evaluation

takes place in the original lexical environment in which the `defstruct` form occurs. If *`init-form`* is omitted, the slot is initialized to `nil`.

`(:static` *`name`* `[`*`init-form`*`])`

This syntax specifies a static slot called *`name`* whose initial value is obtained by evaluating *`init-form`* once, during the evaluation of the `defstruct` form in which it occurs, if the *`init-form`* is present. If *`init-form`* is absent, and a static slot with the same name exists in the *`super`* base type, then this slot is initialized with the value of that slot. Otherwise it is initialized to `nil`.

The definition of a static slot in a `defstruct` causes the new type to have its own instance that slot, even if a same-named static slot occurs in the *`super`* base type, or its bases.

`(:method` *`name`* `(`*`param`*`+) `*`body-form`*`*)`

This syntax creates a static slot called *`name`* which is initialized with an anonymous function. The anonymous function is created during the evaluation of the `defstruct` form. The function takes the arguments specified by the *`param`* symbols, and its body consists of the *`body-form`*s. There must be at least one *`param`*. When the function is invoked as a method, as intended, the leftmost *`param`* receives the structure instance. The *`body-form`*s are evaluated in a context in which a block named *`name`* is visible. Consequently, `return-from` may be used to terminate the execution of a method and return a value. Methods are invoked using the `instance.(name arg ...)` syntax, which implicitly inserts the instance into the argument list.

Due to the semantics of static slots, methods are naturally inherited from a base structure to a derived one, and defining a method in a derived class which also exists in a base class performs OOP-style overriding.

`(:function` *`name`* `(`*`param`*`*) `*`body-form`*`*)`

This syntax creates a static slot called *`name`* which is initialized with an anonymous function. The anonymous function is created during the evaluation of the `defstruct` form. The function takes the arguments specified by the *`param`* symbols, and its body consists of the *`body-form`*s. This specifier differs from `:method` only in one respect: there may be zero parameters. A structure function defined this way is intended to be used as a utility function which doesn't receive the structure instance as an argument. The *`body-form`*s are evaluated in a context in which a block named *`name`* is visible. Consequently, `return-from` may be used to terminate the execution of the function and return a value. Such functions are called using the `(call instance.name arg ...)` or else the DWIM brackets syntax `[instance.name arg ...]`.

The remarks about inheritance and overriding in the description of `:method` also apply to `:function`.

`(:init (`*`param`*`) `*`body-form`*`*)`

The `:init` specifier doesn't describe a slot. Rather, it specifies code which is executed when a structure is instantiated, before the slot initializations specific to the structure type are performed. The code consists of *`body-form`*s which are evaluated in order in a lexical scope in which the variable *`param`* is bound to the structure object.

Multiple `:init` specifiers may appear in the same `defstruct` form. They are executed in their order of appearance, left to right.

When an object with one or more levels of inheritance is instantiated, the `:init` code of a base structure type, if any, is executed before any initializations specific to a derived structure type. Under multiple inheritance, the `:init` code of the rightmost base type is executed first, then that of the remaining bases in right-to-left order.

The `:init` initializations are executed before any other slot initializations. The argument values passed to the `new` or `lnew` operator or the `make-struct` function are not yet stored in the object's slots, and are not accessible. Initialization code which needs these values to be stable can be defined with `:postinit`.

Initializers in base structures must be careful about assumptions about slot kinds, because derived structures can alter static slots to instance slots or vice versa. To avoid an unwanted initialization being applied to the wrong kind of slot, initialization code can be made conditional on the outcome of `static-slot-p` applied to the slot. (Code generated by `defstruct` for initializing instance slots performs this kind of check).

The *body-form*s of an `:init` specifier are not surrounded by an implicit `block`.

`(:postinit (`*param*`) `*body-form**`)`

The `:postinit` specifier is similar to `:init`. Both specify forms which are evaluated during object instantiation. The difference is that the `body-form`s of a `:postinit` are evaluated after other initializations have taken place, including the `:init` initializations, as a second pass. By the time `:postinit` initialization runs, the argument material from the `make-struct`, `new` or `lnew` invocation has already been processed and stored into slots. Like `:init` actions, `:postinit` actions registered at different levels of the type's inheritance hierarchy are invoked in the base-to-derived order, in right-to-left order among multiple bases at the same level. Multiple `:postinit` form in the same `defstruct` are invoked in left-to-right order.

`(:fini (`*param*`) `*body-form**`)`

The `:fini` specifier doesn't describe a slot. Rather, it specifies a finalization function which is associated with the structure instance, as if by use of the `finalize` function. This finalization registration takes place as the first step when an instance of the structure is created, before the slots are initialized and the `:init` code, if any, has been executed. The registration takes place as if by the evaluation of the form `(finalize `*obj*` (lambda (`*param*`) `*body-form*`...) t)` where *obj* denotes the structure instance. Note the `t` argument which requests reverse order of registration, ensuring that if an object has multiple finalizers registered at different levels of inheritance hierarchy, the finalizers specified for a derived structure type are called before inherited finalizers.

The *body-form*s of a `:fini` specifier are not surrounded by an implicit `block`.

Multiple `:fini` clauses may be specified in the same `defstruct`, in which case they are invoked in reverse, right-to-left order.

Note that an object's finalizers can be called explicitly with `call-finalizers`. Note: the `with-objects` macro arranges for finalizers to be called on objects when the execution of a scope terminates by any means.

`(:postfini (`*param*`) `*body-form**`)`

Like `:fini`, `:postfini` specifier doesn't describe a slot. The syntax is identical to `:fini`. Independently of whether `:fini` is specified, at most one `:postfini` may be specified. The only difference between `:fini` and `:postfini` is that `:postfini` arranges for a finalizer to be registered as if by the evaluation of the form `(finalize `*obj*` (lambda (`*param*`) `*body-form*`...))` where *obj* denotes the structure instance. Note the that unlike `:fini`, this omits the `t` parameter, which means that `:postfini` finalizers of derived structures execute after the execution of inherited finalizers. It also means that multiple `:postfini` finalizers appearing in the same

defstruct execute in left-to-right order unlike the reverse right-to-left order of :fini finalizers.

When both :fini and :postfini clauses are specified in the same defstruct form, all the :postfini finalizers execute after all the :fini finalizers regardless of the order in which they appear.

(:inherit *super**)
: The :inherit clause specifies zero or more types to be inherited. Each *super* argument must be a symbol which is the name of an existing struct type. These symbols are appended to the list of supertypes coming from the *super* argument defstruct. Note: the motivation behind :inherit is to make it possible for struct clauses defined by define-struct-clause to inject supertypes. Developers are encouraged to use the regular *super* argument of defstruct to declare inheritance of supertypes, rather than writing visible :inherit clauses that can be moved into the *super* argument.

The slot names given in a defstruct must all be unique among themselves, but they may match the names of existing slots in the *super* base type.

A given structure type can have only one slot under a given symbolic name. If a newly specified slot matches the name of an existing slot in the *super* type or that type's chain of ancestors, it is called a *repeated slot*.

The kind of the repeated slot (static or instance) is not inherited; it is established by the defstruct and may be different from the type of the same-named slot in the supertype or its ancestors.

If a repeated slot is introduced as a static slot, and has no *init-form* then it receives the current of the a static of the same name from the nearest supertype which has such a slot.

If a repeated slot is an instance slot, no such inheritance of value takes place; only the local *init-form* applies to it; if it is absent, the slot it initialized to nil in each newly created instance of the new type.

However, :init and :postinit initializations are inherited from a base type and they apply to the repeated slots, regardless of their kind. These initializations take place on the instantiated object, and the slot references resolve accordingly.

The initialization for slots which are specified using the :method or :function specifiers is reordered with regard to :static slots. Regardless of their placement in the defstruct form, :method and :function slots are initialized before :static slots. This ordering is useful, because it means that when the initialization expression for a given static slot constructs an instance of the struct type, any instance initialization code executing for that instance can use all functions and methods of the struct type. However, note the static slots which follow that slot in the defstruct syntax are not yet initialized. If it is necessary for a structure's initialization code to have access to all static slots, even when the structure is instantiated during the initialization of a static slot, a possible solution may be to use lazy instantiation using the lnew operator, rather than ordinary eager instantiation via new. It is also necessary to ensure that that the instance isn't accessed until all static initializations are complete, since access to the instance slots of a lazily instantiated structure triggers its initialization.

The structure name is specified using two forms, plain *name* or the syntax (*name arg**) If the second form is used, then the structure type will support "boa construction", where "boa" stands for "by order of arguments". The *arg*s specify the list of slot names which are to be initialized in the by-order-of-arguments style. For instance, if three slot names are given, then those slots can be

optionally initialized by giving three arguments in the `new` macro or the `make-struct` function.

Slots are first initialized according to their *`init-form`*s, regardless of whether they are involved in boa construction.

A slot initialized in this style still has a *`init-form`* which is processed independently of the existence of, and prior to, boa construction.

The boa constructor syntax can specify optional parameters, delimited by a colon, similarly to the `lambda` syntax. However, the optional parameters may not be arbitrary symbols; they must be symbols which name slots. Moreover, the `(`*`name init-form`* `[`*`present-p`*`])` optional parameter syntax isn't supported.

When boa construction is invoked with optional arguments missing, the default values for those arguments come from the *`init-form`*s in the remaining `defstruct` syntax.

Examples:

```
(defvar *counter* 0)

;; New struct type foo with no super type:
;; Slots a and b initialize to nil.
;; Slot c is initialized by value of (inc *counter*).
(defstruct foo nil (a b (c (inc *counter*))))

(new foo) -> #S(foo a nil b nil c 1)
(new foo) -> #S(foo a nil b nil c 2)

;; New struct bar inheriting from foo.
(defstruct bar foo (c 0) (d 100))

(new bar) -> #S(bar a nil b nil c 0 d 100)
(new bar) -> #S(bar a nil b nil c 0 d 100)

;; counter was still incremented during
;; construction of d:
*counter* -> 4

;; override slots with new arguments
(new foo a "str" c 17) -> #S(foo a "str" b nil c 17)

*counter* -> 5

;; boa initialization
(defstruct (point x : y) nil (x 0) (y 0))

(new point) -> #S(point x 0 y 0)
(new (point 1 1)) -> #S(point x 1 y 1)

;; property list style initialization
;; can always be used:
(new point x 4 y 5) -> #S(point x 4 y 5)

;; boa applies last:
(new (point 1 1) x 4 y 5) -> #S(point x 1 y 1)
```

```
;; boa with optional argument omitted:
(new (point 1)) -> #S(point x 1 y 0)

;; boa with optional argument omitted and
;; with property list style initialization:
(new (point 1) x 5 y 5) -> #S(point x 1 y 5)
```

9.20.9 Macro `defmeth`

Syntax:

```
(defmeth type-name name param-list body-form*)
```

Description:

Unless *name* is one of the two symbols `:init` or `:postinit`, the `defmeth` macro installs a function into the static slot named by the symbol *name* in the struct type indicated by *type-name*.

If the structure type doesn't already have such a static slot, it is first added, as if by the `static-slot-ensure` function, subject to the same checks.

If the function has at least one argument, it can be used as a method. In that situation, the leftmost argument passes the structure instance on which the method is being invoked.

The function takes the arguments specified by the *param-list* symbols, and its body consists of the *body-form*s.

The *body-form*s are placed into a `block` named `name`.

A method named `lambda` allows a structure to be used as if it were a function. When arguments are applied to the structure as if it were a function, the `lambda` method is invoked with those arguments, with the object itself inserted into the leftmost argument position.

If `defmeth` is used to redefine an existing method, the semantics can be inferred from that of `static-slot-ensure`. In particular, the method will be imposed into all subtypes which inherit (do not override) the method.

If *name* is the keyword symbol `:init`, then instead of operating on a static slot, the macro redefines the *initfun* of the given structure type, as if by a call to the function `struct-set-initfun`.

Similarly, if *name* is the keyword symbol `:postinit`, then the macro redefines the *postinitfun* of the given structure type, as if by a call to the function `struct-set-postinitfun`.

When redefining `:initfun` the admonishments given in the description of `struct-set-initfun` apply: if the type has an *initfun* generated by the `defstruct` macro, then that *initfun* is what implements all of the slot initializations given in the slot specifier syntax. These initializations are lost if the *initfun* is overwritten.

The `defmeth` macro returns a method name: a unit of syntax of the form `(meth` *type-name name*`)` which can be used as an argument to the accessor `symbol-function` and other situations.

9.20.10 Macros `new` and `lnew`

Syntax:

```
(new {name | (name arg*)} {slot init-form}*)
(lnew {name | (name arg*)} {slot init-form}*)
```

Description:

The `new` macro creates a new instance of the structure type named by *name*.

If the structure supports "boa construction", then, optionally, the arguments may be given using the syntax `(name arg*)` instead of *name*.

Slot values may also be specified by the *slot* and *init-form* arguments.

Note: the evaluation order in `new` is surprising: namely, *init-form*s are evaluated before *arg*s if both are present.

When the object is constructed, all default initializations take place first. If the object's structure type has a supertype, then the supertype initializations take place. Then the type's initializations take place, followed by the *slot init-form* overrides from the `new` macro, and lastly the "boa constructor" overrides.

If any of the initializations abandon the evaluation of `new` by a nonlocal exit such as an exception throw, the object's finalizers, if any, are invoked.

The macro `lnew` differs from new in that it specifies the construction of a lazy struct, as if by the `make-lazy-struct` function. When `lnew` is used to construct an instance, a lazy struct is returned immediately, without evaluating any of the *arg* and *init-form* expressions. The expressions are evaluated when any of the object's instance slots is accessed for the first time. At that time, these expressions are evaluated (in the same order as under `new`) and initialization proceeds in the same way.

If any of the initializations abandon the delayed initializations steps arranged by `lnew` by a nonlocal exit such as an exception throw, the object's finalizers, if any, are invoked.

Lazy initialization does not detect cycles. Immediately prior to the lazy initialization of a struct, the struct is marked as no longer requiring initialization. Thus, during initialization, its instance slots may be freely accessed. Slots not yet initialized evaluate as `nil`.

9.20.11 Macros `new*` and `lnew*`

Syntax:

```
(new* {expr | (expr arg*)} {slot init-form}*)
(lnew* {expr | (expr arg*)} {slot init-form}*)
```

Description:

The `new*` and `lnew*` macros are variants, respectively, of `new` and `lnew`.

The difference in behavior in these macros relative to `new` and `lnew` is that the *name* argument is replaced with an expression *expr* which is evaluated. The value of *expr* must be a struct type, or a symbol which is the name of a struct type.

With one exception, if *expr0* is a compound expression, then `(new* expr0 ...)` is interpreted as `(new* (expr1 args...) ...)` where the head of *expr0*, *expr1*, is actually

the expression which is evaluated to produce the type, and the remaining constituents of *expr0*, *args*, become the boa arguments. The same requirement applies to `lnew*`.

The exception is that if *expr1* is the symbol `dwim`, this interpretation does not apply. Thus `(new* [fun args...] ...)` evaluates the `[fun args...]` expression, rather than treating it as `(dwim fun args...)` where `dwim` would be evaluated as a variable reference expected to produce a type.

Examples:

```
;; struct with boa constructor
(defstruct (ab a : b) () a b)

;; error: find-struct-type is interpreted as a variable
(new* (find-struct-type 'ab) a 1) -> ;; error

;; OK: extra nesting.
(new* ((find-struct-type 'ab)) a 1) -> #S(ab a 1 b nil)

;; OK: dwim brackets without nesting.
(new* [find-struct-type 'ab] a 1) -> #S(ab a 1 b nil)

;; boa construction
(new* ([find-struct-type 'ab] 1 2)) -> #S(ab a 1 b 2)
(new* ((find-struct-type 'ab) 1 2)) -> #S(ab a 1 b 2)

;; mixed construction
(new* ([find-struct-type 'ab] 1) b 2) -> #S(ab a 1 b 2)

(let ((type (find-struct-type 'ab)))
  (new* type a 3 b 4))
-> #S(ab a 3 b 4)

(let ((type (find-struct-type 'ab)))
  (new* (type 3 4)))
-> #S(ab a 3 b 4)
```

9.20.12 Macro `with-slots`

Syntax:

```
(with-slots ({slot | (sym slot)}*) struct-expr
   body-form*)
```

Description:

The `with-slots` binds lexical macros to serve as aliases for the slots of a structure.

The *struct-expr* argument is expected to be an expression which evaluates to a struct object. It is evaluated once, and its value is retained. The aliases are then established to the slots of the resulting struct value.

The aliases are specified as zero or more expressions which consist of either a single symbol *slot* or a `(sym slot)` pair. The simple form binds a macro named *slot* to a slot also named *slot*. The pair form binds a macro named *sym* to a slot named *slot*.

The lexical aliases are syntactic places: assigning to an alias causes the value to be stored into the slot which it denotes.

After evaluating *struct-expr* the `with-slots` macro arranges for the evaluation of *body-form*s in the lexical scope in which the aliases are visible.

Dialect Notes:

The intent of the `with-slots` macro is to help reduce the verbosity of code which makes multiple references to the same slot. Use of `with-slots` is less necessary in **TXR Lisp** than other Lisp dialects thanks to the dot operator for accessing struct slots.

Lexical aliases to struct places can also be arranged with considerable convenience using the `placelet` operator. However, `placelet` will not bind multiple aliases to multiple slots of the same object such that the expression which produces the object is evaluated only once.

Example:

```
(defstruct point nil x y)

;; Here, with-slots introduces verbosity because
;; each slot is accessed only once. The function
;; is equivalent to:
;;
;; (defun point-delta (p0 p1)
;;   (new point x (- p1.x p0.x) y (- p1.y p0.y)))
;;
;; Also contrast with the use of placelet:
;;
;; (defun point-delta (p0 p1)
;;   (placelet ((x0 p0.x) (y0 p0.y)
;;              (x1 p1.x) (y1 p1.y))
;;     (new point x (- x1 x0) y (- y1 y0)))))

(defun point-delta (p0 p1)
  (with-slots ((x0 x) (y0 y)) p0
    (with-slots ((x1 x) (y1 y)) p1
      (new point x (- x1 x0) y (- y1 y0)))))
```

9.20.13 Macro `qref`

Syntax:

```
(qref object-form
   {slot | (slot arg*) | [slot arg*]}+)
```

Description:

The `qref` macro ("quoted reference") performs structure slot access. Structure slot access is more conveniently expressed using the referencing dot notation, which works by translating to qref `qref` syntax, according to the following equivalence:

```
a.b.c.d <--> (qref a b c d)  ;; a b c d must not be numbers
```

(See the Referencing Dot section under Additional Syntax.)

The leftmost argument of `qref` is an expression which is evaluated. This argument is followed by one or more reference designators. If there are two or more designators, the following equivalence applies:

```
(qref obj d1 d2 ...)  <---> (qref (qref obj d1) d2 ...)
```

That is to say, `qref` is applied to the object and a single designator. This must yield an object, which to which the next designator is applied as if by another `qref` operation, and so forth.

If the null-safe syntax `(t ...)` is present, the equivalence becomes more complicated:

```
(qref (t obj) d1 d2 ...)  <---> (qref (qref (t obj) d1) d2 ...)

(qref obj (t d1) d2 ...)  <---> (qref (t (qref obj d1)) d2 ...)
```

Thus, `qref` can be understood in terms of the semantics of the binary form `(qref` *`object-form designator`*`)`

Designators come in three basic forms: a lone symbol, an ordinary compound expression consisting of a symbol followed by arguments, or a DWIM expression consisting of a symbol followed by arguments.

A lone symbol designator indicates the slot of that name. That is to say, the following equivalence applies:

```
(qref o n)  <-->  (slot o 'n)
```

where `slot` is the structure slot accessor function. Because `slot` is an accessor, this form denotes the slot as a syntactic place; slots can be modified via assignment to the `qref` form and the referencing dot syntax.

The slot name being implicitly quoted is the basis of the term "quoted reference", giving rise to the `qref` name.

A compound designator indicates that the named slot is a function, and arguments are to be applied to it. The following equivalence applies in this case, except that `o` is evaluated only once:

```
(qref o (n arg ...)) <--> (call (slot o 'n) o arg ...)
```

A DWIM designator similarly indicates that the named slot is a function, and arguments are to be applied to it. The following equivalence applies:

```
(qref obj [name arg ...])  <-->  [(slot obj 'name) o arg ...]
```

Therefore, under this equivalence, this syntax provides the usual Lisp-1-style evaluation rule via the `dwim` operator.

If the *`object-form`* has the syntax `(t` *`expression`*`)` this indicates null-safe access: if *`expression`* evaluates to `nil` then the entire expression `(qref (t` *`expression`*`)` *`designator`*`)` form yields `nil`. This syntax is produced by the `.?` notation.

The null-safe access notation prevents not only slot access, but also method or function calls on

nil. When a method or function call is suppressed due to the object being nil, no aspect of the method or function call is evaluated; not only is the slot not accessed, but the argument expressions are not evaluated.

Example:

```
(defstruct foo nil
  (array (vec 1 2 3))
  (increment (lambda (self index delta)
               (inc [self.array index] delta))))

(defvarl s (new foo))

;; access third element of s.array:
[s.array 2]  -->  3

;; increment first element of array by 42
s.(increment 0 42)  -->  43

;; access array member
s.array  -->  #(43 2 3)
```

Note how increment behaves much like a single-argument-dispatch object-oriented method. Firstly, the syntax s.(increment 0 42) effectively selects the increment function which is particular to the s object. Secondly, the object is passed to the selected function as the leftmost argument, so that the function has access to the object.

9.20.14 Macro uref

Syntax:

```
(uref {slot | (slot arg*) | [slot arg*]}+)
```

Description:

The uref macro ("unbound reference") expands to an expression which evaluates to a function. The function takes exactly one argument: an object. When the function is invoked on an object, it references slots or methods relative to that object.

Note: the uref syntax may be used directly, but it is also produced by the unbound referencing dot syntactic sugar:

```
.a          -->  (uref a)
.?a         -->  (uref t a)
.(f x)      -->  (uref (f x))
.(f x).b    -->  (uref (f x) b)
.a.(f x).b  -->  (uref a (f x) b)
```

The macro may be understood in terms of the following translation scheme:

```
(uref a b ...)    -->  (lambda (o) (qref o a b ...))
(uref t a b ...)  -->  (lambda (o) (if o (qref o a b ...)))
```

where o is understood to be a unique symbol (for instance, as produced by the gensym function).

When only one uref argument is present, these equivalences also hold:

```
(uref (f a b c ...))  <-->  (umeth f a b c ...)
(uref s)  <-->  (usl s)
```

The terminology "unbound reference" refers to the property that `uref` expressions produce a function which isn't bound to a structure object. The function binds a slot or method; the call to that function then binds an object to that function, as an argument.

Examples:

Suppose that the objects in `list` have slots `a` and `b`. Then, a list of the `a` slot values may be obtained using:

```
(mapcar .a list)
```

because this is equivalent to

```
(mapcar (lambda (o) o.a) list)
```

Because `uref` produces a function, its result can be operated upon by functional combinators. For instance, we can use the `juxt` combinator to produce a list of two-element lists, which hold the `a` and `b` slots from each object in `list`:

```
(mapcar (juxt .a .b) list)
```

9.20.15 Macro `meth`

Syntax:

```
(meth struct slot curried-expr*)
```

Description:

The `meth` macro allows indirection upon a method-like function stored in a function slot.

The `meth` macro binds *`struct`* as the leftmost argument of the function stored in *`slot`*, returning a function which takes the remaining arguments. That is to say, it returns a function *`f`* such that `[f arg ...]` calls `[struct.slot struct arg ...]` except that *`struct`* is evaluated only once.

If one or more *`curried-expr`* expressions are present, their values are bound inside *`f`* also, and when *`f`* is invoked, these are passed to the function stored in the slot. Thus if *`f`* is produced by `(meth struct slot c1 c2 c3 ...)` then `[f arg ...]` calls `[struct.slot struct c1v c2v c3v ... arg ...]` except that *`struct`* is evaluated only once, and *`c1v`*, *`c2v`* and *`c3v`* are the values of expressions `c1`, `c2` and `c3`.

The argument *`struct`* must be an expression which evaluates to a struct. The *`slot`* argument is not evaluated, and must be a symbol denoting a slot. The syntax can be understood as a translation to a call of the `method` function:

```
(meth a b)  <-->  (method a 'b)
```

If *`curried-arg`* expressions are present, the translation may be be understood as:

```
(meth a b c1 c2 ...)  <-->  [(fun method) a 'b c1 c2 ...]
```

In other words the *`curried-arg`* expressions are evaluated under the `dwim` operator evaluation

rules.

Example:

```
;; struct for counting atoms eq to key
(defstruct (counter key) nil
  key
  (count 0)
  (:method increment (self key)
    (if (eq self.key key)
      (inc self.count))))

;; pass all atoms in tree to func
(defun map-tree (tree func)
  (if (atom tree)
    [func tree]
    (progn (map-tree (car tree) func)
           (map-tree (cdr tree) func))))

;; count occurrences of symbol a
;; using increment method of counter,
;; passed as func argument to map-tree.
(let ((c (new (counter 'a)))
      (tr '(a (b (a a)) c a d)))
  (map-tree tr (meth c increment))
  c)
--> #S(counter key a count 4
               increment #<function: type 0>)
```

9.20.16 Macro `umeth`

Syntax:

```
(umeth slot curried-expr*)
```

Description:

The `umeth` macro binds the symbol *`slot`* to a function and returns that function.

The *`curried-expr`* arguments, if present, are evaluated as if they were arguments to the `dwim` operator.

When that function is called, it expects at least one argument. The leftmost argument must be an object of struct type.

The slot named *`slot`* is retrieved from that object, and is expected to be a function. That function is called with the object, followed by the values of the *`curried-expr`*s, if any, followed by that function's arguments.

The syntax can be understood as a translation to a call of the `umethod` function:

```
(umeth s ...)  <-->  [umethod 's ...]
```

The macro merely provides the syntactic sugar of not having to quote the symbol, and automatically treating the curried argument expressions using Lisp-1 semantics of the `dwim` operator.

Example:

```
;; seal and dog are variables which hold structures of
;; different types. Both have a method called bark.

(let ((bark-fun (umeth bark)))
  [bark-fun dog]     ;; same effect as dog.(bark)
  [bark-fun seal])   ;; same effect as seal.(bark)
```

The `u` in `umeth` stands for "unbound". The function produced by `umeth` is not bound to any specific object; it binds to an object whenever it is invoked by retrieving the actual method from the object's slot at call time.

9.20.17 Macro `usl`

Syntax:

```
(usl slot)
```

Description:

The `usl` macro binds the symbol *`slot`* to a function and returns that function.

When that function is called, it expects exactly one argument. That argument must be an object of struct type. The slot named *`slot`* is retrieved from that object and returned.

The name `usl` stands for "unbound slot". The term "unbound" refers to the returned function not being bound to a particular object. The binding of the slot to an object takes place whenever the function is called.

9.20.18 Function `make-struct-type`

Syntax:

```
(make-struct-type name super static-slots slots
                  static-initfun initfun boactor
                  boactor postinitfun)
```

Description:

The `make-struct-type` function creates a new struct type.

The *`name`* argument must be a bindable symbol, according to the `bindable` function. It specifies the name property of the struct type as well as the name under which the struct type is globally registered.

The *`super`* argument indicates the supertype for the struct type. It must be either a value of type `struct-type`, a symbol which names a struct type, or else `nil`, indicating that the newly created struct type has no supertype.

The *`static-slots`* argument is a list of symbol which specify static slots. The symbols must be bindable and the list must not contain duplicates.

The *`slots`* argument is a list of symbols which specifies the instance slots. The symbols must be bindable and there must not be any duplicates within the list, or against entries in the *`static-slots`* list.

The new struct type's effective list of slots is formed by appending together *`static-slots`* and

slots, and then appending that to the list of the supertype's slots, and de-duplicating the resulting list as if by the `uniq` function. Thus, any slots which are already present in the supertype are removed. If the structure has no supertype, then the list of supertype slots is taken to be empty. When a structure is instantiated, it shall have all the slots specified in the effective list of slots. Each instance slot shall be initialized to the value `nil`, prior to the invocation of *initfun* and *boactor*.

The *static-initfun* argument either specifies an initialization function, or is `nil`, which is equivalent to specifying a function which does nothing.

Prior to the invocation of *static-initfun*, each new static slot shall be initialized the value `nil`. Inherited static slots retain their values from the supertype.

If specified, *static-initfun* function must accept one argument. When the structure type is created (before the `make-struct-type` function returns) the *static-initfun* function is invoked, passed the newly created structure type as its argument.

The *initfun* argument either specifies an initialization function, or is `nil`, which is equivalent to specifying a function which does nothing. If specified, this function must accept one argument. When a structure is instantiated, every *initfun* in its chain of supertype ancestry is invoked, in order of inheritance, so that the root supertype's *initfun* is called first and the structure's own specific *initfun* is called last. These calls occur before the slots are initialized from the *arg* arguments or the *slot-init-plist* of `make-struct`. Each function is passed the newly created structure object, and may alter its slots. If multiple inheritance occurs, the *initfun* functions of multiple supertypes are called in right-to-left order.

The *boactor* argument either specifies a by-order-of-arguments initialization function ("boa constructor") or is `nil`, which is equivalent to specifying a constructor which does nothing. If specified, it must be a function which takes at least one argument. When a structure is instantiated, and boa arguments are given, the *boactor* is invoked, with the structure as the leftmost argument, and the boa arguments as additional arguments. This takes place after the processing of *initfun* functions, and after the processing of the *slot-init-plist* specified in the `make-struct` call. Note that the *boactor* functions of the supertypes are not called, only the *boactor* specific to the type being constructed.

The *postinitfun* argument either specifies an initialization function, or is `nil`, which is equivalent to specifying a function which does nothing. If specified, this function must accept one argument. The *postinitfun* function is similar to *initfun*. The difference is that *postinitfun* functions are called after all other initialization processing, rather than before. They are are also called in order of inheritance: the *postinitfun* of a structure's supertype is called before its own, and in right-to-left order among multiple supertypes under multiple inheritance.

9.20.19 Function `find-struct-type`

Syntax:

```
(find-struct-type name)
```

Description:

The `find-struct-type` returns a `struct-type` object corresponding to the symbol *name*.

If no struct type is registered under *name*, then it returns `nil`.

A `struct-type` object exists for each structure type and holds information about it. These objects are not themselves structures and are all of the same type, `struct-type`.

9.20.20 Function `struct-type-p`

Syntax:

```
(struct-type-p obj)
```

Description:

The `struct-type-p` function returns `t` if *`obj`* is a structure type, otherwise it returns `nil`.

A structure type is an object of type `struct-type`, returned by `find-struct-type`.

9.20.21 Function `struct-type-name`

Syntax:

```
(struct-type-name type-or-struct)
```

Description:

The `struct-type-name` function determines a structure type from the *`type-or-struct`* argument and returns that structure type's symbolic name.

The *`type-or-struct`* argument must be either a struct type object (such as the return value of a successful lookup via `find-struct-type`), a symbol which names a struct type, or else a struct instance.

9.20.22 Function `super`

Syntax:

```
(super [type-or-struct])
```

Description:

The `super` function determines a structure type from the *`type-or-struct`* argument and returns the struct type object which is the supertype of that type, or else `nil` if that type has no supertype.

The *`type-or-struct`* argument must be either a struct type object, a symbol which names a struct type, or else a struct instance.

9.20.23 Function `make-struct`

Syntax:

```
(make-struct type slot-init-plist arg*)
```

Description:

The `make-struct` function returns a new object which is an instance of the structure type *`type`*.

The *`type`* argument must either be a `struct-type` object, or else a symbol which is the name of a structure.

The *`slot-init-plist`* argument gives a list of slot initializations in the style of a property list, as defined by the `prop` function. It may be empty, in which case it has no effect. Otherwise, it specifies slot names and their values. Each slot name which is given must be a slot of the structure type. The corresponding value will be stored into the slot of the newly created object. If a slot is repeated, it is unspecified which value takes effect.

The optional *arg*s specify arguments to the structure type's boa constructor. If the arguments are omitted, the boa constructor is not invoked. Otherwise the boa constructor is invoked on the structure object and those arguments. The argument list must match the trailing parameters of the boa constructor (the remaining parameters which follow the leftmost argument which passes the structure to the boa constructor).

When a new structure is instantiated by `make-struct`, its slot values are first initialized by the structure type's registered functions as described under `make-struct-type`. Then, the *slot-init-plist* is processed, if not empty, and finally, the *arg*s are processed, if present, and passed to the boa constructor.

If any of the initializations abandon the evaluation of `make-struct` by a nonlocal exit such as an exception throw, the object's finalizers, if any, are invoked.

9.20.24 Function `make-lazy-struct`

Syntax:

```
(make-lazy-struct type argfun)
```

Description:

The `make-lazy-struct` function returns a new object which is an instance of the structure type *type*.

The *type* argument must either be a `struct-type` object, or else a symbol which is the name of a structure.

The *argfun* argument should be a function which can be called with no parameters and returns a cons cell. More requirements are specified below.

The object returned by `make-lazy-struct` is a lazily-initialized struct instance, or *lazy struct*.

A lazy struct remains uninitialized until just before the first access to any of its instance slots. Just before an instance slot is accessed, initialization takes place as follows. The *argfun* function is invoked with no arguments. Its return value must be a cons cell. The `car` of the cons cell is taken to be a property list, as defined by the `prop` function. The `cdr` field is taken to be a list of arguments. These values are treated as if they were, respectively, the *slot-init-plist* and the boa constructor arguments given in a `make-struct` invocation. Initialization of the structure proceeds as described in the description of `make-struct`.

9.20.25 Functions `struct-from-plist` and `struct-from-args`

Syntax:

```
(struct-from-plist type {slot value}*)
(struct-from-args type arg*)
```

Description:

The `struct-from-plist` and `struct-from-args` functions are interfaces to the `make-struct` function.

The `struct-from-plist` function passes its *slot* and *value* arguments as the *slot-init-plist* argument of `make-struct`. It passes no boa constructor arguments.

The `struct-from-args` function calls *make-struct* with an empty *slot-init-plist*, passing down the list of *arg*s.

The following equivalences hold:

```
(struct-from-plist a s0 v0 s1 v1 ...)
<-->  (make-struct a (list s0 v0 s1 v1 ...))

(struct-from-args a v0 v1 v2 ...)
<-->  (make-struct a nil v0 v1 v2 ...)
```

9.20.26 Function `allocate-struct`

Syntax:

```
(allocate-struct type)
```

Description:

The `allocate-struct` provides a low-level allocator for structure objects.

The *`type`* argument must either be a `struct-type` object, or else a symbol which is the name of a structure.

The `allocate-struct` creates and returns a new instance of *`type`* all of whose instance slots take on the value `nil`. No initializations are performed. The struct type's registered initialization functions are not invoked.

9.20.27 Function `copy-struct`

Syntax:

```
(copy-struct struct-obj)
```

Description:

The `copy-struct` function creates and returns a new object which is a duplicate of *`struct-obj`*, which must be a structure.

The duplicate object is a structure of the same type as *`struct-obj`* and has the same slot values.

The creation of a duplicate does not involve calling any of the struct type's initialization functions.

Only instance slots participate in the duplication. Since the original structure and copy are of the same structure type, they already share static slots.

This is a low-level, "shallow" copying mechanism. If an object design calls for a higher level cloning mechanism with deep copying or other additional semantics, one can be built on top of `copy-struct`. For instance, a structure can have a `copy` method similar to the following:

```
(:method copy (me)
  (let ((my-copy (copy-struct me)))
    ;; inform the copy that it has been created
    ;; by invoking its copied method.
    my-copy.(copied)
    my-copy))
```

which can then be invoked on whatever object needs copying. (Note that this method is not a special structure function, and is thus not taken into account by the `copy` function.) Since this logic is generic, it can be placed in a base method. The `copied` method which it calls is the means by which the new object is notified that it is a copy. This method takes on whatever special

responsibilities are required when a copy is produced, such as registering the object in various necessary associations, or performing a deeper copy of some of the objects held in the slots.

The `copied` handler can be implemented at multiple levels of an inheritance hierarchy. The initial call to `copied` from `copy` will call the most derived override of that method.

To call the corresponding method in the base class, a given derived method can use the `call-super-fun` function, or else the `(meth ...)` syntax in the first position of a compound form, in place of a function name. Examples of both are given in the documentation for `call-super-fun`.

Thus derived structs can inherit the copy handling logic from base structs, and extend it with their own.

9.20.28 Accessor `slot`

Syntax:

```
(slot struct-obj slot-name)
(set (slot struct-obj slot-name) new-value)
```

Description:

The `slot` function retrieves a structure's slot. The *`struct-obj`* argument must be a structure, and *`slot-name`* must be a symbol which names a slot in that structure.

Because `slot` is an accessor, a `slot` form is a syntactic place which denotes the slot's storage location.

A syntactic place expressed by `slot` does not support deletion.

9.20.29 Function `slotset`

Syntax:

```
(slotset struct-obj slot-name new-value)
```

Description:

The `slotset` function stores a value in a structure's slot.
The *`struct-obj`* argument must be a structure, and *`slot-name`* must be a symbol which names a slot in that structure.

The *`new-value`* argument specifies the value to be stored in the slot.

If a successful store takes place to an instance slot of *`struct-obj`*, then the dirty flag of that object is set, causing the `test-dirty` function to report true for that object.

The `slotset` function returns *`new-value`*.

9.20.30 Functions `test-dirty`, `clear-dirty` and `test-clear-dirty`

Syntax:

```
(test-dirty struct-obj)
(clear-dirty struct-obj)
(test-clear-dirty struct-obj)
```

Description:

The `test-dirty`, `clear-dirty` and `test-clear-dirty` functions comprise the interface for interacting with structure dirty flags.

Each structure instance has a dirty flag. When this flag is set, the structure instance is said to be dirty, otherwise it is said to be clean. A newly created structure is dirty. A structure remains dirty until its dirty flag is explicitly reset. If a structure is clean, and one of its instance slots is overwritten with a new value, it becomes dirty.

The `test-dirty` function returns the dirty flag of *`struct-obj`*: `t` if *`struct-obj`* is dirty, otherwise `nil`.

The `clear-dirty` function clears the dirty flag of *`struct-obj`* and returns *`struct-obj`* itself.

The `test-clear-dirty` flag combines these operations: it makes a note of the dirty flag of *`struct-obj`* and clears it. Then it returns the noted value, `t` or `nil`.

9.20.31 Function `structp`

Syntax:

```
(structp obj)
```

Description:

The `structp` function returns `t` if *`obj`* is a structure, otherwise it returns `nil`.

9.20.32 Function `struct-type`

Syntax:

```
(struct-type struct-obj)
```

Description:

The `struct-type` function returns the structure type object which represents the type of the structure object instance *`struct-obj`*.

9.20.33 Function `clear-struct`

Syntax:

```
(clear-struct struct-obj [value])
```

Description:

The `clear-struct` replaces all instance slots of *`struct-obj`* with *`value`*, which defaults to `nil` if omitted.

Note that finalizers are not executed prior to replacing the slot values.

9.20.34 Function `reset-struct`

Syntax:

```
(reset-struct struct-obj)
```

Description:

The `reset-struct` function reinitializes the structure object *`struct-obj`* as if it were being newly created. First, all the slots are set to `nil` as if by the `clear-struct` function. Then the slots are initialized by invoking the initialization functions, in order of the supertype ancestry, just as would be done for a new structure object created by `make-struct` with an empty *`slot-init-plist`* and no boa arguments.

Note that finalizers registered against *`struct-obj`* are not invoked prior to the reset operation, and remain registered.

If the structure has state which is cleaned up by finalizers, it is advisable to invoke them using `call-finalizers` prior to using `reset-struct`, or to take other measures to deal with the situation.

If the structure specifies `:fini` handlers, then the reinitialization will cause these to registered, just like when a new object it constructed. Thus if `call-finalizers` is not used prior to `reset-struct`, this will result in the existence of duplicate registrations of the finalization functions.

Finalizers registered against *`struct-obj`* **are** invoked if an exception is thrown during the reinitialization, just like when a new structure is being constructed.

9.20.35 Function `replace-struct`

Syntax:

```
(replace-struct target-obj source-obj)
```

Description:

The `replace-struct` function causes *`target-obj`* to take on the attributes of *`source-obj`* without changing its identity.

The type of `target-obj` is changed to that of `source-obj`.

All instance slots of `target-obj` are discarded, and it is given new slots, which are copies of the instance slots of `source-obj`.

Because of the type change, `target-obj` implicitly loses all of its original static slots, and acquires those of `source obj`.

Note that finalizers registered against *`target-obj`* are not invoked, and remain registered. If *`target-obj`* has state which is cleaned up by finalizers, it is advisable to invoke them using `call-finalizers` prior to using `replace-struct`, or to take other measures to handle the situation.

If the *`target-obj`* and *`source-obj`* arguments are the same object, `replace-struct` has no effect.

The return value is *`target-obj`*.

9.20.36 Function `method`

Syntax:

```
(method struct-obj slot-name curried-arg*)
```

Description:

The `method` function retrieves a function *m* from a structure's slot and returns a new function which binds that function's left argument. If *curried-arg* arguments are present, then they are also stored in the returned function. These are the *curried arguments*.

The *struct-obj* argument must be a structure, and *slot-name* must be a symbol denoting a slot in that structure. The slot must hold a function of at least one argument.

The function *f* which `method` function returns, when invoked, calls the function *m* previously retrieved from the object's slot, passing to that function *struct-obj* as the leftmost argument, followed by the curried arguments, followed by all of *f*'s own arguments.

Note: the `meth` macro is an alternative interface which is suitable if the slot name isn't a computed value.

9.20.37 Function `super-method`

Syntax:

```
(super-method struct-obj slot-name)
```

Description:

The `super-method` function retrieves a function from a static slot belonging to one of the direct supertypes of the structure type of *struct-obj*.

It then returns a function which binds that function's left argument to the structure.

The *struct-obj* argument must be a structure which has at least one supertype, and *slot-name* must be a symbol denoting a static slot in one of those supertypes. The slot must hold a function of at least one argument. The supertypes are searched from left to right for a static slot named *slot-name*; when the first such slot is found, its value is used.

The `super-method` function returns a function which, when invoked, calls the function previously retrieved from the supertype's static slot, passing to that function *struct-obj* as the leftmost argument, followed by the function's own arguments.

9.20.38 Function `umethod`

Syntax:

```
(umethod slot-name curried-arg*)
```

Description:

The `umethod` returns a function which represents the set of all methods named by the slot *slot-name* in all structure types, including ones not yet defined. The *slot-name* argument must be a symbol.

If one or more *curried-arg* argument are present, these values represent the *curried arguments* which are stored in the function object which is returned.

This returned function must be called with at least one argument. Its leftmost argument must be an object of structure type, which has a slot named *slot-name*. The function will retrieve the value of the slot from that object, expecting it to be a function, and calls it, passing to it the following arguments: the object itself; all of the curried arguments, if any; and all of its remaining arguments.

Note: the `umethod` name stands for "unbound method". Unlike the `method` function, `umethod` doesn't return a method whose leftmost argument is already bound to an object; the binding occurs at call time.

9.20.39 Function `uslot`

Syntax:

```
(uslot slot-name)
```

Description:

The `uslot` returns a function which represents all slots named *`slot-name`* in all structure types, including ones not yet defined. The *`slot-name`* argument must be a symbol.

The returned function must be called with exactly one argument. The argument must be a structure which has a slot named *`slot-name`*. The function will retrieve the value of the slot from that object and return it.

Note: the `uslot` name stands for "unbound slot". The returned function isn't bound to a particular object. The binding of `slot-name` to a slot in the structure object occurs when the function is called.

9.20.40 Function `slots`

Syntax:

```
(slots type)
```

Description:

The `slots` function returns a list of all of the slots of struct type *`type`*.

The *`type`* argument must be a structure type, or else a symbol which names a structure type.

9.20.41 Function `slotp`

Syntax:

```
(slotp type name)
```

Description:

The `slotp` function returns `t` if name *`name`* is a symbol which names a slot in the structure type *`type`*. Otherwise it returns `nil`.

The *`type`* argument must be a structure type, or else a symbol which names a structure type.

9.20.42 Function `static-slot-p`

Syntax:

```
(static-slot-p type name)
```

Description:

The `static-slot-p` function returns `t` if name *`name`* is a symbol which names a slot in the structure type *`type`*, and if that slot is a static slot. Otherwise it returns `nil`.

The *`type`* argument must be a structure type, or else a symbol which names a structure type.

9.20.43 Function `static-slot`

Syntax:

```
(static-slot type name)
```

Description:

The `static-slot` function retrieves the value of the static slot named by symbol *name* of the structure type *type*.

The *type* argument must be a structure type or a symbol which names a structure type, and *name* must be a static slot of this type.

9.20.44 Function `static-slot-set`

Syntax:

```
(static-slot-set type name new-value)
```

Description:

The `static-slot-set` function stores *new-value* into the static slot named by symbol *name* of the structure type *type*.

It returns *new-value*.

The *type* argument must be a structure type or the name of a structure type, and *name* must be a static slot of this type.

9.20.45 Function `static-slot-ensure`

Syntax:

```
(static-slot-ensure type name new-value [no-error-p])
```

Description:

The `static-slot-ensure` ensures, if possible, that the struct type *type*, as well as possibly one or more struct types derived from it, have a static slot called *name*, that this slot is not shared with a supertype, and that the value stored in it is *new-value*.

Note: this function supports the redefinition of methods, as the implementation underlying the `defmeth` macro; its semantics is designed to harmonize with expected behaviors in that usage.

The function operates as follows.

If *type* itself already has an instance slot called *name* then an error is thrown, and the function has no effect, unless a true argument is specified for the *no-error-p* Boolean parameter. In that case, in the same situation, the function has no effect and simply returns *new-value*.

If *type* already has a non-inherited static slot called *name* then this slot is overwritten with *new-value* and the function returns *new-value*. Types derived from *type* may also have this slot, via inheritance; consequently, its value changes in those types also.

If *type* already has an inherited static slot called *name* then its inheritance is severed; the slot is converted to a non-inherited static slot of *type* and initialized with *new-value*. Then all struct types derived from *type* are scanned. In each such type, if the original inherited static slot is found, it is replaced with the same newly converted static slot that was just introduced into *type*, so that all these types now inherit this new slot from *type* rather than the original slot from some

supertype of *type*. These types all share a single instance of the slot with *type*, but not with supertypes of *type*.

In the remaining case, *type* has no slot called *name*. The slot is added as a static slot to *type*. Then it is added to every struct type derived from *type* which does not already have a slot by that name, as if by inheritance. That is to say, types to which this slot is introduced share a single instance of that slot. The value of the new slot is *new-value*, which is also returned from the function. Any subtypes of *type* which already have a slot called *name* are ignored, as are their subtypes.

9.20.46 Function `static-slot-home`

Syntax:

```
(static-slot-home type name)
```

Description:

The `static-slot-home` method determines which structure type actually defines the static slot *name* present in struct type *type*.

If *type* isn't a struct type, or the name of a struct type, the function throws an error. Likewise, if *name* isn't a static slot of *type*.

If *name* is a static slot of *type* then the function returns a struct type name symbol which is either then name of *type* itself, if the slot is defined specifically for *type* or else the most distant ancestor of *type* from which the slot is inherited.

9.20.47 Function `call-super-method`

Syntax:

```
(call-super-method struct-obj name argument*)
```

Description:

The `call-super-method` function is deprecated. Solutions involving `call-super-method` should be reworked in terms of `call-super-fun`.

The `call-super-method` retrieves the function stored in the static slot *name* of one of the direct supertypes of *struct-obj* and invokes it, passing to that function *struct-obj* as the leftmost argument, followed by the given *argument*s, if any.

The *struct-obj* argument must be of structure type. Moreover, that structure type must be derived from one or more supertypes, and *name* must name a static slot available from at least one of those supertypes. The supertypes are searched left to right in search of this slot.

The object retrieved from that static slot must be callable as a function, and accept the arguments.

Note that it is not correct for a method that is defined against a particular type to use `call-super-method` to call the same method (or any other method) in the supertype of that particular type. This is because `call-super-method` refers to the type of the object instance *struct-obj*, not to the type against which the calling method is defined.

9.20.48 Function `call-super-fun`

Syntax:

```
(call-super-fun type name argument*)
```

Description:

The `call-super-fun` retrieves the function stored in the slot *name* of one of the supertypes of *type* and invokes it, passing to that function the given *argument*s, if any.

The *type* argument must be a structure type. Moreover, that structure type must be derived from one or more supertypes, and *name* must name a static slot available from at least one of those supertypes. The supertypes are searched left to right in search of this slot.

The object retrieved from that static slot must be callable as a function, and accept the arguments.

Example:

Print a message and call supertype method:

```
(defstruct base nil)

(defstruct derived base)

(defmeth base fun (obj arg)
  (format t "base fun method called with arg ~s\n" arg))

(defmeth derived fun (obj arg)
  (format t "derived fun method called with arg ~s\n" arg)
  (call-super-fun 'derived 'fun obj arg))

;; Interactive Listener:
1> (new derived).(fun 42)
derived fun method called with arg 42
base fun method called with arg 42
```

Note that a static method or function in any structure type can be invoked by using the `(meth ...)` name syntax in the first position of a compound form, as a function name. Thus, the above `derived fun` can also be written:

```
(defmeth derived fun (obj arg)
  (format t "derived fun method called with arg ~s\n" arg)
  ((meth base fun) obj arg))
```

9.20.49 Functions `struct-get-initfun` and `struct-get-postinitfun`

Syntax:

```
(struct-get-initfun type)
(struct-get-postinitfun type)
```

Description:

The `struct-get-initfun` and `struct-get-postinitfun` functions retrieve, respectively, a structure type's *initfun* and *postinitfun* functions. These are the functions which are initially configured in the call to `make-struct-type` via the *initfun* and *postinit-fun* arguments.

Either one may be `nil`, indicating that the type has no *initfun* or *postinitfun*.

9.20.50 Functions `struct-set-initfun` and `struct-set-postinitfun`

Syntax:

```
(struct-set-initfun type function)
(struct-set-postinitfun type function)
```

Description:

The `struct-set-initfun` and `struct-set-postinitfun` functions overwrite, respectively, a structure type's *initfun* and *postinitfun* functions. These are the functions which are initially configured in the call to `make-struct-type` via the *initfun* and *postinit-fun* arguments.

The *function* argument must either be `nil` or else a function which accepts one argument.

Note that *initfun* has the responsibility for all instance slot initializations. The `defstruct` syntax compiles the initializing expressions in the slot specifier syntax into statements which are placed into a function, which becomes the *initfun* of the struct type.

9.20.51 Macro `with-objects`

Syntax:

```
(with-objects ({(sym init-form)}*) body-form*)
```

Description:

The `with-objects` macro provides a binding construct similar to `let*`.

Each *sym* must be a symbol suitable for use as a variable name.

Each *init-form* is evaluated in sequence, and a binding is established for its corresponding *sym* which is initialized with the value of that form. The binding is visible to subsequent *init-form*s.

Additionally, the values of the *init-form*s are noted as they are produced. When the `with-objects` form terminates, by any means, the `call-finalizers` function is invoked on each value which was returned by an *init-form* and had been noted. These calls are performed in the reverse order relative to the original evaluation of the forms.

After the variables are established and initialized, the *body-form*s are evaluated in the scope of the variables. The value of the last form is returned, or else `nil` if there are no forms. The invocations of `call-finalizers` take place just before the value of the last form is returned.

9.20.52 Macro `define-struct-clause`

Syntax:

```
(define-struct-clause keyword params [body-form]*)
```

Description:

The `define-struct-clause` macro makes available a new, application-defined `defstruct` clause. The clause is named by *keyword*, which must be a keyword symbol, and is implemented as a macro transformation by the *params* and *body-form*s of the definition. The definition established by `define-struct-clause` is called a *struct clause macro*.

A struct clause macro is invoked when `defstruct` syntax is processed which contains one or more clauses which are headed by the matching *keyword* symbol.

The *params* comprise a macro-style parameter list which must match the invoking clause, otherwise an error exception is thrown. When *params* successfully matches the clause parameters, the parameters are destructured into the parameters and the *body-form*s are evaluated in the scope of those parameters.

The *body-form*s must return a possibly list of `defstruct` clauses, not a single clause.

Each of the returned clauses is examined for the possibility that it may be a struct clause macro; if so, it is expanded.

The built-in clause keywords `:static`, `:instance`, `:function`, `:method`, `:init`, `:postinit`, `:fini` and `:postfini`. may not be used as the names of a struct clause macro; if any of these symbols is used as the *keyword* parameter of `define-struct-clause`, an error exception is thrown.

The return value of a `define-struct-clause` macro invocation is the *keyword* argument.

Examples:

```
;; Trivial struct clause macro which consumes any number of
;; arguments and produces no slots:

(define-struct-clause :nothing (. ignored-args))

;; Consequently, the following defines a struct with one slot, x:
;; The (:nothing ...) clause disappears by producing no clauses.

(defstruct foo ()
  (:nothing 1 2 3 beeblebrox)
  x)

;; struct clause macro called :multi which takes an initial value
;; and zero or more slot names. It produces instance slot definitions
;; which all use that same initial value.

(define-struct-clause :multi (init-val . names)
  (mapcar (lop list init-val) names))

;; define a struct with three slots initialized to zero:

(defstruct bar ()
  (:multi 0 a b c)) ;; expands to (a 0) (b 0) (c 0)

;; struct clause macro to define a slot along with a
;; get and set method.

(define-struct-clause :getset (slot getter setter : init-val)
  ^((,slot ,init-val)
    (:method ,getter (obj) obj.,slot)
    (:method ,setter (obj new) (set obj.,slot new))))

;; Example use:

(defstruct point ()
  (:getset x get-x set-x 0)
```

```
  (:getset y get-y set-y 0))

;; This has exactly the same effect as the following defstruct:

(defstruct point ()
  (x 0)
  (y 0)
  (:method get-x (obj) obj.x)
  (:method set-x (ob new) (set obj.x new))
  (:method get-y (obj) obj.y)
  (:method set-y (ob new) (set obj.y new)))
```

9.20.53 Struct clause macro `:delegate`

Syntax:

```
(:delegate name (param+) delegate-expr [target-name])
```

Description:

The :delegate struct clause macro provides a way to define a method which is implemented entirely by delegation to a different object. The name of the method is *name* and its parameter list is specified in the same way as in the *:method* clause. Instead of a method body, the `:delegate` clause has an expression *delegate-expr* and an optional *target-name* which defaults to *name*. The *delegate-expr* must be an expression which the delegate method can evaluate to produce a delegate object. The delegate method then passes its arguments to the target method, given by the *target-name* argument, invoked on the delegate object.

If the delegate method specifies an optional parameter without a default initializing expression, and that optional parameter is not given an argument value, it receives the colon symbol `:` as its argument. That value is passed on to the corresponding parameter of the delegate target method. Thus, if the target method has an optional parameter in that same parameter position, that colon symbol argument then has the effect of requesting the default value. If the target method has an ordinary parameter in that position, then the colon symbol is received as an ordinary argument value.

If the delegate method specifies an optional parameter with a default initializing expression, and that optional parameter is not given an argument value, then the expression is evaluated to produce a value for that parameter, in the usual manner, and that value is passed as an argument to the corresponding parameter of the delegate target. Thus, delegates are able to specify different optional argument defaulting from their targets.

A delegate may have an optional parameter in a position where the target has a required parameter and vice versa.

The three-element optional parameter expression, specifying a Boolean variable which indicates whether the optional parameter has been given an argument, is not supported by the `:delegate` clause, and is diagnosed.

If the delegate method has variadic parameters, they are passed on to the target after the fixed parameters.

Example:

Structure definitions:

```
(defstruct worker ()
  name
  (:method work (me)
    `worker @{me.name} works`)
  (:method relax (me : (min 15))
    `worker @{me.name} relaxes for @min min`))

;; "contractor" class has a sub ("subcontractor") slot
;; which is another contractor of the same type.
;; The subcontractor's own sub slot, however is going
;; to be a worker.

(defstruct contractor ()
  sub
  (:delegate work (me) me.sub.sub)
  (:delegate break (me : min) me.sub.sub relax))
```

The `contractor` structure's `work` and `break` methods delegate to the sub-subcontractor, which is going to be instantiated as a `worker` object. Note that the `break` method delegates to a differently named method `relax`.

```
;; The objects are set up as described above.
;; general contractor co has a co.sub subcontractor,
;; and co.sub.sub is a worker:

(defvar co (new contractor
                sub (new contractor
                         sub (new worker name "foo"))))

;; Call work method on general contractor:
;; this invokes co.sub.sub.(work) on the worker.

co.(work) -> "worker foo works"

;; Call break method on general contractor with
;; no argument. This causes co.sub.sub.(relax :)
;; to be invoked, triggering argument defaulting:

co.(break) -> "worker foo relaxes for 15 min"

;; Call break method with argument. This
;; invokes co.sub.sub.(relax 5), specifying a
;; value for the default argument:

co.(break 5) -> "worker foo relaxes for 5 min"
```

9.20.54 Struct clause macro `:mass-delegate`

Syntax:

```
(:mass-delegate self-var delegate-expr
                from-type [*] [method]*)
```

Description:

The :mass-delegate struct macro provides a way to define multiple methods which are

implemented as delegates to corresponding methods on another object. The implementation of `:mass-delegate` depends on the `:delegate` macro.

The *self-var* argument must be a bindable symbol. In each generated delegate method, this symbol will be the first argument. The purpose of this symbol is to enable the *delegate-expr* to refer to the delegating object.

The *delegate-expr* is an expression which is inserted into every method. Its evaluation is expected to produce the delegate object. This expression may reference *self-var* in order to retrieve or otherwise obtain the delegate from the delegating object.

The *from-type* argument is a symbol naming an existing structure type. If no such structure type has been defined, an error exception is thrown.

After the *from-type* argument, either zero or more slot names appear, optionally preceded by the `*` (asterisk) symbol.

If the `*` symbol is present, and isn't followed by any other symbols, it indicates that all methods from *from-type* are to be delegated. If symbols appear after the `*` then those specify exceptions: methods not to be delegated. No validation is performed on the exception list; it may specify nonexistent method names which have no effect.

If the `*` symbol is absent, then every *method* symbol specifies a method to be delegated. It is consequently expected to name a method of the *from-type*: a static slot which contains a function. If any *method* isn't a static slot of *from-type*, or not a static slot which contains a function, an error exception is thrown.

The `:mass-delegate` struct macro iterates over all of the methods of *from-type* that selected for delegation, and for each one it generates a `:delegate` macro clause based on the existing method's parameter list. For instance, the delegate for a method which has two required arguments and one optional will itself have two required arguments and one optional. Delegates are not simply wrapper functions which take any number of arguments and try to pass them to the target.

The generated `:delegate` clauses are then processed by that struct clause macro.

Note: composition with delegation is a useful alternative when multiple inheritance is not applicable or desired for various reasons. One such reason is that structures that would be used as multiple inheritance bases use the same symbols for certain slots, and the semantics of those slots conflict. Under inheritance, same-named slots coming from different bases become one slot,

Note: a particular *from-type* being nominated in the `:mass-delegate` clause doesn't mean that the specific methods of that type shall be called by the generated delegates. The methods that shall be called are those of the calculated delegate object selected by the *delegate-expr*. The *from-type* is used as a source of the argument info, and method existence validation. It is up to the application to ensure that the delegation using *from-type* makes sense with respect to the delegate object that is selected by the *delegate-expr*: for instance, by ensuring that this object is an instance of *from-type* or a subtype thereof.

Example:

```
(defstruct foo-api ()
  name
  (:method begin (me) ^(foo ,me.name begin))
  (:method increment (me delta) ^(foo ,me.name increment ,delta))
```

```
  (:method end (me) ^(foo ,me.name end)))

(defstruct bar-api ()
  name
  (:method open (me) ^(bar ,me.name open))
  (:method read (me buf) ^(bar ,me.name read ,buf))
  (:method write (me buf) ^(bar ,me.name write ,buf))
  (:method close (me) ^(bar ,me.name close)))

;; facade holds the two API objects by composition:

(defstruct facade ()
  (foo (new foo-api name "foo"))
  (bar (new bar-api name "bar"))

  ;; delegate foo-api style calls via me.foo
  (:mass-delegate me me.foo foo-api *)

  ;; delegate bar-api style calls via me.bar
  ;; exclude the write method.
  (:mass-delegate me me.bar bar-api * write))

;; instantiate facade as variable fa
(defvar fa (new facade)) -> fa

;; begin call on facade delegates through foo-api object.
fa.(begin) -> (foo "foo" begin)

fa.(increment) -> ;; error: too few arguments

fa.(increment 3) -> (foo "foo" increment 3)

fa.(open) -> (bar "bar" open)

fa.(write 4) -> ;; error: fa has no such method
```

9.20.55 Function `macroexpand-struct-clause`

Syntax:

```
(macroexpand-struct-clause clause [form])
```

Description:

If `clause` is a compound expression whose operator symbol was defined by `define-struct-clause` then `macroexpand-struct-clause` expands the clause and returns the expansion, which is a list of zero or more clauses. Otherwise, the function returns a one-element list containing the *`clause`* argument, as if by the `(list clause)` expression.

The *`form`* parameter, if present, is used for reporting errors. Note: clauses are usually expanded during the processing of a `defstruct` macro; in that situation, the entire unexpanded `defstruct` form serves the role,

Examples:

```
;; try to expand :delegate, using incorrect syntax.
```

```
(macroexpand-struct-clause '(:delegate x (a b)))
--> error "** source location n/a: nil: too few elements ..."

;; same, but with error reporting form.
(macroexpand-struct-clause '(:delegate x (a b)) '(abc xyz))
--> error: "** expr-1:1: abc: too few elements ..."

;; correct :delegate syntax
(macroexpand-struct-clause '(:delegate x (a b) a.foo))
--> ((:method x (a b) (qref a.foo (x b))))

;; not a defstruct macro clause
(macroexpand-struct-clause '(1 2 3))
-> ((1 2 3))
```

9.20.56 Special variable `*struct-clause-expander*`

Description:

The `*struct-clause-expander*` special variable holds the hash table of associations between keyword symbols and struct clause expander functions, defined by `define-struct-clause`.

If the expression `[*struct-clause-expander* :sym]` yields a function, then symbol `:sym` has a binding as a struct clause macro. If that expression yields `nil`, then there is no such binding.

The macro expanders in `*struct-clause-expander*` are two-parameter functions. The first parameter accepts the clause to be expanded. The second parameter accepts the `defstruct` form in which that clause is found; this is useful for error reporting.

An expander function returns a list of clauses, which may be any, possibly empty mixture of primary clauses accepted by `defstruct` and clause macros.

9.20.57 Macro `define-struct-prelude`

Syntax:

(define-struct-prelude *name struct-name-or-list clause**)

Description:

The `define-struct-prelude` macro defines a *prelude*. A prelude is a named entity which implicitly provides clauses to `defstruct` macro invocations. Preludes are processed during the macroexpansion of `defstruct`; prelude definitions have no effect on previously compiled `defstruct` forms loaded from a file.

A prelude has a *name* which must be a bindable symbol. The purpose of this name is that if multiple `define-struct-prelude` forms are evaluated which specify the same *name*, they replace each others' definition. Only the most recent prelude of a given *name* is retained; the previous definitions are overwritten.

The *struct-name-or-list* argument is either a symbol or a list of symbols, which are valid for use as structure names. The prelude being defined shall be applicable to each of the structures whose names are given by this argument.

The zero or more *clause* arguments give the clauses which comprise the prelude. In the future,

when a `defstruct` form is macroexpanded which targets any of the structures given by the *`struct-name-or-list`* argument, the specified clauses will be inserted into that definition, as if they appeared in the `defstruct` form literally.

Multiple preludes may be defined with different names, which each target the same structure. When the structure is defined, or redefined, it will receive all those preludes, in the order in which they were defined.

Example:

```
;; define init-fini-log prelude which targets fox and bear structs

(define-struct-prelude init-fini-log (fox bear)
  (:init (me) (put-line `@me created`))
  (:fini (me) (put-line `@me finalized`)))

;; The behavior is as if the following defstruct forms included
;; the above :init and :fini clauses

(defstruct fox ())

(defstruct bear ())

(with-object ((f (new fox))
              (b (new bear)))
  (put-line "inside with-object"))
```

Output:

```
#S(fox) created
#S(bear) created
inside with-object
#S(bear) finalized
#S(fox) finalized
```

9.21 Special Structure Functions

Special structure functions are user-defined methods or structure functions which are specially recognized by certain functions in **TXR Lisp**. They endow structure objects with the ability to participate in certain usage scenarios, or to participate in a customized way.

Special functions are required to bound to static slots, which is the case if the `defmeth` macro is used, or when methods or functions are defined using syntax inside a `defstruct` form. If a special function or method is defined as an instance slot, then the behavior of library functions which depend on this method is unspecified.

Special functions introduced below by the word "Method" receive an object instance as an argument. Their syntax is indicated using the same notation which may be used to invoke them, such as:

`object``.(function-name` *`arg`* `...)`

However, those introduced as "Function" do not operate on an instance. Their syntax is likewise indicated using the notation that may be used to invoke them:

`'['`*`object`*`.function-name` *`arg`* `...']'`

If such a invocation is actually used, the *object* instance only serves for identifying the struct type whose static slot `function-name` provides the function; *object* doesn't participate in the call. An object is not strictly required since the function can be called using

```
[(static-slot type 'function-name) arg ...]
```

which looks up the function in the struct *type* directly.

9.21.1 Method `equal`

Syntax:

```
object.(equal)
```

Description:

Normally, two struct values are not considered the same under the `equal` function unless they are the same object.

However, if the `equal` method is defined for a structure type, then instances of that structure type support *equality substitution*.

The `equal` method must not require any arguments other than *object*. Moreover, the method must never return `nil`.

When a struct which supports equality substitution is compared using `equal`, `less` or `greater`, its `equal` method is invoked, and the return value is used in place of that structure for the purposes of the comparison.

The same applies when an struct is hashed using the `hash-equal` function, or implicitly by an `:equal-hash` hash table.

Note: if an `equal` method is defined or redefined with different semantics for a struct type whose instances have already been inserted as keys in an `:equal-based` hash table, the behavior of subsequent insertion and lookup operations on that hash table becomes unspecified.

9.21.2 Method `print`

Syntax:

```
object.(print stream pretty-p)
```

Description:

If a method named by the symbol `print` is defined for a structure type, then it is used for printing instances of that type.

The *stream* argument specifies the output stream to which the printed representation is to be written.

The *pretty-p* argument is a Boolean flag indicating whether pretty-printing is requested. Its value may simply be passed to recursive calls to `print`, or used to select between `~s` or `~a` formatting if `format` is used.

The value returned by the `print` method is significant. If the special keyword symbol `:` (colon) is returned, then the system will print the object in the default way, as if no `print` method existed: it is understood that the method declined the responsibility for printing the object.

If any other value is returned, then it is understood that the method `print` method accepted the responsibility for printing the object, and the system consequently will generate into *`stream`* any output output pertaining to *`object`*'s representation.

9.21.3 Method `lambda`

Syntax:

```
object.(lambda arg*)
```

Description:

If a structure type provides a method called `lambda` then it can be used as a function.

This method can be called by name, using the syntax given in the above syntactic description.

However, the intended use is that it allows the structure instance itself to be used as a function. When arguments are applied to a structure object as if it were a function, this is erroneous, unless that object has a `lambda` method. In that case, the arguments are passed to the lambda method. The leftmost argument of the method is the structure instance itself.

That is to say, the following equivalences apply, except that `s` is evaluated only once:

```
(call s args ...)  <-->  s.(lambda args ...)

[s args ...]  <-->  [s.lambda s args ...]

(mapcar s list)  <-->  (mapcar (meth s lambda) list)
```

Note: a form such as `[s args ...]` where `s` is a structure can be treated as a place if the method `lambda-set` is also implemented.

9.21.4 Method `lambda-set`

Syntax:

```
object.(lambda-set arg* new-value)
```

Description:

The `lambda-set` method, in conjunction with a `lambda` method, allows structures to be used as place accessors. If structure `s` supports a *`lambda-set`* with four arguments, then the following use of the `dwim` operator is possible:

```
(set [s a b c d] v)
(set (dwim s a b c d) v) ;; precisely equivalently
```

This has an effect which can be described by the following code:

```
(progn
  s s.(lambda-set a b c d v)
  v)
```

except that `s` and `v` are evaluated only once, and `a` through `d` are evaluated using the Lisp-1 semantics due the `dwim` operator.

If a place-mutating operator is used on this form which requires the prior value, such as the `inc` macro, then the structure must support the `lambda` function also.

If `lambda` takes *n* arguments, then `lambda-set` should take *n+1* arguments. The first *n* arguments of these two methods are congruent; the extra rightmost argument of `lambda-set` is the new value to be stored into the place denoted by the prior arguments.

The return value of `lambda-set` is ignored.

Note: the `lambda-set` method is also used by the `rplaca` function, if no `rplaca` method exists.

Example

The following defines a structure with a single instance slot `hash` which holds a hash table, as well as `lambda` and `lambda-set` methods:

```
(defstruct hash-wrapper nil
  (hash (hash))

  (:method lambda (self key)
    [self.hash key])

  (:method lambda-set (self key new-val)
    (set [self.hash key] new-val) self))
```

An instance of this structure can now be used as follows:

```
(let ((s (new hash-wrapper)))
  (set [s "apple"] 3
       [s "orange] 4)
  [s "apple"]) -> 3
```

9.21.5 Method `length`

Syntax:

```
object.(length)
```

Description:

If a structure has `length` method, then it can be used as an argument to the `length` function.

Structures which implement the methods `lambda`, `lambda-set` and `length` can be treated as abstract vector-like sequences, because such structures support the `ref`, `refset` and `length` functions.

For instance, the `nreverse` function will operate on such objects.

Note: a structure which supports the `car` method also supports the `length` function, in a different way. Such a structure is treated by `length` as a list-like sequence, and its length is measured by walking the sequence with `cdr` operations. If a structure supports both `length` and `car`, preference is given to `length`, which is likely to be much more efficient.

9.21.6 Methods `car`, `cdr` and `nullify`

Syntax:

```
object.(car)
object.(cdr)
```

```
object.(nullify)
```

Description:

Structures may be treated as sequences if they define methods named by the symbols `car`, `cdr`, and `nullify`.

If a structure supports these methods, then these methods are used by the functions `car`, `cdr`, `nullify`, `empty` and various other sequence manipulating functions derived from them, when those functions are applied to that object.

An object which implements these three methods can be considered to represent a *list-like* abstract sequence.

The object's `car` method should return the first value in that abstract sequence, or else `nil` if that sequence is empty.

The object's `cdr` method should return an object denoting the remainder of the sequence, or else `nil` if the sequence is empty or contains only one value. This returned object can be of any type: it may be of the same structure type as that object, a different structure type, a list, or whatever else. If a non-sequence object is returned.

The `nullify` method should return `nil` if the object is considered to denote an empty sequence. Otherwise it should either return that object itself, or else return the sequence which that object represents.

9.21.7 Methods `rplaca` and `rplacd`

Syntax:

```
object.(rplaca new-car-value)
object.(rplacd new-cdr-value)
```

Description:

If a structure type defines the methods `rplaca` and `rplacd` then, respectively, the `rplaca` and `rplacd` functions will use these methods if they are applied to instances of that type.

That is to say, when the function call `(rplaca o v)` is evaluated, and *o* is a structure type, the function inquires whether *o* supports a `rplaca` method. If so, then, effectively, `o. (rplaca v)` is invoked. The return value of this method call is ignored; `rplaca` returns *o*. The analogous requirements apply to `rplacd` in relation to the `rplacd` method.

Note: if the `rplaca` method doesn't exist, the `rplaca` function falls back on trying to store *new-car-value* by means of the structure type's `lambda-set` method, using an index of zero. That is to say, if the type has no `rplaca` method, but does have a `lambda-set` method, then `o. (lambda-set 0 v)` is invoked.

9.21.8 Function `from-list`

Syntax:

```
'['object.from-list list']'
```

Description:

If a `from-list` structure function is defined for a structure type, it is called in certain situations with an argument which is a list object. The function's purpose is to construct a new instance of the structure type, derived from that list.

The purpose of this function is to allow sequence processing operations such as `mapcar` and `remove` to operate on a structure object as if it were a sequence, and return a transformed sequence of the same type. This is analogous to the way such functions can operate on a vector or string, and return a vector or string.

If a structure object behaves as a sequence thanks to providing `car`, `cdr` and `nullify` methods, but does not have a `from-list` function, then those sequence-processing operations which return a sequence will always return a plain list of items.

9.21.9 Function `derived`

Syntax:

```
'['object.derived supertype subtype']'
```

Description:

If a structure type supports a function called *`derived`*, this function is called whenever a new type is defined which names that type as its supertype.

The function is called with two arguments which are both struct types. The *`supertype`* argument gives the type that is being inherited from. The *`subtype`* gives the new type that is inheriting from *`supertype`*.

When a new structure type is defined, its list of immediate supertypes is considered. For each of those supertypes which defines the `derived` function, the function is invoked.

The function is not retroactively invoked. If it is defined for a structure type from which subtypes have already been derived, it is not invoked for those existing subtypes.

If *`derived`* directly inherits *`supertype`* more than once, it is not specified whether this function is called once, or multiple times.

Note: the *`supertype`* parameter exists because the `derived` function is itself inherited. If the same version of this function is shared by multiple structure types due to inheritance, this argument informs the function which of those types it is being invoked for.

9.21.10 Methods `iter-begin` and `iter-reset`

Syntax:

```
object.(iter-begin)
object.(iter-reset iter)
```

Description:

If an object supports the `iter-begin` method, it is considered iterable; the `iterable` function will return `t` if invoked on this object.

The responsibility of the `iter-begin` method is to return an iterator object: an object which supports certain special methods related to iteration, according to one of two protocols, described below.

The `iter-reset` method is optional. It is similar to `iter-begin` but takes an additional *`iter`* argument, an iterator object that was previously returned by the `iter-begin` method of the same *`object`*.

If `iter-reset` determines that *`iter`* can be reused for a new iteration, then it can suitably

mutate the state of *iter* and return it. Otherwise, it behaves like `iter-begin` and returns a new iterator.

There are two protocols for iteration: the fast protocol, and the canonical protocol. Both protocols require the iterator object returned by the `iter-begin` method to provide the methods `iter-item` and `iter-step`. If the iterator also provides the `iter-more` method, then the protocol which applies is the canonical protocol. If that method is absent, then the fast protocol is followed.

Under the fast protocol, the `iter-more` method does not exist and is not involved. The iterable object's `iter-begin` method must return `nil` if the abstract sequence is empty. If an iterator is returned, it is assumed that an object can be retrieved from the iterator by invoking its `iter-item` method. The iterator's `iter-next` method should return `nil` if there are no more objects in the abstract sequence, or else it should return an iterator that obeys the fast protocol (possibly itself).

Under the canonical protocol, the iterator implements the `iter-more` function. The iterable object's `iter-begin` always returns an iterator object. The iterator object's `iter-more` method is always invoked to determine whether another item is available from the sequence. The iterator object's `iter-step` method is expected to return an iterator object which conforms to the canonical protocol.

9.21.11 Method `iter-item`

Syntax:

object`.(iter-item)`

Description:

The `iter-item` method is invoked on an iterator *object* to retrieve the next item in the sequence.

Under the fast protocol, it is assumed that if *object* was returned by an iterable object's `iter-begin` method, or by an iterator's `iter-step` method, that an item is available. This method will be unconditionally invoked.

Under the canonical protocol for iteration, the `iter-more` method will be invoked on *object* first. If that method yields true, then `iter-item` is expected to yield the next available item in the sequence.

Note: calls to the `iter-item` function, with *object* as its argument, invoke the `iter-item` method. It is possible for an application to call `iter-item` through this function or directly as a method call without first calling `iter-more`. No iteration mechanism in the **TXR Lisp** standard library behaves this way. If the iterator *object* has no more items available and `iter-more` is invoked anyway, no requirements apply to its behavior or return value.

9.21.12 Method `iter-step`

Syntax:

object`.(iter-step)`

Description:

The `iter-step` method is invoked on an iterator object to produce an iterator object for the remainder of the sequence, excluding the current item.

Under the fast iteration protocol, this method returns `nil` if there are no more items in the

sequence.

Under the canonical iteration protocol, this method always returns an iterator object. If no items remain in the sequence, then that iterator object's `iter-more` method returns `nil`. Furthermore, under this protocol, `iter-step` is not called if `iter-more` returns `nil`.

Note: calls to the `iter-step` function, with *`object`* as its argument, invoke the `iter-step` method. It is possible for an application to call `iter-step` through this function or directly as a method call without first calling `iter-more`. No iteration mechanism in the **TXR Lisp** standard library behaves this way. If the iterator *`object`* has no more items available and `iter-step` is invoked anyway, no requirements apply to its behavior or return value.

9.21.13 Method `iter-more`

Syntax:

```
object.(iter-more)
```

Description:

If an iterator *`object`* returned by `iter-begin` supports the `iter-more` method, then the canonical iteration protocol applies to that iteration session. All subsequent iterators that are involved in the iteration are assumed to conform to the protocol and should implement the `iter-more` method also. The behavior is unspecified otherwise.

The `iter-more` method is used to interrogate an iterator whether more unvisited items remain in the sequence. This method does not advance the iteration, and does not change the state of the iterator. It is idempotent: if it is called multiple times without any intervening call to any other method, it yields the same value.

If an iterator does not implement the `iter-more` method, then if the `iter-more` function is applied to that iterator, it unconditionally returns `t`.

9.22 Sequence Manipulation

Functions in this category uniformly manipulate abstract sequences. Lists, strings and vectors are sequences.

Structure objects can behave like sequences, either list-like or vector-like sequences, if they have certain methods: see the previous section Special Structure Functions.

Moreover, hash tables behave like sequences of key-value entries represented by `cons` pairs. Not all sequence-processing functions accept hash-table sequences.

Additionally, some sequence-processing functions work not only with sequences but with all iterable objects: objects that can be used as arguments to the `iter-begin` function. Such arguments are called *`iterable`* rather than *`sequence`*, possibly abbreviated to *`iter`* with or without a numeric suffix. Hash tables are always supported if they appear as *`iterable`* arguments.

9.22.1 Function `seqp`

Syntax:

```
(seqp object)
```

Description:

The function `seqp` returns `t` if *`object`* is a sequence, otherwise `nil`.

Lists, vectors and strings are sequences. The object `nil` denotes the empty list and so is a sequence.

Objects of type `buf` and `carray` are sequences, as are hash tables.

Structures which implement the `length` or `car` methods are considered sequences.

No other objects are sequences. However, future revisions of the language may specify additional objects that are sequences.

9.22.2 Function `iterable`

Syntax:

```
(iterable object)
```

Description:

The `iterable` function returns `t` if *`object`* is iterable, otherwise `nil`.

If *`object`* is a sequence according to the `seqp` function, then it is iterable.

If *`object`* is a structure which supports the `iter-begin` method, then it is iterable.

Additional objects that are not sequences are also iterable: numeric or character ranges, and numbers. Future revisions of the language may specify additional iterable objects.

9.22.3 Function `make-like`

Syntax:

```
(make-like list object)
```

Description:

The *`list`* argument must be a list. If *`object`* is a sequence type, then *`list`* is converted to the same type of sequence and returned. Otherwise the original *`list`* is returned.

Conversion is supported to string and vector type.

Conversion to a structure type is possible for structures. If *`object`* is an object of a structure type which has a static function `from-list`, then `make-like` calls that function, passing to it, and the resulting value is returned. *`list`* and returns whatever value that function returns.

If *`object`* is a `carray`, then *`list`* is passed to the `carray-list` function, and the resulting value is returned. The second argument in the `carray-list` call is the element type taken from *`object`*. The third argument is `nil`, indicating that the resulting `carray` is not to be null terminated.

The *`object`* may be an iterator returned by `iter-begin`. In this situation, if that object makes the original sequence available, then `make-like` takes that sequence in place of *`object`*,

Note: the `make-like` function is a helper which supports the development of unoptimized versions of a generic function that accepts any type of sequence as input, and produces a sequence of

the same type as output. The implementation of such a function can internally accumulate a list, and then convert the resulting list to the same type as an input value by using `make-like`.

9.22.4 Functions `list-seq`, `vec-seq` and `str-seq`

Syntax:

```
(list-seq iterable)
(vec-seq iterable)
(str-seq iterable)
```

Description:

The `list-seq`, `vec-seq` and `str-seq` functions convert an iterable object of any type into a list, vector or string, respectively.

The list returned by `list-seq` is lazy.

The `list-seq` and `vec-seq` iterate the items of *`iterable`* and accumulate these items into a new list or vector.

The `str-seq` similarly iterates the items of *`iterable`*, requiring them to be a mixture of characters and strings.

9.22.5 Functions `length` and `len`

Syntax:

```
(length iterable)
(len iterable)
```

Description:

The `length` function returns the number of items contained in *`iterable`*.

The `len` function is a synonym of `length`.

An attempt to calculate the length of infinite lazy lists will not terminate. Iterable objects representing infinite ranges, such as integers and characters are invalid arguments.

9.22.6 Function `empty`

Syntax:

```
(empty iterable)
```

Description:

If *`iterable`* is a suitable argument for the `length` function, then the `empty` Returns `t` if `(length iterable)` is zero, otherwise `nil`.

The `empty` function also supports certain objects not suitable as arguments for `length`.

An infinite lazy list is not empty, and so `empty` returns `nil` for such an object.

The function also returns `nil` for iterable objects representing nonempty spaces, even if those spaces are infinite. For instance `(empty 0)` yields `nil` because the set of integers beginning with 0 isn't empty.

9.22.7 Function `nullify`

Syntax:

```
(nullify iterable)
```

Description:

The `nullify` function returns `nil` if *`iterable`* denotes an empty sequence. Otherwise, if *`iterable`* is not an empty sequence, or isn't a sequence, then *`iterable`* itself is returned.

If *`iterable`* is a structure object which supports the `nullify` method, then that method is called. If it returns `nil` then `nil` is returned. If the `nullify` method returns a substitute object other than the *`iterable`* object itself, then `nullify` is invoked on that returned substitute object.

Note: the `nullify` function is a helper to support unoptimized generic traversal of sequences. Thanks to the generalized behavior of `cdr`, non-list sequences can be traversed using `cdr`, similarly to proper lists, by checking for `cdr` returning the terminating value `nil`. However, empty non-list sequences are handled incorrectly because since they are not the `nil` object, they look nonempty under this paradigm of traversal. The `nullify` function provides a correction: if the input sequence is filtered through `nullify` then the subsequent list-like iteration works correctly.

Examples:

```
;; Incorrect for empty strings:

(defun print-chars (string)
  (while string
    (prinl (pop string))))

;; Corrected with nullify:

(defun print-chars (string)
  (let ((s (nullify string)))
    (while s
      (prinl (pop s)))))
```

Note: optimized generic iteration is available in the form of iteration based on `iter-begin` rather than `car`/`cdr` and `nullify`.

Examples:

```
;; Efficient with iterators,
;; at the cost of verbosity:

(defun print-chars (string)
  (let ((i (iter-begin string)))
    (while (iter-more i)
      (prinl (iter-item s))
      (set s (iter-step s)))))

;; Using mapping function built on iterators:

(defun print-chars (string)
  [mapdo prinl string])
```

9.22.8 Accessor `sub`

Syntax:

```
(sub sequence [from [to]])
(set (sub sequence [from [to]]) new-val)
```

Description:

The `sub` function extracts a slice from input sequence *sequence*. The slice is a sequence of the same type as *sequence*.

If the *from* argument is omitted, it defaults to `0`. If the *to* parameter is omitted, it defaults to `t`. Thus `(sub a)` means `(sub a 0 t)`.

The following semantic equivalence exists between a call to the `sub` function and the DWIM-bracket syntax, except that `sub` is an ordinary function call form, which doesn't apply the Lisp-1 evaluation semantics to its arguments:

```
;; from is not a list
(sub seq from to) <--> [seq from..to]
```

The description of the `dwim` operator—in particular, the section on Range Indexing—explains the semantics of the range specification.

The output sequence may share structure with the input sequence.

If *sequence* is a `carray` object, then the function behaves like `carray-sub`.

If *sequence* is a `buf` object, then the function behaves like `sub-buf`.

If *sequence* is a `tree` object, then the function behaves like `sub-tree`. Note: because `sub-tree` is not an accessor, assigning to the `sub` syntax in this case will produce an error.

The *sequence* argument may also be any other object type that is suitable as input to the `iter-begin` function. In this situation, assigning to `sub` syntax produces an error. The behavior is complex. In cases where the *from* and *to* arguments imply that a suffix of *sequence* is required, an iterator may be returned which traverses the suffix of the sequence. In other cases, a list of the elements selected by `sub` is returned.

If *sequence* is a structure, it must support the `lambda` method. The `sub` operation is transformed into a call to the `lambda` method according to the following equivalence:

```
(sub o from to) <--> o.(lambda (rcons from to))
(sub o : to)    <--> o.(lambda (rcons : to))
(sub o from)    <--> o.(lambda (rcons from :))
(sub o)         <--> o.(lambda (rcons : :))
```

That is to say, the *from* and `to` arguments are converted to range object. If either argument is missing, the `:` (colon) keyword symbol is used for the corresponding element of the range.

When a `sub` form is used as a syntactic place, that place denotes a slice of *seq*. The *seq* argument must be itself be syntactic place, because receives a new value, which may be different from its original value in cases when *seq* is a list.

Overwriting that slice is equivalent to using the `replace` function. The following equivalences give the semantics, except that `x`, `a`, `b` and `v` are evaluated only once, in left-to-right order:

```
(set (sub x a b) v)  <-->  (progn (set x (replace x v a b))
                                  v)

(del (sub x a b))    <-->  (prog1 (sub x a b)
                                  (set x (replace x nil a b)))
```

Note that the value of `x` is overwritten with the value returned by `replace`. If `x` is a vector or string, then the return value of `replace` is just `x`: the identity of the object doesn't change under mutation. However, if `x` is a list, its identity changes when items are added to or removed from the front of the list, and in those cases `replace` will return a value different from its first argument. Similarly, if `x` is an object with a `lambda-set` method, that method's return value becomes the return value of `replace` and must be taken into account.

9.22.9 Function `replace`

Syntax:

```
(replace sequence replacement-sequence [from [to]])
(replace sequence replacement-sequence index-list)
```

Description:

The *`replace`* function modifies *`sequence`* in the ways described below.

The operation is destructive: it may work "in place" by modifying the original sequence. The caller should retain the return value and stop relying on the original input sequence.

The return value of `replace` is the modified version of *`sequence`*. This may be the same object as *`sequence`* or it may be a newly allocated object.

Note that the form:

```
(set seq (replace seq new fr to))
```

has the same effect on the variable `seq` as the form:

```
(set [seq fr..to] new)
```

except that the former `set` form returns the entire modified sequence, whereas the latter returns the value of the `new` argument.

The `replace` function has two invocation styles, distinguished by the type of the third argument. If the third argument is a sequence, then it is deemed to be the *`index-list`* parameter of the second form. Otherwise, if the third argument is missing, or is not a list, then it is deemed to be the *`from`* argument of the first form.

The first form of the replace function replaces a contiguous subsequence of the *`sequence`* with *`replacement-sequence`*. The replaced subsequence may be empty, in which case an insertion is performed. If *`replacement-sequence`* is empty (for example, the empty list `nil`), then a deletion is performed.

If the *`from`* and *`to`* arguments are omitted, their values default to `0` and `t` respectively.

The description of the dwim operator—in particular, the section on Range Indexing—explains the semantics of the range specification.

The second form of the replace function replaces a subsequence of elements from *sequence* given by *index-list*, with their counterparts from *replacement-sequence*. This form of the replace function does not insert or delete; it simply overwrites elements. If *replacement-sequence* and *index-list* are of different lengths, then the shorter of the two determines the maximum number of elements which are overwritten. Whenever a negative value occurs in *index-list* the length of *sequence* is added to that value. Furthermore, similar restrictions apply on *index-list* as under the select function. Namely, the replacement stops when an index value in *index-list* is encountered which is out of range for *sequence*. furthermore, if *sequence* is a list, then *index-list* must be monotonically increasing, after consideration of the displacement of negative values.

If *replacement-sequence* shares storage with the target range of *sequence*, or, in the case when that range is resized by the `replace` operation, shares storage with any portion of *sequence* above that range, then the effect of `replace` on either object is unspecified.

If *sequence* is a `carray` object, then `replace` behaves like `carray-replace`.

If *sequence* is a `buf` object, then `replace` behaves like `buf-replace`.

If *sequence* is a structure, then the structure must support the `lambda-set` method. The `replace` operation is translated into a call of the `lambda-set` method according to the following equivalences:

```
(replace o items from to)
<--> o.(lambda-set (rcons from to) items)

(replace o items index-list)
<--> o.(lambda-set index-list items)
```

Thus, the *from* and *to* arguments are converted to single range object, whereas an *index-list* is passed as-is. It is an error if the `from` argument is a sequence, indicating an *index-list*, and a `to` argument is also given; the situation is diagnosed. If either `from` or `to` are omitted, the range object contains the `:` (colon) keyword symbol in the corresponding place:

```
(replace o items from)
<--> o.(lambda-set (rcons from :) items)

(replace o items : to)
<--> o.(lambda-set (rcons : to) items)

(replace o items)
<--> o.(lambda-set (rcons : :) items)
```

It is the responsibility of the object's `lambda-set` method to implement semantics consistent with the description of `replace`.

9.22.10 Function `take`

Syntax:

```
(take count sequence)
```

Description:

The `take` function returns *sequence* with all except the first *count* items removed.

If *sequence* is a list, then take returns a lazy list which produces the first *count* items of sequence.

For other kinds of sequences, including lazy strings, take works eagerly.

If *count* exceeds the length of *sequence* then a sequence is returned which has all the items. This object may be *sequence* itself, or a copy.

If *count* is negative, it is treated as zero.

9.22.11 Functions take-while and take-until

Syntax:

```
(take-while predfun sequence [keyfun])
(take-until predfun sequence [keyfun])
```

Description:

The take-while and take-until functions return a prefix of *sequence* whose items satisfy certain conditions.

The take-while function returns the longest prefix of *sequence* whose elements, accessed through *keyfun* satisfy the function *predfun*.

The *keyfun* argument defaults to the identity function: the elements of *sequence* are examined themselves.

The take-until function returns the longest prefix of *sequence* which consists of elements, accessed through *keyfun*, that do **not** satisfy *predfun* followed by an element which does satisfy *predfun*. If *sequence* has no such prefix, then an empty sequence is returned of the same kind as *sequence*.

If *sequence* is a list, then these functions return a lazy list.

9.22.12 Function drop

Syntax:

```
(drop count sequence)
```

Description:

The drop function returns *sequence* with the first *count* items removed.

If *count* is negative, it is treated as zero.

If *count* is zero, then *sequence* is returned.

If *count* exceeds the length of *sequence* then an empty sequence is returned of the same kind as *sequence*.

9.22.13 Functions drop-while and drop-until

Syntax:

```
(drop-while predfun sequence [keyfun])
(drop-until predfun sequence [keyfun])
```

Description:

The `drop-while` and `drop-until` functions return *sequence* with a prefix of that sequence removed, according to conditions involving *predfun* and *keyfun*.

The `drop-while` function removes the longest prefix of *sequence* whose elements, accessed through *keyfun* satisfy the function *predfun*, and returns the remaining sequence.

The *keyfun* argument defaults to the identity function: the elements of *sequence* are examined themselves.

The `drop-until` function removes the longest prefix of *sequence* which consists of elements, accessed through *keyfun*, that do **not** satisfy *predfun* followed by an element which does satisfy *predfun*. A sequence of the remaining elements is returned.

If *sequence* has no such prefix, then a sequence same as *sequence* is returned, which may be *sequence* itself or a copy.

9.22.14 Accessor `last`

Syntax:

```
(last sequence [num])
(set (last sequence [num]) new-value)
```

Description:

The *last* function returns a subsequence of *sequence* consisting of the last *num* of its elements, where *num* defaults to 1.

If *num* is zero or negative, then an empty sequence is returned. If *num* is positive, and greater than or equal to the length of sequence, then sequence *sequence* is returned.

If a `last` form is used as a place, then `sequence` must be a place. The following equivalence gives the semantics of assignment to a `last`:

```
(set (last x n) v)   <-->  (set (sub x (- (max n 0)) t) v)
```

A `last` place is deletable. The semantics of deletion may be understood in terms of the following equivalence:

```
(del (last x n))   <-->  (del (sub x (- (max n 0)) t))
```

9.22.15 Accessor `butlast`

Syntax:

```
(butlast sequence [num])
(set (butlast sequence [num]) new-value)
```

Description:

The `butlast` function returns the prefix of *sequence* consisting of a copy of it, with the last *num* items removed.

The parameter *num* defaults to 1 if an argument is omitted.

If *sequence* is empty, an empty sequence is returned.

If *num* is zero or negative, then *sequence* is returned.

If *num* is positive, and meets or exceeds the length of *sequence*, then an empty sequence is returned.

If a `butlast` form is used as a place, then *sequence* must itself be a place. The following equivalence gives the semantics of assignment to a `last`:

```
(set (butlast x n) v)   <-->  (set (sub x 0 (- (max n 0))) v)
```

A `butlast` place is deletable. The semantics of deletion may be understood in terms of the following equivalence:

```
(del (last x n))   <-->  (del (sub x 0 (- (max n 0))))
```

Note: the **TXR Lisp** `take` function also computes the prefix of a list; however, it counts items from the beginning, and provides lazy semantics which allow it to work with infinite lists.

See also: the `butlastn` accessor, which operates on lists. That function has useful semantics for improper lists and treats an atom as the terminator of a zero-length improper list.

Dialect Note: a destructive function similar to Common Lisp's `nbutlast` isn't provided. Assignment to a `butlast` form is destructive; Common Lisp doesn't support `butlast` as a place.

9.22.16 Function `ldiff`

Syntax:

```
(ldiff sequence tail-sequence)
```

Description:

The `ldiff` function is a somewhat generalized version of the same-named classic Lisp function found in traditional Lisp dialects.

The `ldiff` function supports the original `ldiff` semantics when both inputs are lists. It determines whether the *tail-sequence* list is a structural suffix of *sequence*, which is to say: is *tail-sequence* one of the `cons` cells which comprise *sequence*? If so, then a list is returned consisting of all the items of *sequence* before *tail-sequence*: a copy of *sequence* with the *tail-sequence* part removed, and replaced by the `nil` terminator. If *tail-sequence* is `nil` or the lists are unrelated, then *sequence* is returned.

The **TXR Lisp** `ldiff` function supports the following additional semantics.

1. The basic description of `ldiff` is extended to work with list-like sequences, not merely lists; that is to say, objects which support the `car` method.
2. If *sequence* is any kind of sequence, and *tail-sequence* is any kind of empty sequence, then *sequence* is returned.
3. If either argument is an atom that is not a sequence, `ldiff` returns *sequence*.
4. If *sequence* is a list-like sequence, and *tail-sequence* isn't, then the terminating atom of *sequence* is determined. This atom is compared using `equal` to the *tail-sequence* object. If they are equal, then a proper list is returned containing the items of *sequence* excluding the terminating atom.

5. If both arguments are vector-like sequences, then `ldiff` determines whether *`sequence`* has a suffix which is `equal` to *`tail-sequence`*. If this is the case, then a sequence is returned, of the same kind as *`sequence`*, consisting of the items of *`sequence`* before that suffix. If *`tail-sequence`* is not `equal` to a suffix of *`sequence`*, then *`sequence`* is returned.
6. In all other cases, *`sequence`* and *`tail-sequence`* are compared with `equal`. If the comparison is true, `nil` is returned, otherwise *`sequence`* is returned.

Examples:

```
;;; unspecified: the compiler could make
;;; '(2 3) a suffix of '(1 2 3),
;;; or they could be separate objects.
(ldiff '(1 2 3) '(2 3)) -> either (1) or (1 2 3)

;; b is the (1 2) suffix of a, so the ldiff is (1)
(let* ((a '(1 2 3)) (b (cdr a)))
  (ldiff a b))
-> (1)

;; Rule 5: strings and vector
(ldiff "abc" "bc") -> "a"
(ldiff "abc" nil) -> "abc"
(ldiff #(1 2 3) #(3)) -> #(1 2)

;; Rule 5: mixed vector kinds
(ldiff "abc" #(#\b #\c)) -> "abc"

;; Rule 6:
(ldiff #(1 2 3) '(3)) -> #(1 2 3)

;; Rule 4:
(ldiff '(1 2 3) #(3)) -> '(1 2 3)
(ldiff '(1 2 3 . #(3)) #(3)) -> '(1 2 3)
(ldiff '(1 2 3 . 4) #(3)) -> '(1 2 3 . 4)

;; Rule 6
(ldiff 1 2) -> 1
(ldiff 1 1) -> nil
```

9.22.17 Function `search`

Syntax:

```
(search haystack needle [testfun [keyfun]])
```

Description:

The `search` function determines whether the sequence *`needle`* occurs as substring within *`haystack`*, under the given comparison function *`testfun`* and key function *`keyfun`*. If this is the case, then the zero-based position of the leftmost occurrence of *`key`* within *`haystack`* is returned. Otherwise `nil` is returned to indicate that *`key`* does not occur within *`haystack`*. If *`key`* is empty, then zero is always returned.

The arguments *`haystack`* and *`needle`* are sequences. They may not be hash tables.

If *needle* is not empty, then it occurs at some position N within *haystack* if the first element of *needle* matches the element at position N of *haystack*, the second element of *needle* matches the element at position N+1 of *haystack* and so forth, for all elements of *needle*. A match between elements is determined by passing each element through *keyfun*, and then comparing the resulting values using *testfun*.

If *testfun* is supplied, it must be a function which can be called with two arguments. If it is not supplied, it defaults to `eql`.

If *keyfun* is supplied, it must be a function which can be called with one argument. If it is not supplied, it defaults to `identity`.

Examples:

```
;; fails because 3.0 doesn't match 3
;; under the default eql function
[search #(1.0 3.0 4.0 7.0) '(3 4)] -> nil

;; occurrence found at position 1:
;; (3.0 4.0) matches (3 4) under =
[search #(1.0 3.0 4.0 7.0) '(3 4) =] -> 1

;; "even odd odd odd even" pattern
;; matches at position 2
[search #(1 1 2 3 5 7 8) '(2 1 1 1 2) : evenp] -> 2

;; Case insensitive string search
[search "abcd" "CD" : chr-toupper] -> 2

;; Case insensitive string search
;; using vector of characters as key
[search "abcd" #(#\C #\D) : chr-toupper] -> 2
```

9.22.18 Function `contains`

Syntax:

```
(contains needle haystack [testfun [keyfun]])
```

Description:

The syntax of the `contains` function differs from that of `search`: that the *needle* and *haystack* arguments are reversed. The semantics is identical.

9.22.19 Function `rsearch`

Syntax:

```
(rsearch haystack needle [testfun [keyfun])
```

Description:

The `rsearch` function is like `search` except for two differences.

Firstly, if *needle* matches *haystack* in multiple places, `rsearch` returns the rightmost matching position rather than the leftmost.

Secondly, if *needle* is an empty sequence, then `rsearch` returns the length of `haystack`,

thereby effectively declaring that the rightmost match for an empty *needle* key occurs at the imaginary position past the element of *haystack*.

9.22.20 Function search-all

Syntax:

```
(search-all haystack needle [testfun [keyfun]])
```

Description:

The search-all function is closely related to the search and rsearch functions. Whereas those two functions return the leftmost or rightmost position, respectively, of *needle* within *haystack*, the search-all function returns a list of all the positions where *needle* occurs. The positions of overlapping matches are included in the list.

If *needle* is not found in *haystack*, search-all returns the empty list nil.

If *needle* is empty, then search-all returns a list of all positions in *haystack* including the one position past the last element. In this situation, if *haystack* is empty, the list (0) is returned. If *haystack* contains one item, then the list (0 1) is returned and so forth.

In all situations in which search-all returns a non-empty list, the first element of that list is what search would return for the same arguments, and the last element is what rsearch would return.

9.22.21 Functions ref and refset

Syntax:

```
(ref sequence index)
(refset sequence index new-value)
```

Description:

The ref and refset functions perform array-like indexing into sequences, as well as objects of type buf and carray.

If the *sequence* parameter is a hash, then these functions perform has retrieval and storage; in that case *index* isn't restricted to an integer value.

If *sequence* is a structure, it supports ref directly if it has a lambda method. The *index* argument is passed to that method, and the resulting value is returned. If a structure lacks a lambda method, but has a car method, then ref treats it as a list, traversing the structure using car/cdr operations. In the absence of support for these operations, the function fails with an error exception.

Similarly, a structure supports refset directly if it has a lambda-set method. This gets called with *index* and *new-value* as arguments. Then *new-value* is returned. If a structure lacks a lambda-set method, then refset treats it as a list, traversing the structure using car/cdr operations, and storing *new-value* using rplaca. In the absence of support for these operations, the function fails with an error exception.

The ref function retrieves an element of *sequence*, whereas refset overwrites an element of *sequence* with a new value.

If *sequence* is a sequence then *index* argument must be an integer. The first element of the sequence is indexed by zero. Negative values are permitted, denoting backward indexing from the

end of the sequence, such that the last element is indexed by -1, the second last by -2 and so on. See also the Range Indexing section under the description of the `dwim` operator.

If *`sequence`* is a list, then out-of-range indices, whether positive or negative, are treated leniently by `ref`: such accesses produce the value `nil`, rather than an error. For other sequence types, such accesses are erroneous. For hashes, accesses to nonexistent elements are treated leniently, and produce `nil`.

If *`sequence`* is a search tree, then `ref` behaves like `tree-lookup`. The `refset` function is not supported by search trees.

The `refset` function is strict for out-of-range indices over all sequences, including lists. In the case of hashes, a `refset` of a nonexistent key creates the key.

The `refset` function returns `new-value`.

The following equivalences hold between `ref` and `refset`, and the DWIM bracket syntax, provided that *`idx`* is a scalar index and *`sequence`* is a sequence object, rather than a hash.

```
(ref seq idx) <--> [seq idx]

(refset seq idx new) <--> (set [seq idx] new)
```

The difference is that `ref` and `refset` are first class functions which can be used in functional programming as higher order functions, whereas the bracket notation is syntactic sugar, and `set` is an operator, not a function. Therefore the brackets cannot replace all uses of `ref` and `refset`.

9.22.22 Function `update`

Syntax:

```
(update sequence function)
```

Description:

The `update` function replaces each elements in *`sequence`* in a hash table, with the result of *`function`* being applied to that element value.

The *`sequence`* is returned.

The *`sequence`* may be a hash table. In that case, *`function`* is invoked with each hash value, which is replaced with the function's return value.

9.22.23 Functions `remq`, `remql` and `remqual`

Syntax:

```
(remq object sequence [key-function])
(remql object sequence [key-function])
(remqual object sequence [key-function])
```

Description:

The `remq`, `remql` and `remqual` functions produce a new sequence based on *`sequence`*, removing the elements whose associated keys are `eq`, `eql` or `equal` to *`object`*.

The input *`sequence`* is unmodified, but the returned sequence may share substructure with it. If no items are removed, it is possible that the return value is *`sequence`* itself.

If *key-function* is omitted, then the element keys compared to *object* are the elements themselves. Otherwise, *key-function* is applied to each element and the resulting value is that element's key which is compared to *object*.

9.22.24 Functions remq*, remql* and remqual*

Syntax:

```
(remq* object sequence)
(remql* object sequence)
(remqual* object sequence)
```

Description:

The remq*, remql* and remqual* functions are lazy analogs of remq, remql and remqual. Rather than computing the entire new sequence prior to returning, these functions return a lazy list.

Caution: these functions can still get into infinite looping behavior. For instance, in (remql* 0 (repeat '(0))), remql will keep consuming the 0 values coming out of the infinite list, looking for the first item that does not have to be deleted, in order to instantiate the first lazy value.

Examples:

```
;; Return a list of all the natural numbers, excluding 13,
;; then take the first 100 of these.
;; If remql is used, it will loop until memory is exhausted,
;; because (range 1) is an infinite list.

[(remql* 13 (range 1)) 0..100]
```

9.22.25 Functions keepq, keepql and keepqual

Syntax:

```
(keepq object sequence [key-function])
(keepql object sequence [key-function])
(keepqual object sequence [key-function])
```

Description:

The keepq, keepql and keepqual functions produce a new sequence based on *sequence*, removing the items whose keys are not eq, eql or equal to *object*.

The input *sequence* is unmodified, but the returned sequence may share substructure with it. If no items are removed, it is possible that the return value is *sequence* itself.

The optional *key-function* is applied to each element from the *sequence* to convert it to a key which is compared to *object*. If *key-function* is omitted, then each element itself of *sequence* is compared to *object*.

9.22.26 Functions remove-if, keep-if, separate, remove-if* and keep-if*

Syntax:

```
(remove-if predicate-function sequence [key-function])
(keep-if predicate-function sequence [key-function])
(separate predicate-function sequence [key-function])
(remove-if* predicate-function sequence [key-function])
```

(keep-if* *predicate-function sequence* [*key-function*])

Description:

The `remove-if` function produces a sequence whose contents are those of *sequence* but with those elements removed which satisfy *predicate-function*. Those elements which are not removed appear in the same order. The result sequence may share substructure with the input sequence, and may even be the same sequence object if no items are removed.

The optional *key-function* specifies how each element from the *sequence* is transformed to an argument to *predicate-function*. If this argument is omitted then the predicate function is applied to the elements directly, a behavior which is identical to *key-function* being `(fun identity)`.

The `keep-if` function is exactly like `remove-if`, except the sense of the predicate is inverted. The function `keep-if` retains those items which `remove-if` will delete, and removes those that `remove-if` will preserve.

The `separate` function combines `keep-if` and `remove-if` into one, returning a list of two elements whose `car` and `cadr` are the result of calling `keep-if` and `remove-if`, respectively, on *sequence* (with the *predicate-function* and *key-function* arguments passed through). One of the two elements may share substructure with the input sequence, and may even be the same sequence object if all items are either kept or removed (in which case the other element will be `nil`).

Note: the `separate` function may be understood in terms of the following reference implementation:

```
(defun separate (pred seq : (keyfun :))
  [(juxt (op keep-if pred @1 keyfun)
         (op remove-if pred @1 keyfun))
   seq])
```

The `remove-if*` and `keep-if*` functions are like `remove-if` and `keep-if`, but produce lazy lists.

Examples:

```
;; remove any element numerically equal to 3.
(remove-if (op = 3) '(1 2 3 4 3.0 5)) -> (1 2 4 5)

;; remove those pairs whose first element begins with "abc"
[remove-if (op equal [@1 0..3] "abc")
           '(("abcd" 4) ("defg" 5))
           car]
-> (("defg" 5))

;; equivalent, without test function
(remove-if (op equal [(car @1) 0..3] "abc")
           '(("abcd" 4) ("defg" 5)))
-> (("defg" 5))
```

9.22.27 Functions `countqual`, `countql` and `countq`

Syntax:

```
(countq object iterable)
(countql object iterable)
(countqual object iterable)
```

Description:

The countq, countql and countqual functions count the number of objects in *iterable* which are eq, eql or equal to *object*, and return the count.

9.22.28 Functions count and count-if

Syntax:

```
(count key sequence [testfun [keyfun]])
(count-if predfun iterable [keyfun])
```

Description:

The count and count-if functions search through *sequence* for items which match *key*, or satisfy the predicate function *predfun*, respectively. They return the number of matching or predicate-satisfying items.

The *keyfun* argument specifies a function which is applied to the elements of *sequence* to produce the comparison key. If this argument is omitted, then the untransformed elements of *sequence* are examined.

The count function's *testfun* argument specifies the test function which is used to compare the comparison keys from *sequence* to *key*. If this argument is omitted, then the equal function is used. The count function returns the number of elements of *sequence* whose comparison key (as retrieved by *keyfun*) matches the *key* object, as compared by *testfun*.

The count-if function's *predfun* argument specifies a predicate function which is applied to the successive comparison keys taken from *sequence*. The function returns the count of the number keys for which *predfun* returns true.

9.22.29 Functions posq, posql and posqual

Syntax:

```
(posq object sequence)
(posql object sequence)
(posqual object sequence)
```

Description:

The posq, posql and posqual functions return the zero-based position of the first item in *sequence* which is, respectively, eq, eql or equal to *object*.

9.22.30 Functions pos and pos-if

Syntax:

```
(pos key sequence [testfun [keyfun]])
(pos-if predfun sequence [keyfun])
```

Description:

The pos and pos-if functions search through *sequence* for an item which matches *key*, or satisfies the predicate function *predfun*, respectively. They return the zero-based position of the matching item.

The *keyfun* argument specifies a function which is applied to the elements of *sequence* to produce the comparison key. If this argument is omitted, then the untransformed elements of *sequence* are examined.

The pos function's *testfun* argument specifies the test function which is used to compare the comparison keys from *sequence* to *key*. If this argument is omitted, then the equal function is used. The pos function returns the position of the first element of *sequence* whose comparison key (as retrieved by *keyfun*) matches *key*, as compared by the *testfun* function. If no such element is found, nil is returned.

The pos-if function's *predfun* argument specifies a predicate function which is applied to the successive comparison keys taken from *sequence* by applying *keyfun* to successive elements. The position of the first element for which *predfun* yields true is returned. If no such element is found, nil is returned.

9.22.31 Functions rposq, rposql, rposqual, rpos and rpos-if

Syntax:

```
(rposq object sequence)
(rposql object sequence)
(rposqual object sequence)
(rpos key sequence [testfun [keyfun]])
(rpos-if predfun sequence [keyfun])
```

Description:

These functions are counterparts of rposq, rposql, rposqual, rpos and rpos-if which report position of the rightmost matching item, rather than the leftmost.

9.22.32 Functions pos-max and pos-min

Syntax:

```
(pos-max sequence [testfun [keyfun]])
(pos-min sequence [testfun [keyfun]])
```

Description:

The pos-min and pos-max functions implement exactly the same algorithm; they differ only in their defaulting behavior with regard to the *testfun* argument. If *testfun* is not given, then the pos-max function defaults *testfun* to the greater function, whereas pos-min defaults it to the less function.

If *sequence* is empty, both functions return nil.

Without a *testfun* argument, the pos-max function finds the zero-based position index of the numerically maximum value occurring in *sequence*, whereas pos-min without a *testfun* argument finds the index of the minimum value.

If a *testfun* argument is given, the two functions are equivalent. The *testfun* function must be callable with two arguments. If *testfun* behaves like a greater-than comparison, then pos-max and pos-min return the index of the maximum element. If *testfun* behaves like a less-than comparison, then the functions return the index of the minimum element.

The *keyfun* argument defaults to the identity function. Each element from *sequence* is passed through this one-argument function, and the resulting value is used in its place.

If a sequence contains multiple equivalent maxima, whether the position of the leftmost or rightmost such maximum is reported depends on whether *testfun* compares for strict inequality, or whether it reports true for equal arguments also. Under the default *testfun*, which is `less`, the `pos-max` function will return the position leftmost of a duplicate set of maximum elements. To find the rightmost of the maxima, the `lequal` function can be substituted. Analogous reasoning applies to other test functions.

9.22.33 Function `subst`

Syntax:

```
(subst old new seq [testfun [keyfun]])
```

Description:

The `subst` function returns a sequence of the same type as *seq* in which elements of *seq* which match the *old* object have been replaced with the *new* object.

To form the comparison keys, the elements of *seq* are projected through the *testfun* function, which defaults to `identity`, so the items themselves are used as keys by default.

Keys are compared to the *old* value using *testfun*, which defaults to `equal`.

Examples:

```
(subst "brown" "black" #("how" "now" "brown" "cow"))
-> #("how" "now" "black" "cow"))

;; elements are converted to lower case to form keys
[subst "brown" "black"
       #("how" "now" "BROWN" "cow") : downcase-str]
-> #("how" "now" "black" "cow")

;; using < instead of equality, replace elements
;; greater than 5 with 0
[subst 5 0 '(1 2 3 4 5 6 7 8 9 10) <] (1 2 3 4 5 0 0 0 0 0))
```

9.22.34 Functions `subq`, `subql` and `subqual`

Syntax:

```
(subq old new sequence)
(subql old new sequence)
(subqual old new sequence)
```

Description:

The `subq`, `subql` and `subqual` functions return a sequence of the same kind as *sequence* in which elements matching the *old* object are replaced by *new* object.

The matching elements are identified by comparing with *old* using, respectively, the functions `eq`, `eql`, and `equal`.

Examples:

```
(subq #\b #\z "abc") -> "azc"
(subql 1 3 #(0 1 2)) -> #(0 3 2)
```

```
(subqual "are" "do" '#"how are you")
-> ("how" "do" "you")
```

9.22.35 Function `mismatch`

Syntax:

```
(mismatch left-seq right-seq [testfun [keyfun]])
```

Description:

The `mismatch` function compares corresponding elements from the sequences *`left-seq`* and *`right-seq`*, returning the position at which the first mismatch occurs.

If the sequences are of the same length, and their corresponding elements are the same, then `nil` is returned.

If one sequence is shorter than the other, and matches a prefix of the other, then the mismatching position returned is one position after the last element of the shorter sequence, the same value as its length. An empty sequence is a prefix of every sequence.

The *`keyfun`* argument defaults to the `identity` function. Each element from *`sequence`* is passed to *`keyfun`* and the resulting value is used in its place.

After being converted through *`keyfun`*, items are then compared using *`testfun`*, which must accept two arguments, and defaults to `equal`.

9.22.36 Function `where`

Syntax:

```
(where function iterable)
```

Description:

If *`iterable`* is a sequence, the `where` function returns a lazy list of the numeric indices of those of its elements which satisfy *`function`*. The numeric indices appear in increasing order.

If *`iterable`* is a hash, the following special behavior applies: `where` returns a lazy list of of keys which have values which satisfy *`function`*. These keys are not subject to an order.

`function` must be a function that can be called with one argument. For each element of *`iterable`*, *`function`* is called with that element as an argument. If a non-`nil` value is returned, then the zero-based index of that element is added to a list. Finally, the list is returned.

9.22.37 Function `rmismatch`

Syntax:

```
(rmismatch left-seq right-seq [testfun [keyfun]])
```

Description:

Similarly to `mismatch`, the `rmismatch` function compares corresponding elements from the sequences *`left-seq`* and *`right-seq`*, returning the position at which the first mismatch occurs. All of the arguments have the same semantics as that of `mismatch`.

Unlike `mismatch`, `rmismatch` compares the sequences right-to-left, finding the suffix which they have in common, rather than prefix.

If the sequences match, then `nil` is returned. Otherwise, a negative index is returned giving the mismatching position, regarded from the end. If the sequences match only in the rightmost element, then -1 is returned. If they match in two elements then -2 and so forth.

9.22.38 Functions `starts-with` and `ends-with`

Syntax:

```
(starts-with short-seq long-seq [testfun [keyfun]])
(ends-with short-seq long-seq [testfun [keyfun]])
```

Description:

The `starts-with` and `ends-with` functions compare corresponding elements from sequences *`short-seq`* and *`long-seq`*.

The `starts-with` function returns `t` if *`short-seq`* is prefix of *`long-seq`*; otherwise, it returns `nil`.

The `ends-with` function returns `t` if *`short-seq`* is suffix of *`long-seq`*; otherwise, it returns `nil`.

Element from both sequences are mapped to comparison keys using *`keyfun`*, which defaults to `identity`.

Comparison keys are compared using *`testfun`* which defaults to `equal`.

9.22.39 Function `select`

Syntax:

```
(select sequence {index-list | function})
```

Description:

The `select` function returns a sequence, of the same kind as *`sequence`*, which consists of those elements of *`sequence`* which are identified by the indices in *`index-list`*, which may be a list or a vector.

If *`function`* is given instead of *`index-list`*, then *`function`* is invoked with *`sequence`* as its argument. The return value is then taken as if it were the *`index-list`* argument .

If *`sequence`* is a sequence, then *`index-list`* consists of numeric indices. The length of the sequence, as reported by the `length` function, is added to every *`index-list`* value which is negative. The `select` function stops collecting values upon encountering an index value which is greater than or equal to the length of the sequence. (Rationale: without this strict behavior, `select` would not be able to terminate if *`index-list`* is infinite.)

If *`sequence`* is, more specifically, a list-like sequence, then *`index-list`* must contain monotonically increasing numeric values, even if no value is out of range, since the `select` function makes a single pass through the list based on the assumption that indices are ordered. (Rationale: optimization.) This requirement for monotonicity applies to the values which result after negative indices are displaced by the sequence length Also, in this list-like sequence case, values taken from *`index-list`* which are still negative after being displaced by the sequence length are ignored.

If *`sequence`* is a hash, then *`index-list`* is a list of keys. A new hash is returned which contains those elements of *`sequence`* whose keys appear in *`index-list`*. All of *`index-list`* is

processed, even if it contains keys which are not in *sequence*. The nonexistent keys are ignored.

The `select` function also supports objects of type `carray`, in a manner similar to vectors. The indicated elements are extracted from the input sequence, and a new `carray` is returned whose storage is initialized by converting the extracted values back to the foreign representation.

9.22.40 Function `reject`

Syntax:

```
(reject sequence {index-list | function})
```

Description:

The `reject` function returns a sequence, of the same kind as *sequence*, which consists of all those elements of *sequence* which are not identified by the indices in *index-list*, which may be a list or a vector.

If *function* is given instead of *index-list*, then *function* is invoked with *sequence* as its argument. The return value is then taken as if it were the *index-list* argument .

If `sequence` is a hash, then *index-list* represents a list of keys. The `reject` function returns a duplicate of the hash, in which the keys specified in *index-list* do not appear.

Otherwise if *sequence* is a vector-like sequence, then the behavior of `reject` may be understood by the following equivalence:

```
(reject seq idx)  -->  (make-like
                         [apply append (split* seq idx)]
                         seq)
```

where it is to be understood that *seq* is evaluated only once.

If *sequence* is a list, then, similarly, the following equivalence applies:

```
(reject seq idx)  -->  (make-like
                         [apply append* (split* seq idx)]
                         seq)
```

The input sequence is split into pieces at the indicated indices, such that the elements at the indices are removed and do not appear in the pieces. The pieces are then appended together in order, and the resulting list is coerced into the same type of sequence as the input sequence.

9.22.41 Function `relate`

Syntax:

```
(relate domain-seq range-seq [default-val])
```

Description:

The `relate` function returns a one-argument function which implements the relation formed by mapping the elements of *domain-seq* to the positionally corresponding elements of *range-seq*. That is to say, the function searches through the sequence *domain-seq* to determine the position where its argument occurs, using `equal` as the comparison function. Then it returns the element from that position in the *range-seq* sequence. This returned function is called the *relation function*.

If the relation function's argument is not found in *domain-seq*, then the behavior depends on the optional parameter *default-val*. If an argument is given for *default-val*, then the relation function returns that value. Otherwise, the relation function returns its argument.

Note: the `relate` function may be understood in terms of the following equivalences:

```
(relate d r)  <-->  (lambda (arg)
                      (iflet ((p (posqual arg d)))
                        [r p]
                        arg))

(relate d r v)  <-->  (lambda (arg)
                        (iflet ((p (posqual arg d)))
                          [r p]
                          v))
```

Note: `relate` may return a hash table instead of a function, if such an object can satisfy the semantics required by the arguments.

Examples:

```
(mapcar (relate "_" "-") "foo_bar")  ->  "foo-bar"

(mapcar (relate "0123456789" "ABCDEFGHIJ" "X") "139D-345")
-> "BJDXXDEF"

(mapcar (relate '(nil) '(0)) '(nil 1 2 nil 4)) -> (0 1 2 0 4)
```

9.22.42 Function `in`

Syntax:

```
(in sequence key [testfun [keyfun]])
(in hash key)
```

Description:

The `in` function tests whether *key* is found inside *sequence* or *hash*.

If the *testfun* argument is specified, it specifies the function which is used to comparison keys from the sequence to *key*. Otherwise the `equal` function is used.

If the *keyfun* argument is specified, it specifies a function which is applied to the elements of *sequence* to produce the comparison keys. Without this argument, the elements themselves are taken as the comparison keys.

If the object being searched is a hash, then if neither of the arguments *keyfun* nor *testfun* is specified, `in` performs a hash lookup for `key`, returning `t` if the key is found, `nil` otherwise. If either of *keyfun* or *testfun* is specified, then `in` performs an exhaustive search of the hash table, as if it were a sequence of `cons` cells whose `car` fields are keys, and whose `cdr` keys are values. Thus to search by key, the `car` function must be specified as *keyfun*.

The `in` function returns `t` if it finds *key* in *sequence* or *hash*, otherwise `nil`.

9.22.43 Function `partition`

Syntax:

```
(partition sequence {index-list | index | function})
```

Description:

If *sequence* is empty, then `partition` returns an empty list, and the second argument is ignored; if it is *function*, it is not called.

Otherwise, `partition` returns a lazy list of partitions of *sequence*. Partitions are consecutive, non-overlapping, nonempty substrings of *sequence*, of the same kind as *sequence*, such that if these substrings are catenated together in their order of appearance, a sequence `equal` to the original is produced.

If the second argument is of the form *index-list*, or if an *index-list* was produced from the *index* or *function* arguments, each value in that list must be an integer. Each integer value which is nonnegative specifies the index position given by its value. Each integer value which is negative specifies an index position given by adding the length of *sequence* to its value. The sequence index positions thus denoted by *index-list* shall be strictly nondecreasing. Each successive element is expected to designate an index position at least as high as all previous elements, otherwise the behavior is unspecified. Leading index positions which are (still) negative, or zero, are effectively ignored.

If *index-list* is empty then a one-element list containing the entire *sequence* is returned.

If *index-list* is an infinite lazy list, the function shall terminate if that list eventually produces an index position which is greater than or equal to the length of *sequence*.

If the second argument is a function, then this function is applied to *sequence*, and the return value of this call is then used in place of the second argument, which must be a single index value, which is then taken as if it were the *index* argument, or else a list of indices, which are taken as the *index-list* argument.

If the second argument is an atom other than a function, it is assumed to be an integer index, and is turned into an *index-list* of one element.

After the *index-list* is obtained as an argument, or determined from the *index* or *function* arguments, the `partition` function then divides *sequence* according to the indices given by that list. The first partition begins with the first element of *sequence*. The second partition begins at the first position in *index-list*, and so on. Indices beyond the length of the sequence are ignored, as are indices less than or equal to zero.

Examples:

```
(partition '(1 2 3) 1) -> ((1) (2 3))

;; split the string where there is a "b"
(partition "abcbcbd" (op where (op eql #\b))) -> ("a" "bc"
                                                   "bc" "bd")
```

9.22.44 Functions `split` and `split*`

Syntax:

```
(split sequence {index-list | index | function})
```

(split* *sequence* {*index-list* | *index* | *function*})

Description:

If *sequence* is empty, then both split and split* return an empty list, and the second argument is ignored; if it is *function*, it is not called.

Otherwise, split returns a lazy list of pieces of *sequence*: consecutive, non-overlapping, possibly empty substrings of *sequence*, of the same kind as *sequence*. A catenation of these pieces in the order they appear would produce a sequence that is equal to the original sequence.

The split* function differs from split in that the elements indicated by the split indices are removed.

The *index*, *index-list*, and *function* arguments are subject to the same restrictions and treatment as the corresponding arguments of the partition function, with the following difference: the index positions indicated by index-list are required to be strictly increasing, rather than nondecreasing.

If the second argument is of the form *index-list*, or if an *index-list* was produced from the *index* or *function* arguments, then the split function divides *sequence* according to the indices indicated in the list. The first piece always begins with the first element of *sequence*. Each subsequent piece begins with the position indicated by an element of *index-list*. Negative indices are ignored. If *index-list* includes index zero, then an empty first piece is generated. If *index-list* includes an index greater than or equal to the length of *sequence* (equivalently, an index beyond the last element of the sequence) then an additional empty last piece is generated. The length of *sequence* is added to any negative indices. An index which is still negative after being thus displaced is discarded.

Note: the principal difference between split and partition is that partition does not produce empty pieces.

Examples:

```
(split '(1 2 3) 1) -> ((1) (2 3))

(split "abc" 0) -> ("" "abc")
(split "abc" 3) -> ("abc" "")
(split "abc" 1) -> ("a" "bc")
(split "abc" '(0 1 2 3)) -> ("" "a" "b" "c" "")
(split "abc" '(1 2)) -> ("a" "b" "c")

(split "abc" '(-1 1 2 15)) -> ("a" "b" "c")

;; triple split at makes two additional empty pieces
(split "abc" '(1 1 1)) -> ("a" "" "" "bc")

(split* "abc" 0) -> ("" "bc") ;; "a" is removed

;; all characters removed
(split* "abc" '(0 1 2)) -> ("" "" "" "")
```

9.22.45 Function partition*

Syntax:

```
(partition* sequence {index-list | index | function})
```

Description:

If *sequence* is empty, then `partition*` returns an empty list, and the second argument is ignored; if it is *function*, it is not called.

The *index*, *index-list*, and *function* arguments are subject to the same restrictions and treatment as the corresponding arguments of the `partition` function, with the following difference: the index positions indicated by `index-list` are required to be strictly increasing, rather than nondecreasing.

If the second argument is of the form *index-list*, then `partition*` produces a lazy list of pieces taken from *sequence*. The pieces are formed by deleting from *sequence* the elements at the positions given in *index-list*, such that the pieces are the remaining nonempty substrings from between the deleted elements, maintaining their order.

If *index-list* is empty then a one-element list containing the entire *sequence* is returned.

Examples:

```
(partition* '(1 2 3 4 5) '(0 2 4)) -> ((2) (4))

(partition* "abcd" '(0 3)) -> "bc"

(partition* "abcd" '(0 1 2 3)) -> nil
```

9.22.46 Functions `find`, `find-if` and `find-true`

Syntax:

```
(find key sequence [testfun [keyfun]])
(find-if predfun {sequence | hash} [keyfun])
(find-true predfun {sequence | hash} [keyfun])
```

Description:

The `find` and `find-if` functions search through a sequence for an item which matches a key, or satisfies a predicate function, respectively. The `find-true` function is a variant of `find-if` which returns the value of the predicate function instead of the item.

The *keyfun* argument specifies a function which is applied to the elements of *sequence* to produce the comparison key. If this argument is omitted, then the untransformed elements of the *sequence* are searched.

The `find` function's *testfun* argument specifies the test function which is used to compare the comparison keys from *sequence* to the search key. If this argument is omitted, then the `equal` function is used. The first element from the list whose comparison key (as retrieved by *keyfun*) matches the search (under *testfun*) is returned. If no such element is found, `nil` is returned.

The `find-if` function's *predfun* argument specifies a predicate function which is applied to the successive comparison keys pulled from the list by applying *keyfun* to successive elements. The first element for which *predfun* yields true is returned. If no such element is found, `nil` is returned.

In the case of `find-if`, a hash table may be specified instead of a sequence. The *hash* is treated as if it were a sequence of hash key and hash value pairs represented as cons cells, the `car`

slots of which are the hash keys, and the `cdr` of which are the hash values. If the caller doesn't specify a *keyfun* then these cells are taken as their keys.

The `find-true` function's argument conventions and search semantics are identical to those of `find-if`, but the return value is different. Instead of returning the found item, `find-true` returns the value which *predfun* returned for the found item's key.

9.22.47 Functions `rfind` and `rfind-if`

Syntax:

```
(rfind key sequence [testfun [keyfun]])
(rfind-if predfun {sequence | hash} [keyfun])
```

Description:

The `rfind` and `rfind-if` functions are almost exactly like `find` and `find-if` except that if there are multiple matches for *key* in *sequence*, they return the rightmost element rather than the leftmost.

In the case of `rfind-if` when a *hash* is specified instead of a *sequence*, the function searches through the hash entries in the same order as `find-if`, but finds the last match rather than the first. Note: hashes are inherently not ordered; the relative order of items in a hash table can change when other items are inserted or deleted.

9.22.48 Functions `find-max` and `find-min`

Syntax:

```
(find-max iterable [testfun [keyfun]])
(find-min iterable [testfun [keyfun]])
```

Description:

The `find-min` and `find-max` function implement exactly the same algorithm; they differ only in their defaulting behavior with regard to the *testfun* argument. If *testfun* is not given, then the `find-max` function defaults it to the `greater` function, whereas `find-min` defaults it to the `less` function.

Without a *testfun* argument, the `find-max` function finds the numerically maximum value occurring in *iterable*, whereas `pos-min` without a *testfun* argument finds the minimum value.

If a *testfun* argument is given, the two functions are equivalent. The *testfun* function must be callable with two arguments. If *testfun* behaves like a greater-than comparison, then `find-max` and `find-min` both return the maximum element. If *testfun* behaves like a less-than comparison, then the functions return the minimum element.

The *keyfun* argument defaults to the `identity` function. Each element from *sequence* is passed through this one-argument function, and the resulting value is used in its place for the purposes of the comparison. However, the original element is returned.

If there are multiple equivalent maxima, then under the default *testfun*, that being `less`, the first one encountered while traversing *iterable* is the one that is reported. See the notes under `pos-max` regarding duplicate maxima.

9.22.49 Functions `find-max-key` and `find-min-key`

Syntax:

```
(find-max-key iterable [ testfun [keyfun]])
(find-min-key iterable [ testfun [keyfun]])
```

Description:

The `find-min-key` and `find-max-key` functions have the same argument conventions as, respectively, `find-max` and `find-min` and agree with those functions in regard to which element of the input sequence is identified: all these functions identify the element which maximizes or minimizes the value of *keyfun*.

Whereas `find-max` and `find-min` return the maximizing or minimizing element itself, the `find-max-key` and `find-min-key` functions return the value of *keyfun* applied to the element.

Under the default *keyfun* value, that being the `identity` function, these functions behave the same as `find-max` and `find-min`.

9.22.50 Functions `uni`, `isec`, `isecp`, `diff` and `symdiff`

Syntax:

```
(uni iter1 iter1 [testfun [keyfun]])
(isec iter1 iter1 [testfun [keyfun]])
(isecp iter1 iter1 [testfun [keyfun]])
(diff iter1 iter1 [testfun [keyfun]])
(symdiff iter1 iter2 [testfun [keyfun]])
```

Description:

The functions `uni`, `isec`, `diff` and `symdiff` treat the sequences *iter1* and *iter2* as if they were sets.

They, respectively, compute the set union, set intersection, set difference and symmetric difference of *iter1* and *iter2*, returning a new sequence.

The `isecp` is Boolean: it returns `t` for those arguments for which `isec` returns a non-empty list, otherwise `nil`.

The arguments *iter1* and *iter2* need not be of the same kind. They may be hash tables.

The returned sequence is of the same kind as *iter1*. If *iter1* is a hash table, the returned sequence is a list.

For the purposes of these functions, an input which is a hash table is considered as if it were a sequence of hash key and hash value pairs represented as cons cells, the `car` slots of which are the hash keys, and the `cdr` of which are the hash values. This means that if no *keyfun* is specified, these pairs are taken as keys.

Since the input sequences are defined as representing sets, they are assumed not to contain duplicate elements. These functions are not required, but may, de-duplicate the sequences.

The union sequence produced by `uni` contains all of the elements which occur in both *iter1* and *iter2*. If a given element occurs exactly once only in *iter1* or exactly once only in *iter2*, or exactly once in both sequences, then it occurs exactly once in the union sequence. If a

given element occurs at least once in either *iter1*, *iter2* or both, then it occurs at least once in the union sequence.

The intersection sequence produced by `isec` contains all of the elements which occur in both *iter1* and *iter2*. If a given element occurs exactly once in *iter1* and exactly once in *iter2*, then in occurs exactly once in the intersection sequence. If a given element occurs at least once in *iter1* and at least once in *iter2*, then in occurs at least once in the intersection sequence.

The difference sequence produced by `diff` contains all of the elements which occur in *iter1* but do not occur in *iter2*. If an element occurs exactly once in *iter1* and does not occur in *iter2*, then it occurs exactly once in the difference sequence. If an element occurs at least once in *iter1* and does not occur in *iter2*, then it occurs at least once in the difference sequence. If an element occurs at least once in *iter2*, then it does not occur in the difference sequence.

The symmetric difference sequence produced by `symdiff` contains all of the elements of *iter1* which do not occur in *iter2* and vice versa: it also contains all of the elements of *iter2* which do not occur in *iter1*.

Element equivalence is determined by a combination of *testfun* and *keyfun*. Elements are compared pairwise, and each element of a pair is passed through *keyfun* function to produce a comparison value. The comparison values are compared using *testfun*. If *keyfun* is omitted, then the untransformed elements themselves are compared, and if *testfun* is omitted, then the `equal` function is used.

Note: a function similar to `diff` named `set-diff` exists. This became deprecated starting in **TXR** 184. For the `set-diff` function, the requirement was specified to preserve the original order of items from *iter1* that survive into the output sequence. This requirement is not documented for the `diff` function, but is de facto honored by the implementation for at as long as the `set-diff` synonym continues to be available. The `set-diff` function doesn't support hash tables and is inefficient for vectors and strings.

Note: these functions are not efficient for the processing of hash tables, even when both inputs are hashes, the *keyfun* argument is `car`, and *testfun* matches the equality used by both hash-table inputs. If applicable, the operations `hash-uni`, `hash-isec` and `hash-diff` should be used instead.

9.22.51 Functions `mapcar`, `mappend`, `mapcar*` and `mappend*`

Syntax:

```
(mapcar function iterable*)
(mappend function iterable*)
(mapcar* function iterable*)
(mappend* function iterable*)
```

Description:

When given only one argument, the `mapcar` function returns `nil`. *function* is never called.

When given two arguments, the `mapcar` function applies *function* to each elements of *iterable* and returns a sequence of the resulting values in the same order as the original values. The returned sequence is the same kind as *iterable*, if possible. If the accumulated values cannot be elements of that type of sequence, then a list is returned.

When additional sequences are given as arguments, this filtering behavior is generalized in the

following way: `mapcar` traverses the sequences in parallel, taking a value from each sequence as an argument to the function. If there are two lists, *`function`* is called with two arguments and so forth. The traversal is limited by the length of the shortest sequence. The return values of the function are collected into a new sequence which is returned. The returned sequence is of the same kind as the leftmost input sequence, unless the accumulated values cannot be elements of that type of sequence, in which case a list is returned.

The `mappend` function works like `mapcar`, with the following difference. Rather than accumulating the values returned by the function into a sequence, mappend expects the items returned by the function to be sequences which are catenated with `append`, and the resulting sequence is returned. The returned sequence is of the same kind as the leftmost input sequence, unless the values cannot be elements of that type of sequence, in which case a list is returned.

The `mapcar*` and `mappend*` functions work like `mapcar` and `mappend`, respectively. However, they return lazy lists rather than generating the entire output list prior to returning.

Caveats:

Like `mappend`, `mappend*` must "consume" empty lists. For instance, if the function being mapped puts out a sequence of `nil`s, then the result must be the empty list `nil`, because `(append nil nil nil nil ...)` is `nil`.

But suppose that `mappend*` is used on inputs which are infinite lazy lists, such that the function returns `nil` values indefinitely. For instance:

```
;; Danger: infinite loop!!!
(mappend* (fun identity) (repeat '(nil)))
```

The `mappend*` function is caught in a loop trying to consume and squash an infinite stream of `nil`s, and so doesn't return.

Examples:

```
;; multiply every element by two
(mapcar (lambda (item) (* 2 item)) '(1 2 3)) -> (4 6 8)

;; "zipper" two lists together
(mapcar (lambda (le ri) (list le ri)) '(1 2 3) '(a b c))
-> '((1 a) (2 b) (3 c)))

;; like append, mappend allows a lone atom or a trailing atom:
(mappend (fun identity) 3) -> (3)
(mappend (fun identity) '((1) 2)) -> (1 . 2)

;; take just the even numbers
(mappend (lambda (item) (if (evenp x) (list x))) '(1 2 3 4 5))
-> (2 4)
```

9.22.52 Functions `maprod`, `maprend` and `maprodo`

Syntax:

```
(maprod function iterable*)
(maprend function iterable*)
(maprodo function iterable*)
```

Description:

The `maprod`, `maprend` and `maprodo` functions resemble `mapcar`, `mappend` and `mapdo`, respectively. When given no *`iterable`* arguments or exactly one *`iterable`* argument, they behave exactly like those three functions.

When two or more *`iterable`* arguments are present, `maprod` differs from `mapcar` in the following way, as do the remaining functions from their aforementioned counterparts. Whereas `mapcar` iterates over the *`iterable`* values in parallel, taking successive tuples of element values and passing them to *`function`*, the `maprod` function iterates over all *combinations* of elements from the sequences: the Cartesian product. The `prod` suffix stands for "product".

If one or more *`iterable`* arguments specify an empty sequence, then the Cartesian product is empty. In this situation, *`function`* is not called. The result of the function is then `nil` converted to the same kind of sequence as the leftmost *`iterable`*.

The `maprod` function collects the values into a list just as `mapcar` does. Just like `mapcar`, it converts the resulting list into the same kind of sequence as the leftmost *`iterable`* argument, if possible. For instance, if the resulting list is a list or vector of characters, and the leftmost *`iterable`* is a character string, then the list or vector of characters is converted to a character string and returned.

The `maprend` function ("map product through function and append") iterates the *`iterable`* element combinations exactly like `maprod`, passing them as arguments to *`function`*. The values returned by *`function`* are then treated exactly as by the `mappend` function. The return values are expected to be sequences which are appended together as if by `append`, and the final result is converted to the same kind of sequence as the leftmost *`iterable`* if possible.

The `maprodo` function, like `mapdo`, ignores the result of *`function`* and returns `nil`.

The combination iteration gives priority to the rightmost *`iterable`*, which means that the rightmost element of each generated tuple varies fastest: the tuples are traversed in "rightmost major" order. This is made clear in the examples.

Examples

```
[maprod list '(0 1 2) '(a b) '(i ii iii)]
->
((0 a i) (0 a ii) (0 a iii) (0 b i) (0 b ii) (0 b iii)
 (1 a i) (1 a ii) (1 a iii) (1 b i) (1 b ii) (1 b iii)
 (2 a i) (2 a ii) (2 a iii) (2 b i) (2 b ii) (2 b iii))

;; Vectors #(#\a #\x) #(#\a #\y) ... are appended
;; together resulting in #(#\a #\x #\a #\y ...)
;; which is converted to a string:

[maprend vec "ab" "xy"] -> "axaybxby"

;; One of the sequences is empty, so the product is an
;; empty sequence of the same kind as the leftmost
;; sequence argument, thus an empty string:
[maprend vec "ab" ""] -> ""
```

9.22.53 Function `mapdo`

Syntax:

```
(mapdo function iterable*)
```

Description:

The `mapdo` function is similar to `mapcar`, but always returns `nil`. It is useful when *`function`* performs some kind of side effect, hence the "do" in the name, which is a mnemonic for the execution of imperative actions.

When only the *`function`* argument is given, *`function`* is never called, and `nil` is returned.

If a single *`iterable`* argument is given, then `mapdo` iterates over *`iterable`*, invoking *`function`* on each element.

If two or more *`iterable`* arguments are given, then `mapdo` iterates over the sequences in parallel, extracting parallel tuples of items. These tuples are passed as arguments to *`function`*, which must accept as many arguments as there are sequences.

9.22.54 Functions `transpose` and `zip`

Syntax:

```
(transpose iterable)
(zip iterable*)
```

Description:

The `transpose` function performs a transposition on *`iterable`*. This means that the elements of *`iterable`* must be iterable. These iterables are understood to be columns; transpose exchanges rows and columns, returning a sequence of the rows which make up the columns. The returned sequence is of the same kind as *`iterable`*, and the rows are also the same kind of sequence as the first column of the original sequence. The number of rows returned is limited by the shortest column among the sequences.

All of the input sequences (the elements of *`iterable`*) must have elements which are compatible with the first sequence. This means that if the first element of *`iterable`* is a string, then the remaining sequences must be strings, or else sequences of characters, or of strings.

The `zip` function takes variable arguments, and is equivalent to calling `transpose` on a list of the arguments. The following equivalences hold:

```
(zip . x) <--> (transpose x)

[apply zip x] <--> (transpose x)
```

Examples:

```
;; transpose list of lists
(transpose '((a b c) (c d e))) ->  ((a c) (b d) (c e))

;; transpose vector of strings:
;; - string columns become string rows
;; - vector input becomes vector output
(transpose #("abc" "def" "ghij")) -> #("adg" "beh" "cfi")
```

```
;; error: transpose wants to make a list of strings
;; but 1 is not a character
(transpose #("abc" "def" '(1 2 3))) ;; error!

;; String elements are catenated:
(transpose #("abc" "def" ("UV" "XY" "WZ")))
-> #("adUV" "beXY" "cfWZ")

;; Transpose list of ranges
(transpose (list 1..4 4..8 8..12))
-> ((1 4 8) (2 5 9) (3 6 10))

(zip '(a b c) '(c d e)) ->  ((a c) (b d) (c e))
```

9.22.55 Functions `window-map`, `window-mappend` and `window-mapdo`

Syntax:

```
(window-map range boundary function sequence)
(window-mappend range boundary function sequence)
(window-mapdo range boundary function sequence)
```

Description:

The `window-map` and `window-mappend` functions process the elements of *`sequence`* by passing arguments derived from each successive element to *`function`*. Both functions return, if possible, a sequence of the same kind as `sequence`, otherwise a list.

Under `window-map`, values returned by *`function`* are accumulated into a sequence of the same type as *`sequence`* and that sequence is returned. Under `window-mappend`, the values returned by the calls to *`function`* are expected to be sequence which are appended together to form the output sequence.

These functions are analogous to `mapcar` and `mappend`. Unlike these, they operate only on a single sequence, and over this sequence they perform a *sliding window mapping*, whose description follows.

The function `window-mapdo` avoids accumulating a sequence, and instead returns `nil`; it is analogous to `mapdo`.

The argument to the *`range`* parameter must be a positive integer, not exceeding 512. This parameter specifies the amount of ahead/behind context on either side of each element which is processed. It indirectly determines the window size for the mapping. The window size is twice *`range`*, plus one. For instance if range is 2, then the window size is 5: the element being processed lies at the center of the window, flanked by two elements on either side, making five.

The *`function`* argument must specify a function which accepts a number of arguments corresponding to the window size. For instance if *`range`* is 2, making the window size 5, then *`function`* must accept 5 arguments. These arguments constitute the sliding window being processed. Each time *`function`* is called, the middle argument is the element being processed, and the arguments surrounding it are its window.

When an element is processed from somewhere in the interior of a sequence, where it is flanked on either side by at least *`range`* elements, then the window is populated by those flanking elements taken from *`sequence`*.

The *boundary* parameter specifies the window contents which are used for the processing of elements which are closer than *range* to either end of the sequence. The argument may be a sequence containing at least twice *range* number of elements (one less than the window size): if it has additional elements, they are not used. If it is a list, it may be shorter than twice *range*. The argument may also be one of the two keyword symbols `:wrap` or `:reflect`, described below.

If *boundary* is a sequence, it may be regarded as divided into two pieces of *range* length. If it is a list of insufficient length, then missing elements are supplied as `nil` to make two *range*'s worth of elements. These two pieces then flank `sequence` on either end. The left half of *boundary* is effectively prepended to the sequence, and the right half effectively appended. When the sliding window extends beyond the boundary of *sequence* near its start or end, the window is populated from these flanking elements obtained from *boundary*.

If *boundary* argument is specified as the keyword `:wrap`, then the sequence is imagined to be flanked at either end by an infinite repetition of copies of itself. These flanks are trimmed to the window size to generate the boundary.

For instance if the sequence is `(1 2 3)` and the window size is 9 due to the value of *range* being 4, then the behavior of `:wrap` is as if *boundary* value of `(3 1 2 3 1 2 3 1)` were specified. The left flank is `(3 1 2 3)`, being the last four elements of an infinite repetition of `1 2 3`; and the right flank is similarly `(1 2 3 1)`, being the first four elements of an infinite repetition of `1 2 3`.

If *boundary* is given as the keyword `:reflect`, then the sequence is imagined to be flanked at either end by an infinite repetition of reversed copies of itself. These flanks are trimmed to the window size to generate the boundary. For instance if the sequence is `(1 2 3)` and the window size is 9 due to the value of *range* being 4, then the behavior of `:reflect` is as if *boundary* value of `(1 3 2 1 3 2 1 3)` were specified. The left flank is `(1 3 2 1)`, being the last four elements of an infinite repetition of `3 2 1`; and the right flank is similarly `(3 2 1 3)`, being the first four elements of an infinite repetition of `3 2 1`.

Examples:

```
;; change characters between angle brackets to upper case.
[window-map 1 nil (lambda (x y z)
                    (if (and (eq x #\<)
                             (eq z #\>))
                      (chr-toupper y)
                      y))
            "ab<c>de<f>g"]
--> "ab<C>de<F>g"

;; collect all numbers which are the centre element of
;; a monotonically increasing triplet
[window-mappend 1 :reflect (lambda (x y z)
                             (if (< x y z)
                               (list y)))
                '(1 2 1 3 4 2 1 9 7 5 7 8 5)]
--> (3 7)

;; calculate a moving average with a five-element
;; window, flanked by zeros at the boundaries:
[window-map 2 #(0 0 0 0)
            (lambda (. args) (/ (sum args) 5))
```

```
                #(4 7 9 13 5 1 6 11 10 3 8)]
--> #(4.0 6.6 7.6 7.0 6.8 7.2 6.6 6.2 7.6 6.4 4.2))
```

9.22.56 Function `interpose`

Syntax:

```
(interpose sep sequence)
```

Description:

The `interpose` function returns a sequence of the same type as *`sequence`*, in which the elements from *`sequence`* appear with the *`sep`* value inserted between them.

If *`sequence`* is an empty sequence or a sequence of length 1, then a sequence identical to *`sequence`* is returned. It may be a copy of *`sequence`* or it may be *`sequence`* itself.

If *`sequence`* is a character string, then the value *`sep`* must be a character.

It is permissible for *`sequence`*, or for a suffix of *`sequence`* to be a lazy list, in which case interpose returns a lazy list, or a list with a lazy suffix.

Examples:

```
(interpose #\- "xyz") -> "x-y-z"
(interpose t nil) -> nil
(interpose t #()) -> #()
(interpose #\a "") -> ""
(interpose t (range 0 0)) -> (0)
(interpose t (range 0 1)) -> (0 t 1)
(interpose t (range 0 2)) -> (0 t 1 t 2)
```

9.22.57 Functions `reduce-left` and `reduce-right`

Syntax:

```
(reduce-left binary-function list
             [init-value [key-function]])

(reduce-right binary-function list
              [init-value [key-function]])
```

Description:

The `reduce-left` and `reduce-right` functions reduce lists of operands specified by *`list`* and *`init-value`* to a single value by the repeated application of *`binary-function`*.

In the case of `reduce-left`, the *`list`* argument is required to be an object which is iterable according to the `iter-begin` function. The `reduce-right` function treats the *`list`* argument using list operations.

An effective list of operands is formed by combining *`list`* and *`init-value`*. If *`key-function`* is specified, then the items of *`list`* are mapped to new values through *`key-function`*, as if by `mapcar`. If *`init-value`* is supplied, then in the case of `reduce-left`, the effective list of operands is formed by prepending *`init-value`* to *`list`*. In the case of `reduce-right`, the effective operand list is produced by appending *`init-value`* to *`list`*. The *`init-value`* isn't mapped through *`key-function`*.

The production of the effective list can be expressed like this, though this is not to be understood as the actual implementation:

```
(append (if init-value-present (list init-value))
        [mapcar (or key-function identity) list])))
```

In the `reduce-right` case, the arguments to `append` are reversed.

If the effective list of operands is empty, then *`binary-function`* is called with no arguments at all, and its value is returned. This is the only case in which *`binary-function`* is called with no arguments; in all remaining cases, it is called with two arguments.

If the effective list contains one item, then that item is returned.

Otherwise, the effective list contains two or more items, and is decimated as follows.

Note that an *`init-value`* specified as `nil` is not the same as a missing *`init-value`*; this means that the initial value is the object `nil`. Omitting *`init-value`* is the same as specifying a value of `:` (the colon keyword symbol). It is possible to specify *`key-function`* while omitting an *`init-value`* argument. This is achieved by explicitly specifying `:` as the *`init-value`* argument.

Under `reduce-left`, the leftmost pair of operands is removed from the list and passed as arguments to *`binary-function`*, in the same order that they appear in the list, and the resulting value initializes an accumulator. Then, for each remaining item in the list, *`binary-function`* is invoked on two arguments: the current accumulator value, and the next element from the list. After each call, the accumulator is updated with the return value of *`binary-function`*. The final value of the accumulator is returned.

Under `reduce-right`, the list is processed right to left. The rightmost pair of elements in the effective list is removed, and passed as arguments to *`binary-function`*, in the same order that they appear in the list. The resulting value initializes an accumulator. Then, for each remaining item in the list, *`binary-function`* is invoked on two arguments: the next element from the list, in right to left order, and the current accumulator value. After each call, the accumulator is updated with the return value of *`binary-function`*. The final value of the accumulator is returned.

Examples:

```
;;; effective list is (1) so 1 is returned
(reduce-left (fun +) () 1 nil)  ->  1

;;; computes (- (- (- 0 1) 2) 3)
(reduce-left (fun -) '(1 2 3) 0 nil) -> -6

;;; computes (- 1 (- 2 (- 3 0)))
(reduce-right (fun -) '(1 2 3) 0 nil) -> 2

;;; computes (* 1 2 3)
(reduce-left (fun *) '((1) (2) (3)) nil (fun first)) -> 6

;;; computes 1 because the effective list is empty
;;; and so * is called with no arguments, which yields 1.
(reduce-left (fun *) nil)
```

9.22.58 Functions `some`, `all` and `none`

Syntax:

```
(some sequence [predicate-fun [key-fun]])
(all sequence [predicate-fun [key-fun]])
(none sequence [predicate-fun [key-fun]])
```

Description:

The `some`, `all` and `none` functions apply a predicate-test function *predicate-fun* over a list of elements. If the argument *key-fun* is specified, then elements of *sequence* are passed into *key-fun*, and *predicate-fun* is applied to the resulting values. If *key-fun* is omitted, the behavior is as if *key-fun* were the `identity` function. If *predicate-fun* is omitted, the behavior is as if *predicate-fun* were the `identity` function.

These functions have short-circuiting semantics and return conventions similar to the `and` and `or` operators.

The `some` function applies *predicate-fun* to successive values produced by retrieving elements of *list* and processing them through *key-fun*. If the list is empty, it returns `nil`. Otherwise it returns the first non-`nil` return value returned by a call to *predicate-fun* and stops evaluating the elements. If *predicate-fun* returns `nil` for all elements, `some` returns *nil*.

The `all` function applies *predicate-fun* to successive values produced by retrieving elements of *list* and processing them through *key-fun*. If the list is empty, it returns `t`. Otherwise, if *predicate-fun* yields `nil` for any value, the `all` function immediately returns without invoking *predicate-fun* on any more elements. If all the elements are processed, then the `all` function returns the value which *predicate-fun* yielded for the last element.

The `none` function applies *predicate-fun* to successive values produced by retrieving elements of *list* and processing them through *key-fun*. If the list is empty, it returns `t`. Otherwise, if *predicate-fun* yields non-`nil` for any value, the `none` function immediately returns `nil`. If *predicate-fun* yields `nil` for all values, the `none` function returns `t`.

Examples:

```
;; some of the integers are odd
[some '(2 4 6 9) oddp] -> t

;; none of the integers are even
[none '(1 3 4 7) evenp] -> t
```

9.22.59 Function `multi`

Syntax:

```
(multi function sequence*)
```

Description:

The `multi` function distributes an arbitrary list processing function *multi* over multiple sequences given by the *list* arguments.

The *sequence* arguments are first transposed into a single list of tuples. Each successive element of this transposed list consists of a tuple of the successive items from the lists. The length of the transposed list is that of the shortest *list* argument.

The transposed list is then passed to *function* as an argument.

The *function* is expected to produce a list of tuples, which are transposed again to produce a list of lists which is then returned.

Conceptually, the input sequences are columns and *function* is invoked on a list of the rows formed from these columns. The output of *function* is a transformed list of rows which is reconstituted into a list of columns.

Example:

```
;; Take three lists in parallel, and remove from all of them the
;; element at all positions where the third list has an element of 20.

(multi (op remove-if (op eql 20) @1 third)
       '(1 2 3)
       '(a b c)
       '(10 20 30))

-> ((1 3) (a c) (10 30))

;; The (2 b 20) "row" is gone from the three "columns".

;; Note that the (op remove if (op eql 20) @1 third)
;; expression can be simplified using the ap operator:
;;
;; (op remove-if (ap eql @3 20))
```

9.22.60 Functions `sort` and `nsort`

Syntax:

```
(sort sequence [lessfun [keyfun]])
(nsort sequence [lessfun [keyfun]])
```

Description:

The `nsort` function destructively sorts *sequence*, producing a sequence which is sorted according to the *lessfun* and *keyfun* arguments.

The *keyfun* argument specifies a function which is applied to elements of the sequence to obtain the key values which are then compared using the lessfun. If *keyfun* is omitted, the identity function is used by default: the sequence elements themselves are their own sort keys.

The *lessfun* argument specifies the comparison function which determines the sorting order. It must be a binary function which can be invoked on pairs of keys as produced by the key function. It must return a non-`nil` value if the left argument is considered to be lesser than the right argument. For instance, if the numeric function < is used on numeric keys, it produces an ascending sorted order. If the function > is used, then a descending sort is produced. If *lessfun* is omitted, then it defaults to the generic `less` function.

The `sort` function has the same argument requirements as `nsort` but is non-destructive: it returns a new object, leaving the input *sequence* unmodified, as if a copy of the input object were made using the function `copy` and then that copy were sorted in-place using `nsort`.

The `sort` and `nsort` functions are stable for sequences which are lists. This means that the

original order of items which are considered identical is preserved. For strings and vectors, `sort` is not stable.

The `sort` and `nsort` functions can be applied to hashes. It produces meaningful behavior for a hash table which contains *N* keys which are the integers from 0 to *N - 1*. Such as hash is treated as if it were a vector. The values are sorted and reassigned to sorted order to the integer keys. The behavior of `sort` is not specified for hashes whose contents do not conform to this convention.

Note: `nsort` was introduced in **TXR** 238. Prior to that version, `sort` behaved like `nsort`.

9.22.61 Function `grade`

Syntax:

```
(grade sequence [lessfun [keyfun]])
```

Description:

The `grade` function returns a list of integer indices which indicate the position of the elements of *`sequence`* in sorted order.

The *`lessfun`* and *`keyfun`* arguments behave like those of the `sort` function.

The *`sequence`* object is not modified.

The internal sort performed by `grade` is not stable. The indices of any elements considered equivalent under `lessfun` may appear in any order in the returned index sequence.

Note: the `grade` function is inspired by the "grade up" and "grade down" operators in the APL language.

Examples:

```
;; Order of the 2 3 positions of the "l"
;; characters is not specified:

[grade "Hello"] -> (0 1 2 3 4)
[grade "Hello" >] -> (4 2 3 1 0)
```

9.22.62 Functions `shuffle` and `nshuffle`

Syntax:

```
(shuffle sequence [random-state])
(nshuffle sequence [random-state])
```

Description:

The `nshuffle` function pseudorandomly rearranges the elements of *`sequence`*. This is performed in place: *`sequence`* object is modified.

The return value is *`sequence`* itself.

The rearrangement depends on pseudorandom numbers obtained from the `rand` function. The *`random-state`* argument, if present, is passed to that function.

The `nshuffle` function supports hash tables in a manner analogous to the way `nsort` supports hash tables; the same remarks apply as in the description of that function.

The `shuffle` function has the same argument requirements and semantics, but differs from `nshuffle` in that it avoids in-place modification of *`sequence`*: a new, shuffled sequence is returned, as if a copy of *`sequence`* were made using `copy` and then that copy were shuffled in-place and returned.

Note: `nshuffle` was introduced in **TXR** 238. Prior to that version, `shuffle` behaved like `nshuffle`.

9.22.63 Functions `rot` and `nrot`

Syntax:

```
(rot sequence [displacement])
(nrot sequence [displacement])
```

Description:

The `nrot` and `rot` functions rotate the elements of *`sequence`*, returning a rotated sequence.

The `nrot` function does this destructively; it modifies *`sequence`* in-place, whereas `rot` returns a new sequence without modifying the original.

The `rot` function always returns a new sequence. In cases when no rotation is performed, it copies *`sequence`* as if using the `copy` function. In cases when no rotation is performed, the `nrot` function returns the original sequence, which is unmodified.

The *`displacement`* parameter, an integer, has a default value of 1.

To rotate elements means to displace their position within the *`sequence`* by some amount, that being given by the *`displacement`* parameter, while partially preserving their circular order. Circular order means that for the purposes of rotation, the sequence is regarded to be cyclic: the first element of the sequence is considered to be the successor of the last element and vice versa. Thus, when an element is displaced past the first or last position, it wraps to the end or beginning of the sequence.

If the sequence is empty, or contains only one element, then `rot` and `nrot` terminate, performing no rotation. The following remarks apply to situations when *`sequence`* has two or more elements.

The *`displacement`* parameter, which may be negative, is first reduced to the smallest positive residue modulo the length of the sequence, resulting in a value ranging from zero to one less than the sequence length. If the resulting value is zero, then no rotation is performed.

The *`displacement`* has a negative orientation: each element's position is decreased by this amount. Those elements whose position would become negative move to the end of the sequence.

The default displacement of 1 causes the first element to become last, the second element to become first, and so forth. The opposite rotation can be obtained using -1 as the displacement.

Note: even though `nrot` operates destructively, the returned object may not be the same object as *`sequence`*. Only the returned object is required to be the rotated sequence. If this is different from the original *`sequence`* input, the contents of that original object are unspecified.

Note: the symbol `rotate` is the name of a place-mutating macro, which is much older than these functions. If `S` is a three-element sequence, then:

```
(set S (nrot S)) ;; alternatively: (upd S nrot)
```

has the same effect as:

```
(rotate [S 0] [S 1] [S 2])
```

Examples:

```
(rot "abc") -> "bca"
(rot #(1 2 3) -1) -> (3 1 2)

;; lower-case rot-13
(mapcar (relate (range #\a #\z)
                (rot (range #\a #\z) 13))
        "hello, world!")
-> "uryyb, jbeyq!"
```

9.22.64 Function `sort-group`

Syntax:

```
(sort-group sequence [keyfun [lessfun]])
```

Description:

The `sort-group` function sorts *`sequence`* according to the *`keyfun`* and *`lessfun`* arguments, and then breaks the resulting sequence into groups, based on the equivalence of the elements under *`keyfun`*.

The following equivalence holds:

```
(sort-group sq lf kf)

<-->

(partition-by kf (sort sq kf lf))
```

Note the reversed order of *`keyfun`* and *`lessfun`* arguments between `sort` and `sort-group`.

9.22.65 Function `uniq`

Syntax:

```
(uniq sequence)
```

Description:

The `uniq` function returns a sequence of the same kind as *`sequence`*, but with duplicates removed. Elements of *`sequence`* are considered equal under the `equal` function. The first occurrence of each element is retained, and the subsequent duplicates of that element, of any, are suppressed, such that the order of the elements is otherwise preserved.

The `uniq` function is an alias for the one-argument case of `unique`. That is to say, this equivalence holds:

```
(uniq s) <--> (unique s)
```

9.22.66 Function `unique`

Syntax:

```
(unique sequence [keyfun {hash-arg}*])
```

Description:

The `unique` function is a generalization of `uniq`. It returns a sequence of the same kind as *`sequence`*, but with duplicates removed.

If neither *`keyfun`* nor *`hash-arg`*s are specified, then elements of sequence are considered equal under the `equal` function. The first occurrence of each element is retained, and the subsequent duplicates of that element, of any, are suppressed, such that the order of the elements is otherwise preserved.

If *`keyfun`* is specified, then that function is applied to each element, and the resulting values are compared for equality. In other words, the behavior is as if *`keyfun`* were the `identity` function.

If one or more *`hash-arg`*s are present, these specify the arguments for the construction of the internal hash table used by `unique`. The arguments are like those of the `hash` function.

9.22.67 Function `tuples`

Syntax:

```
(tuples length sequence [fill-value])
```

Description:

The `tuples` function produces a lazy list which represents a reorganization of the elements of *`sequence`* into tuples of *`length`*, where *`length`* must be a positive integer.

The length of the sequence might not be evenly divisible by the tuple length. In this case, if a *`fill-value`* argument is specified, then the last tuple is padded with enough repetitions of *`fill-value`* to make it have *`length`* elements. If *`fill-value`* is not specified, then the last tuple is left shorter than *`length`*.

The output of the function is a list, but the tuples themselves are sequences of the same kind as *`sequence`*. If *`sequence`* is any kind of list, they are lists, and not lazy lists.

Examples:

```
(tuples 3 #(1 2 3 4 5 6 7 8) 0) -> (#(1 2 3) #(4 5 6)
                                    #(7 8 0))
(tuples 3 "abc") -> ("abc")
(tuples 3 "abcd") -> ("abc" "d")
(tuples 3 "abcd" #\z) -> ("abc" "dzz")
(tuples 3 (list 1 2) #\z) -> ((1 2 #\z))
```

9.22.68 Function `tuples*`

Syntax:

```
(tuples* length sequence [fill-value])
```

Description:

The `tuples*` function produces a lazy list of overlapping tuples taken from *`sequence`*. The

length of the tuples is given by the *length* argument.

The *length* argument must be a positive integer.

Tuples are subsequences of consecutive items from the input *sequence*, beginning with consecutive elements. The first tuple in the returned list begins with the first item of *sequence*; the second tuple begins with the second item, and so forth.

The output of the function is a list, but the tuples themselves are sequences of the same kind as *sequence*. If *sequence* is any kind of list, they are lists, and not lazy lists.

If *sequence* is shorter than *length* then it contains no tuples of that length. In this case, if no *fill-value* argument is specified, then the empty list is returned. In this same situation, if *fill-value* is specified, then a one-element list is returned, consisting of a tuple of the required length, consisting of the elements from *sequence* followed by repetitions of *fill-value*, which must be of a type suitable as an element of the sequence. The *fill-value* is otherwise ignored.

Examples:

```
(tuples* 1 "abc") -> ("a" "b" "c")
(tuples* 2 "abc") -> ("ab" "bc")
(tuples* 3 "abc") -> ("abc")
(tuples* 4 "abc") -> nil
(tuples* 4 "abc" #)-> ("abcz")
(tuples* 6 "abc" #)-> ("abczzz")
(tuples* 6 "abc" 4) -> error
(tuples* 2 '(a b c)) -> ((a b) (b c))
(take 3 (tuples* 3 0)) -> ((0 1 2) (1 2 3) (2 3 4))
```

9.22.69 Function `partition-by`

Syntax:

```
(partition-by function sequence)
```

Description:

If *sequence* is empty, then `partition-by` returns an empty list, and *function* is never called.

Otherwise, `partition-by` returns a lazy list of partitions of the sequence *sequence*. Partitions are consecutive, nonempty substrings of *sequence*, of the same kind as *sequence*.

The partitioning begins with the first element of *sequence* being placed into a partition.

The subsequent partitioning is done according to *function*, which is applied to each element of *sequence*. Whenever, for the next element, the function returns the same value as it returned for the previous element, the element is placed into the same partition. Otherwise, the next element is placed into, and begins, a new partition.

The return values of the calls to *function* are compared using the `equal` function.

Examples:

```
[partition-by identity '(1 2 3 3 4 4 4 5)] -> ((1) (2) (3 3)
```

```
                                                    (4 4 4) (5))

(partition-by (op = 3) #(1 2 3 4 5 6 7)) -> (#(1 2) #(3)
                                             #(4 5 6 7))
```

9.22.70 Function `partition-if`

Syntax:

```
(partition-if function iterable [count])
```

Description:

The `partition-if` function separates the *`iterable`* sequence into partitions which are identified by the two-argument *`function`*. The principal idea is that successive overlapping pairs from *`iterable`* are passed as arguments to *`function`*, and whenever *`function`* yields true, those elements are identified as belonging to separate partitions: a partitioning division shall take place between them. The detailed semantics is given below, as a procedure.

Firstly, if *`sequence`* is empty, then `partition-if` returns an empty list, and *`function`* is never called.

Otherwise, `partition-if` returns a lazy list of partitions of *`iterable`*. Partitions are consecutive, nonempty substrings of *`iterable`*, of the same kind as *`iterable`*.

The partitioning begins with the first element of *`iterable`* being placed into the first partition.

The subsequent partitioning is done according to a Boolean *`function`*, which must accept two arguments. Whenever the function yields true, it indicates that a partition is to be terminated and a new partition to begin.

The *`count`* argument, if present, must be a nonnegative integer. It indicates a limit on how many partitions will be delimited; after this limit is reached, the remainder of the *`iterable`* sequence is placed into a single partition.

After the first element is placed into a partition, the following partition-building process is repeated until the partition is terminated.

1. If *`iterable`* contains no more elements, then the partition terminates.
2. Otherwise, if the *`count`* is present, and has a value of zero, then the next available element is unconditionally deposited into the current partition, and the process repeats from step 1.
3. Otherwise, *`function`* is invoked on two values: the previous element which has most recently been deposited into the partition, and its successor from *`iterable`*.
4. If *`function`* returns `nil`, then the partition continues: the next element is added to the partition, and the process repeats from step 1.
5. Otherwise, *`function`* has returned true and the partition is terminated. In this case, if *`count`* is present, it is decremented.

When the current partition is terminated, it is converted to a sequence of the same kind as *`iterable`* as if by using the `make-like` function, and incorporated as the next element of the lazy list of partitions.

If, after a partition is thus produced, a next element is available, it is placed into a new partition, and the above partition-building process takes place from step 1. Otherwise, the lazy list terminates.

Examples:

```
;; Start new partition for unequal characters.
[partition-if neql "aaaabbcdee"] -> ("aaaa" "bb" "c" "d" "ee")

;; As above, but partition only twice
[partition-if neql "aaaabbcdee" 2] -> ("aaaa" "bb" "cdee")

;; Start new partition when non-digit follows digit:
[partition-if (do and
                (chr-isdigit @1)
                (not (chr-isdigit @2)))
              "a13cd9foo42z"]
-> ("a13" "cd9" "foo42" "z")

;; Place ascending runs of consecutive integers
;; into partitions. I.e. start a partition whenever the
;; difference from the previous element isn't 1:
(partition-if (op /= (- @2 @1) 1)
              '(1 3 4 5 7 8 9 10 9 8 6 5 3 2))
-> ((1) (3 4 5) (7 8 9 10) (9) (8) (6) (5) (3) (2))

;; Place runs of adjacent integers into partitions.
;; I.e. start a new partition if the the absolute value of
;; the difference from the previous exceeds 1:
(partition-if (op > (abs (- @2 @1)) 1)
              '(1 3 4 5 7 8 9 10 9 8 6 5 3 2))
-> ((1) (3 4 5) (7 8 9 10 9 8) (6 5) (3 2))
```

9.23 Open Sequence Traversal

Functions in this category perform efficient traversal of sequences.

There are two flavors of these functions: functions in the `iter-begin` group, and functions in the `seq-begin` group. The latter are obsolescent.

User-defined iteration is possible via defining special methods on structures. An object supports iteration by defining the special method `iter-begin` which is different from the `iter-begin` function. This special function returns an iterator object which supports special methods `iter-item`, `iter-more` and `iter-step`. Two protocols are supported, one of which is more efficient by eliminating the `iter-more` method. Details are specified in the section **Special Structure Functions**.

9.23.1 Function `iter-begin`

Syntax:

(`iter-begin` *`seq`*)

Description:

The `iter-begin` function returns an iterator object suitable for traversing the elements of the sequence denoted by the *`seq`* object.

If *`seq`* is a list-like sequence, then `iter-begin` may return *`seq`* itself as the iterator. Likewise if *`seq`* is a number.

If *`seq`* is an iterator produced by `iter-begin` then an iterator similar to that iterator is returned,

which can continue iterating the same sequence. The iterator may be *seq* itself or share state with *seq*, and thus may not be relied on to produce an independent, parallel iteration.

If *seq* is a structure which supports the `iter-begin` method, then that method is called and its return value is returned. A structure which does not support this method is possibly considered to be a sequence according to the usual criteria, based on whether it supports the `nullify`, `length` or `car` methods. An struct object supporting none of these methods is deemed not iterable.

In all other cases, if *seq* is iterable, an object of type `seq-iter` is returned.

Range objects are iterable if they are numeric. A range consisting of two strings may also be iterable, as described below.

A range is considered to be a numeric or character range if the `from` element is a number or character. The `to` is then required to be either a value which is comparable with that number or character using the < function, or else it must be one of the two objects `t` or `:`, either of which indicate that the range is unbounded. In this unbounded range case, the expressions `(iter-begin X..:)` and `(iter-begin X..t)` are equivalent to `(iter-begin X)`. Numeric ranges are half-open: the `to` value of ascending ranges is excluded, as is the `from` value of descending ranges, so that `0..10` steps through the values `0` through `9`, and `10..0` steps through the same values in reverse order.

A string range consists of two strings of equal length. If the strings are of unequal length, an error exception is thrown. For the range to operate as intended, the strings must meet some additional requirements. If the `from` string is lexicographically lesser than the `to` string, as determined by the `less` function, then the range is ascending, otherwise it is descending. The string range iterates by incrementing (or decrementing, in the case of a descending range) the characters of the `from` string until they are equal to those of the `to` string. The last character has priority. For instance, the range `"AA"."CC"` iterates over the strings `AA`, `AB`, `AC`, `BA`, `BB`, `BC`, `CA`, `CB` and `CC`. The descending range `"CC"."AA"` iterates over the same strings, in reverse order. Whenever the incrementing character attains the value of the corresponding character in the `to` string, that character is reset to its starting value, and its left neighbor, if it exists, is incremented instead. If no left neighbor exists, the iteration terminates.

Search trees are iterable. Iteration entails an in-order visits of the elements of a tree. A tree iterator created by `tree-begin` is also iterable. It is unspecified whether iteration over a `tree-iter` object modifies that object to perform the traversal, or whether it uses a copy of the iterator.

If `seq` is not an iterable object, an error exception is thrown.

9.23.2 Function `iter-more`

Syntax:

```
(iter-more iter)
```

Description:

The `iter-more` function returns `t` if there remain more elements to be traversed. Otherwise it returns `nil`.

The *iter* argument must be a valid iterator returned by a call to *iter-begin*, *iter-step* or *iter-reset*.

The `iter-more` function doesn't change the state of *iter*.

If `iter` is the object `nil` then `nil` is returned. Note: the `iter-begin` may return `nil` if its argument is `nil` or any empty sequence, or an empty range (a range whose `to` and `from` fields are the same number or character).

If *`iter`* is a `cons` cell, then `iter-more` returns `t`.

If *`iter`* is a number, then `iter-more` returns `t`. This is the case even if calculating the successor of that number isn't possible due to floating-point overflow or insufficient system resources.

If *`iter`* is a character, then `iter-more` returns `t` if *`iter`* isn't the highest possible character code, otherwise `nil`.

If *`iter`* was formed from a descending range, meaning that `iter-begin` was invoked on a range with a `from` fielding exceeding its `to` value, then `iter-begin` returns true while the current iterator value is greater than the the limiting value given by the `to` field. For an ascending range, it returns true if the current iterator value is lower than the limiting value. However, note the peculiar semantics of `iter-item` with regard to descending range iteration.

If *`iter`* is a structure, then if it supports an `iter-more` method, then that method is called with no arguments, and its return value is returned. If the structure does not have an `iter-more` method, then `t` is returned.

9.23.3 Function `iter-item`

Syntax:

```
(iter-item iter)
```

Description:

If the `iter-more` function indicates that more items remain to be visited, then the next item can be retrieved using `iter-item`.

The *`iter`* argument must be a valid iterator returned by a call to *`iter-begin`*, *`iter-step`* or *`iter-reset`*.

The `iter-more` function doesn't change the state of *`iter`*.

If `iter-more` is invoked on an iterator which indicates that no more items remain to be visited, the return value is `nil`.

If *`iter`* is a `cons` cell, then `iter-item` returns the `car` field of that cell.

If *`iter`* is a character or number, then `iter-item` returns that character or number itself.

If *`iter`* is based on an ascending numeric or character range, then `iter-item` returns the current iteration value, which is initialized by `iter-begin` as a copy of the range's `from` field. Thus, the range `0..3` traverses the values `0`, `1` and `2`, excluding the `3`.

If *`iter`* is based on a descending numeric or character range, then `iter-item` returns the predecessor of the current iteration value, which is initialized `iter-begin` as a copy of the range's `from` field. Thus, the range `3..0` traverses the values `2`, `1` and `0`, excluding the `3`: exactly the same values are visited as for the range `0..3` only in reverse order.

If *`iter`* is a structure which supports the `iter-item` method, then that method is called and its return value is returned.

9.23.4 Function `iter-step`

Syntax:

```
(iter-step iter)
```

Description:

If the `iter-more` function indicates that more items remain to be visited, then the `iter-step` function may be used to consume the next item.

The function returns an iterator denoting the traversal of the remaining items in the sequence.

The *`iter`* argument must be a valid iterator returned by a call to *`iter-begin`*, *`iter-step`* or *`iter-reset`*.

The `iter-step` function may return a new object, in which case it avoids changing the state of *`iter`*, or else it may change the state of *`iter`* and return it.

If the application discontinues the use of *`iter`*, and continues the traversal using the returned iterator, it will work correctly in either situation.

If `iter-step` is invoked on an iterator which indicates that no more items remain to be visited, the return value is unspecified.

If *`iter`* is a `cons` cell, then `iter-step` returns the `cdr` field of that cell. That value must itself be a `cons` or else `nil`, otherwise an error is thrown. This is to prevent iteration from wrongly iterating into the non-null terminators of improper lists. Without this rule, iteration of a list like `(1 2 . 3)` would reach the `cons` cell `(2 . 3)` at which point a subsequent `iter-step` would return the `cdr` field `3`. But that value is a valid iterator which will then continue by stepping through `4`, `5` and so on.

If *`iter`* is a list-like sequence, then `cdr` is invoked on it and that value is returned. The value must also be a list-like sequence, or else `nil`. The reasoning for this is the same as for the similar restriction imposed in the case when *`iter`* is a `cons`.

If *`iter`* is a character or number, then `iter-step` returns its successor, as if using the `succ` function.

If *`iter`* is a structure which supports the `iter-step` method, then that method is called and its return value is returned.

9.23.5 Function `iter-reset`

Syntax:

```
(iter-reset iter seq)
```

Description:

The `iter-reset` function returns an iterator object specialized for the task of traversing the sequence *`seq`*.

If it is possible for *`iter`* to be that object, then the function may adjust the state of *`iter`* and return it.

If `iter-reset` doesn't use *`iter`*, then it behaves exactly like `iter-begin` being invoked on *`seq`*.

If *seq* is a structure which supports the `iter-reset` method, then that method is called and its return value is returned. Note the reversed arguments. The `iter-reset` method is of the *seq* object, not of *iter*. That is to say, the call `(iter-reset iter obj)` results in the *obj*`.(iter-reset iter)` call. If *seq* is a structure which doesn't support `iter-reset` then *iter* is ignored, `iter-begin` is invoked on *seq* and the result is returned.

9.23.6 Function `seq-begin`

Syntax:

```
(seq-begin object)
```

Description:

The obsolescent `seq-begin` function returns an iterator object specialized to the task of traversing the sequence represented by the input *object*.

If *object* isn't a sequence, an exception is thrown.

Note that if *object* is a lazy list, the returned iterator maintains a reference to the head of that list during the traversal; therefore, generic iteration based on iterators from `seq-begin` is not suitable for indefinite iteration over infinite lists.

9.23.7 Function `seq-next`

Syntax:

```
(seq-next iter end-value)
```

Description:

The obsolescent `seq-next` function retrieves the next available item from the sequence iterated by *iter*, which must be an object returned by `seq-begin`.

If the sequence has no more items to be traversed, then *end-value* is returned instead.

Note: to avoid ambiguities, the application should provide an *end-value* which is guaranteed distinct from any item in the sequence, such as a freshly allocated object.

9.23.8 Function `seq-reset`

Syntax:

```
(seq-reset iter object)
```

Description:

The obsolescent `seq-reset` reinitializes the existing iterator object *iter* to begin a new traversal over the given *object*, which must be a value of a kind that would be a suitable argument for `seq-begin`.

The `seq-reset` function returns *iter*.

9.24 Procedural List Construction

TXR Lisp provides an a structure type called `list-builder` which encapsulates state and methods for constructing lists procedurally. Among the advantages of using `list-builder` is that lists can be constructed in the left-to-right direction without requiring multiple traversals or reversal. For example, `list-builder` naturally combines with iteration or recursion: items visited in an iterative or recursive process can be collected easily using `list-builder` in the order they are visited.

The `list-builder` type provides methods for adding and removing items at either end of the list, making it suitable where a *dequeue* structure is required.

The basic workflow begins with the instantiation of a `list-builder` object. This object may be initialized with a piece of list material which begins the to-be-constructed list, or it may be initialized to begin with an empty list. Methods such as `add` and `pend` are invoked on this object to extend the list with new elements. At any point, the list constructed so far is available using the `get` method, which is also how the final version of the list is eventually retrieved.

The `list-builder` methods which add material to the list all return the list builder, making chaining possible.

```
(new list-builder).(add 1).(add 2).(pend '(3 4 5)).(get)
-> (1 2 3 4 5)
```

The `build` macro is provided which syntactically streamlines the process. It implicitly creates a `list-builder` instance and binds it to a hidden lexical variable. It then evaluates forms in a lexical scope in which shorthand macros are available for building the list.

9.24.1 Structure `list-builder`

Syntax:

```
(defstruct list-builder nil
  head tail)
```

Description:

The `list-builder` structure encapsulates the state for a list building process. Programs should use the `build-list` function for creating an instance of `list-builder`. The `head` and `tail` slots should be regarded as internal variables.

9.24.2 Function `build-list`

Syntax:

(`build-list` [*`initial-list`*])

Description:

The `build-list` function instantiates and returns an object of struct type `list-builder`.

If no *`initial-list`* argument is supplied, then the object is implicitly initialized with an empty list.

If the argument is supplied, then it is equivalent to calling `build-list` without an argument to produce an object *`obj`* by invoking the method call *`obj`*`.(ncon` *`initial-list`*`)` on this object. The object produced by the expression *`list`* is installed (without being copied) into the object as the prefix of the list to be constructed.

The *`initial-list`* argument can be a sequence other than a list.

Example:

```
;; build the list (a b) trivially

(let ((lb (build-list '(a b))))
  lb.(get)
```

```
-> (a b)
```

9.24.3 Methods `add` and `add*`

Syntax:

```
list-builder.(add element*)
list-builder.(add* element*)
```

Description:

The `add` and `add*` methods extend the list being constructed by a `list-builder` object by adding individual elements to it. The `add` method adds elements at the tail of the list, whereas `add*` adds elements at the front.

These methods return the *`list-builder`* object.

The precise semantics is as follows. All of the *`element`* arguments are combined into a list as if by the `list` function, and the resulting list combined with the current contents of the `list-builder` object as if using the `append` function. The resulting list becomes the new contents.

Examples:

```
;; Build the list (1 2 3 4)

(let ((lb (build-list)))
  lb.(add 3 4)
  lb.(add* 1 2)
  lb.(get))
-> (1 2 3 4)

;; Add "c" to "abc"
;; same semantics as (append "abc" #\c)

(let ((lb (build-list "ab")))
  lb.(add #\c)
  lb.(get))
-> "abc"
```

9.24.4 Methods `pend` and `pend*`

Syntax:

```
list-builder.(pend list*)
list-builder.(pend* list*)
```

Description:

The `pend` and `pend*` methods extend the list being constructed by a `list-builder` object by adding lists to it. The `pend` method catenates the *`list`* arguments together as if by the `append` function, then appends the resulting list to the end of the list being constructed. The `pend*` method is similar, except it prepends the catenated lists to the front of the list being constructed.

The `pend` and `pend*` operations do not mutate the input lists, but may cause the resulting list to share structure with the input lists.

These functions may mutate the list already contained in *`list-builder`*; however, they avoid mutating those parts of the current list that are shared with inputs that were given in earlier calls to

these functions.

These methods return the *list-builder* object.

Example:

```
;; Build the list (1 2 3 4)

(let ((lb (build-list)))
  lb.(pend '(3 4))
  lb.(pend* '(1 2))
  lb.(get))
-> (1 2 3 4)
```

9.24.5 Methods `ncon` and `ncon*`

Syntax:

```
list-builder.(ncon list*)
list-builder.(ncon* list*)
```

Description:

The `ncon` and `ncon*` methods extend the list being constructed by a `list-builder` object by adding lists to it. The `ncon` method destructively catenates the *list* arguments as if by the `nconc` function. The resulting list is appended to the list being constructed. The `ncon*` method is similar, except it prepends the catenated lists to the front of the list being constructed.

These methods may destructively manipulate the list already contained in the *list-builder* object, and likewise may destructively manipulate the input lists. They may cause the list being constructed to share substructure with the input lists.

Additionally, these methods may destructively manipulate the list already contained in the *list-builder* object without regard for shared structure between that list and inputs given earlier any of the `pend`, `pend*`, `ncon` or `ncon*` functions.

The `ncon*` function can be called with a single argument which is an atom. This atom will simply be installed as the terminating atom of the list being constructed, if the current list is an ordinary list.

These methods return the *list-builder* object.

Example:

```
;; Build the list (1 2 3 4 . 5)

(let ((lb (build-list)))
  lb.(ncon* (list 1 2))
  lb.(ncon (list 3 4))
  lb.(ncon 5)
  lb.(get))
-> (1 2 3 4 . 5)
```

9.24.6 Method `oust`

Syntax:

```
list-builder.(oust list*)
```

Description:

The `oust` method discards the list constructed so far, optionally replacing it with new material.

The `oust` method catenates the *`list`* arguments together as if by the `append` function. The resulting list, which is empty if there are no *`list`* arguments, then replaces the object's list constructed so far.

The `oust` method returns the *`list-builder`* object.

Examples:

```
;; Build the list (3 4) by first building (1 2),
;; then discarding that and adding 3 and 4:

(let ((lb (build-list)))
  lb.(add 1 2)
  lb.(oust)
  lb.(add 3 4)
  lb.(get))
-> (3 4)

;; Build the list (3 4 5 6) by first building (1 2),
;; then replacing with catenation of (3 4) and (5 6):
(let ((lb (build-list)))
  lb.(pend '(1 2))
  lb.(oust '(3 4) '(5 6))
  lb.(get))
-> (3 4 5 6)
```

9.24.7 Method `get`

Syntax:

```
list-builder.(get)
```

Description:

The `get` method retrieves the list constructed so far by a `list-builder` object. It doesn't change the state of the object. The retrieved list may be passed as an argument into the construction methods on the same object.

Examples:

```
;; Build the circular list (1 1 1 1 ...)
;; by appending (1) to itself destructively:

(let ((lb (build-list '(1))))
  lb.(ncon* lb.(get))
  lb.(get))
-> (1 1 1 1 ...)

;; build the list (1 2 1 2 1 2 1 2)
;; by doubling (1 2) twice:
```

```
(let ((lb (build-list)))
  lb.(add 1 2)
  lb.(pend lb.(get))
  lb.(pend lb.(get))
  lb.(get))
-> (1 2 1 2 1 2 1 2)
```

9.24.8 Methods `del` and `del*`

Syntax:

```
list-builder.(del)
list-builder.(del*)
```

Description:

The `del` and `del*` methods each remove an element from the list and return it. If the list is empty, they return `nil`.

The `del` method removes an element from the front of the list, whereas `del*` removes an element from the end of the list.

Note: this orientation is opposite to `add` and `add*`. Thus `del` pairs with `add` to produce FIFO queuing behavior.

9.24.9 Macros `build` and `buildn`

Syntax:

```
(build form*)
(buildn form*)
```

Description:

The `build` and `buildn` macros provide a shorthand notation for constructing lists using the `list-builder` structure. They eliminate the explicit call to the `build-list` function to construct the object, and eliminate the explicit references to the object.

Both of these macros create a lexical environment in which a `list-builder` object is implicitly constructed and bound to a hidden variable. This lexical environment also provides local functions named `add`, `add*`, `pend`, `pend*`, `ncon`, `ncon*`, `oust`, `get`, `del` and `del*`, which mimic the `list-builder` methods, but operate implicitly on this hidden variable, so that the object need not be mentioned as an argument. With the exception of `get`, `del` and `del*`, the local functions return `nil`, unlike like the same-named `list-builder` methods, which return the `list-builder` object.

In this lexical environment, each *form* is evaluated in order.

When the last *form* is evaluated, `build` returns the constructed list, whereas `buildn` returns the value of the last *form*.

If no forms are enclosed, both macros return `nil`.

Note: because the local function `del` has the same name as a global macro, it is implemented as a `macrolet`. Inside a `build` or `buildn`, if `del` is invoked with no arguments, then it denotes a call to the `list-builder del` method. If invoked with an argument, then it resolves to the global `del` macro for deleting a place.

Examples:

```
;; Build the circular list (1 1 1 1 ...)
;; by appending (1) to itself destructively:

(build
  (add 1)
  (ncon* (get))) -> (1 1 1 1 ...)

;; build the list (1 2 1 2 1 2 1 2)
;; by doubling (1 2) twice:

(build
  (add 1 2)
  (pend (get))
  (pend (get))) -> (1 2 1 2 1 2 1 2)

;; build a list by mapping over the local
;; add function:

(build [mapdo add (range 1 3)]) -> (1 2 3)

;; breadth-first traversal of nested list;
(defun bf-map (tree visit-fn)
  (buildn
    (add tree)
    (whilet ((item (del)))
      (if (atom item)
        [visit-fn item]
        (each ((el item))
          (add el))))))

(let (flat)
  (bf-map '(1 (2 (3 4 (5))) ((6 7) 8)) (do push @1 flat))
   (nreverse flat))
-> (1 2 8 3 4 6 7 5)
```

9.25 Permutations and Combinations

9.25.1 Function `perm`

Syntax:

```
(perm seq [len])
```

Description:

The `perm` function returns a lazy list which consists of all length *len* permutations of formed by items taken from *seq*. The permutations do not use any element of *seq* more than once.

Argument *len*, if present, must be a positive integer, and *seq* must be a sequence.

If *len* is not present, then its value defaults to the length of *seq*: the list of the full permutations of the entire sequence is returned.

The permutations in the returned list are sequences of the same kind as `seq`.

If *len* is zero, then a list containing one permutation is returned, and that permutation is of zero length.

If *len* exceeds the length of *seq*, then an empty list is returned, since it is impossible to make a single nonrepeating permutation that requires more items than are available.

The permutations are lexicographically ordered.

9.25.2 Function `rperm`

Syntax:

```
(rperm seq len)
```

Description:

The `rperm` function returns a lazy list which consists of all the repeating permutations of length *len* formed by items taken from *seq*. "Repeating" means that the items from *seq* can appear more than once in the permutations.

The permutations which are returned are sequences of the same kind as *seq*.

Argument *len* must be a nonnegative integer, and *seq* must be a sequence.

If *len* is zero, then a single permutation is returned, of zero length. This is true regardless of whether *seq* is itself empty.

If *seq* is empty and *len* is greater than zero, then no permutations are returned, since permutations of a positive length require items, and the sequence has no items. Thus there exist no such permutations.

The first permutation consists of *le* repetitions of the first element of *seq*. The next repetition, if there is one, differs from the first repetition in that its last element is the second element of *seq*. That is to say, the permutations are lexicographically ordered.

Examples:

```
(rperm "01" 3) -> ("000" "001" "010" "011"
                   "100" "101" "110" "111")

(rperm #(1) 3) -> (#(1 1 1))

(rperm '(0 1 2) 2) -> ((0 0) (0 1) (0 2) (1 0)
                       (1 1) (1 2) (2 0) (2 1) (2 2))
```

9.25.3 Function `comb`

Syntax:

```
(comb seq len)
```

Description:

The `comb` function returns a lazy list which consists of all length *len* nonrepeating combinations formed by taking items taken from *seq*. "Nonrepeating combinations" means that the combinations do not use any element of *seq* more than once. If *seq* contains no duplicates, then the combinations contain no duplicates.

Argument *len* must be a nonnegative integer, and *seq* must be a sequence or a hash table.

The combinations in the returned list are objects of the same kind as *seq*.

If *len* is zero, then a list containing one combination is returned, and that combination is of zero length.

If *len* exceeds the number of elements in *seq*, then an empty list is returned, since it is impossible to make a single nonrepeating combination that requires more items than are available.

If *seq* is a sequence, the returned combinations are lexicographically ordered. This requirement is not applicable when *seq* is a hash table.

Example:

```
;; powerset function, in terms of comb.
;; Yields a lazy list of all subsets of s,
;; expressed as sequences of the same type as s.

(defun powerset (s)
  (mappend* (op comb s) (range 0 (length s))))
```

9.25.4 Function `rcomb`

Syntax:

```
(rcomb seq len)
```

Description:

The `comb` function returns a lazy list which consists of all length *len* repeating combinations formed by taking items taken from *seq*. "Repeating combinations" means that the combinations can use an element of *seq* more than once.

Argument *len* must be a nonnegative integer, and *seq* must be a sequence.

The combinations in the returned list are sequences of the same kind as *seq*.

If *len* is zero, then a list containing one combination is returned, and that combination is of zero length. This is true even if *seq* is empty.

If *seq* is empty, and *len* is nonzero, then an empty list is returned.

The combinations are lexicographically ordered.

9.26 Macros

Because **TXR Lisp** supports structural macros, **TXR** processes **TXR Lisp** expressions in two separate phases: the expansion phase and the evaluation/compilation phase. During the expansion phase, a top-level expression is recursively traversed, and all macro invocations in it are expanded. The result is a transformed expression which contains only function calls and invocations of special operators. This expanded form is then evaluated or compiled, depending on the situation.

Macro invocations are compound forms and whose operator symbol has a macro definition in scope. A macro definition is a kind of function which operates on syntax during macro-expansion, called upon to

calculate a transformation of the syntax. The return value of a macro replaces its invocation, and is traversed to look for more opportunities for macro expansion. Macros differ from ordinary functions in three ways: they are called at macro-expansion time, they receive pieces of unevaluated syntax as their arguments, and their parameter lists are macro parameter lists which support destructuring, as well as certain special parameters.

TXR Lisp also supports symbol macros. A symbol macro definition associates a symbol with an expansion. When that symbol appears as a form, the macro-expander replaces it with the expansion.

TXR source files are treated somewhat differently with regard to macro expansion compared to **TXR Lisp**. When **TXR Lisp** forms are read from a file by `load` or `compile` or read by the interactive listener, each form is expanded and evaluated or compiled before the subsequent form is processed. In contrast, when a **TXR** file is loaded, expansion of the Lisp forms are its arguments takes place during the parsing of the entire source file, and is complete for the entire file before any of the code is executed.

9.26.1 Macro parameter lists

TXR macros support destructuring, similarly to Common Lisp macros. This means that macro parameter lists are like function argument lists, but support nesting. A macro parameter list can specify a nested parameter list in every place where an argument symbol may appear. For instance, consider this macro parameter list:

```
((a (b c)) : (c frm) ((d e) frm2 de-p) . g)
```

The top-level of this nested form has the structure

```
(I : J K . L)
```

in which we can identify the major constituent positions as *I*, *J*, *K* and *L*.

The constituent at position *I* is the mandatory parameter `(a (b c))`. Position *J* holds the optional parameter `c` (with default init form `frm`). At *K* is found the optional parameter `(d e)` (with default init form `frm2` and presence-indicating variable `de-p`). Finally, the parameter in the dot position *L* is `g`, which captures trailing arguments.

Obviously, some of the parameters are compound expressions rather than symbols: `(a (b c))` and `(d e)`. These compounds express nested macro parameter lists.

Starting in **TXR** 285, the symbol `t` can be used in a macro parameter list in place of a parameter name. This indicates that an object is expected at that position in the corresponding structure, but no variable will be bound. For completeness, the `t` symbol may also be used for a presence-indicating variable. When the name of an optional parameter is specified as `t`, and the corresponding structure is missing, the *`init-val`* expression, if present, is still evaluated under the same circumstances as it would if a variable were present.

Nested macro parameter lists recursively match the corresponding structure in the argument object. For instance if a simple argument would capture the structure `(1 (2 3))` then we can replace the argument with the nested argument list `(a (b c))` which destructures the `(1 (2 3))` such that the parameters `a`, `b` and `c` will end up bound to `1`, `2` and `3`, respectively.

Nested macro parameter lists have all the features of the top-level macro parameter lists: they can have optional arguments with default values, use the dotted position, and contain the `:env`, `:whole` and `:form` special parameters, which are described below. In nested parameter lists, the binding strictness is relaxed for optional parameters. If `(a (b c))` is optional, and the argument is, say, `(1)`, then `a` gets `1`, and `b` and `c` receive `nil`.

Macro parameter lists also supports three special keywords, namely `:env`, `:whole` and `:form`.

The parameter list `(:whole x :env y :form z)` will bind parameter `x` to the entire macro parameter list, bind parameter `y` to the macro environment and bind parameter `z` to the entire macro form (the original compound form used to invoke the macro).

The `:env`, `:whole` and `:form` notations can occur anywhere in a macro parameter list, other than to the right of the consing dot. They can be used in nested macro parameter lists also. Note that in a nested macro parameter list, `:form` and `:env` do not change meaning: they bind the same object as they would in the top-level of the macro parameter list. However the `:whole` parameter inside has a restricted scope in a nested parameter list: its parameter will capture just that part of the argument material which matches that parameter list, rather than the entire argument list.

The processing of macro parameter lists omits the feature that when the `:` (colon) keyword symbol is given as the argument to an optional parameter, that argument is treated as a missing argument. This special logic is implemented only in the function argument passing mechanism, not in the binding of macro parameters to object structure. If the colon symbol appears in the object structure and is matched against an optional parameter, it is an ordinary value. That parameter is considered present, and takes on the colon symbol as its value.

Dialect Note:

> In ANSI Common Lisp, the lambda list keyword `&whole` binds its corresponding variable to the entire macro form, whereas **TXR Lisp**'s `:whole` binds its variable only to the arguments of the macro form.
>
> Note, however, that ANSI CL distinguishes between destructuring and macro lambda lists, and the `&whole` parameter has a different behavior in each. Under `destructuring-bind`, the `&whole` parameter receives just the arguments, just like the behavior of **TXR Lisp**'s `:whole` parameter.
>
> **TXR Lisp** does not distinguish between destructuring and macro lambda lists; they are the same and behave the same way. Thus `:whole` is treated the same way in macros as in `tree-bind` and related binding operators: it binds just the arguments to the parameter. **TXR Lisp** has the special parameter `:form` by means of which macros can access their invoking form. This parameter is also supported in `tree-bind` and binds to the entire `tree-bind` form.
>
> ANSI CL doesn't support the convention that the `t` symbol may appear instead of a parameter symbol to suppress the binding of a variable.

9.26.2 The Macro Expansion Process

The following description omits the treatment of top-level forms by `eval` and the compiler. This is described, respectively, in the description of `eval` and the section Top-Level Forms inside the LISP COMPILATION chapter. Certain other details are also omitted, such as the dynamic evolution of the macro-time environment, the expansion of macrolet forms.

Macro expansion is, generally speaking, a recursive process. The expression to be expanded is classified into cases, and as necessary, the constituent expressions are recursively expanded, depending on these cases. Certain aspects of the process may be regarded as iterative. Macro expansion maintains a macro-time lexical environment which is extended and contracted as the expander descends into various nested binding constructs.

The expander may encounter a bindable symbol. If such a symbol has a binding as a symbol macro, then it is replaced by its expansion, and the expander iterates on the resulting form. The form may be another

object, including a symbol. If it is the same symbol, than macro expansion terminates; the symbol remains unsubstituted. Symbols are treated differently by the expander if they are in the Lisp-1-style context of the `dwim` operator, or the equivalent square bracket notation. The expander takes into consideration the semantics of the combined function and variable namespace.

The expander may encounter a compound form headed by a symbol which has a macro binding. In this situation, the macro expander function is called, and the form is replaced by the resulting form. That form is considered again as a potential macro. In any case, the expander makes a note that it has expanded a macro,

If a form isn't a macro, then it's either a function call, special from or an atomic form: a symbol (that has no binding as a symbol macro) or other atom. The interesting cases are special forms and function calls, since the atomic forms are simply returned as-is without expansion. Special forms and function call forms contain other forms, some or all of which require expansion. The expander recognizes the shape of each special form or function call, pulls out the constituent expressions and expands them recursively, combining the results into a new version of the special form or function call form.

Because **TXR Lisp** allows the same symbol to have a macro and function binding, the expander allows for interplay between the two, which produces useful behaviors. Recall from two paragraphs ago that whenever the expander expands a macro, it makes a note that it has done so. Subsequently, suppose that the rounds of macro expansion happen to terminate in such a way that the result is a function call form. The form's constituents are expanded, If the expansion of those constituents produces any change, then the resulting replacement function call form is again examined for the possibility that it may be a macro. This special requirement, not typically implemented by Lisp macro expanders, greatly simplifies the writing of macros which provide algebraic optimizations of function calls.

An example follows to illustrate the benefit of the rule. Note that the example involves some simple macros which change the number of times that an argument expression is evaluated. A more careful handling of this issue is omitted in order to keep the examples simple.

Suppose a macro is written for the `sqrt` function like this:

```
(defmacro sqrt (:match :form f)
  (((* @exp @exp)) exp)
  (@else f))
```

The macro uses pattern matching to recognize cases like `(sqrt (* a a))` when the argument is a product expression with two identical terms. This pattern implements the arithmetic identity that the positive square root of a real term multiplied by itself is just that term.

Now suppose that a similar macro is written to optimize a certain case of the `expt` function:

```
(defmacro expt (:match :form f)
  ((@exp 2) ^(* ,exp ,exp))
  (@else f))
```

This macro recognizes when the argument is being squared, turning `(expt x 2)` into `(* x x)`: a strength reduction from exponentiation to multiplication.

What if the following expression is then written:

```
(sqrt (expt x 2))
```

The special provision in the expander algorithm allows the above combination to reduce to just `x`, as follows. Firstly, the `(sqrt (expt x 2))` expression is treated as a macro call. It doesn't match the main

case in the macro, only the fallback case which returns the form unexpanded. The expander notes that it has invoked a macro, and then proceeds to treat the form as a function call. The function call's argument expression `(expt x 2)` is expanded as a macro. This produces a transformation: our `expt` macro reduces this quadratic term to `(* x x)`. Here is where the special rule comes into play. The expander sees that the function's arguments have been transformed. It knows that the original function call was the result of expansion. To promote more opportunities for expansion, it tries the transformed function call again as a macro. The `(sqrt (* x x))` form is handed to the `sqrt` macro, which this time has a match for the `(* x x)` argument pattern, reducing the entire form to `x`. Effectively, the `sqrt` macro has the opportunity to work with both the unexpanded argument syntax `(expt x 2)` as well as its expanded version. It is first offered the one, and when it declines to expand, then the other.

9.26.3 Operator `defmacro`

Syntax:

```
(defmacro name
          (param* [: opt-param* ] [. rest-param ])
   body-form*)
```

Description:

The `defmacro` operator is evaluated at expansion time. It defines a macro-expander function under the name *name*, effectively creating a new operator.

Note that the above syntax synopsis describes only the canonical parameter syntax which remains after parameter list macros are expanded. See the section Parameter List Macros.

Note that the parameter list is a macro parameter list, and not a function parameter list. This means that each *param* and *opt-param* can be not only a symbol, but it can itself be a parameter list. The corresponding argument is then treated as a structure which matches that parameter list. This nesting of parameter lists can be carried to an arbitrary depth.

A macro is called like any other operator, and resembles a function. Unlike in a function call, the macro receives the argument expressions themselves, rather than their values. Therefore it operates on syntax rather than on values. Also, unlike a function call, a macro call occurs in the expansion phase, rather than the evaluation phase.

The return value of the macro is the macro expansion. It is substituted in place of the entire macro call form. That form is then expanded again; it may itself be another macro call, or contain more macro calls.

A global macro defined using `defmacro` may decline to expand a macro form. Declining to expand is achieved by returning the original unexpanded form, which may be captured using the `:form` parameter. When a global macro declines to expand a form, the form is taken as-is. At evaluation time, it will be treated as a function call. Note: when a local macro defined by `macrolet` declines, more complicated requirements apply; see the description of `macrolet`.

Dialect Notes:

A macro in the global namespace introduced by `defmacro` may coexist with a function of the same name introduced by `defun`. This is not permitted in ANSI Common Lisp.

ANSI Common Lisp doesn't describe the concept of declining to expand, except in the area of compiler macros. Since TXR Lisp allows global macros and functions of the same name to coexist, ordinary macros can be used to optimize functions in a manner similar to Common Lisp compiler macros. A macro can be written of the same name as a function, and can optimize certain

cases of the function call by expanding them to some alternative syntax. Cases which it doesn't optimize are handled by declining to expand, in which case the form remains as the original function call.

Example:

```
;; dolist macro similar to Common Lisp's:
;;
;; The following will print 1, 2 and 3
;; on separate lines:
;; and return 42.
;;
;; (dolist (x '(1 2 3) 42)
;;   (format t "~s\n" x))

(defmacro dolist ((var list : result) . body)
  (let ((i (gensym)))
    ^(for ((,i ,list)) (,i ,result) ((set ,i (cdr ,i)))
       (let ((,var (car ,i)))
         ,*body))))
```

9.26.4 Operator `macrolet`

Syntax:

```
(macrolet ({(name macro-style-params
               macro-body-form*)}*)
   body-form*)
```

Description:

The `macrolet` binding operator extends the macro-time lexical environment by making zero or more new local macros visible.

The `macrolet` symbol is followed by a list of macro definitions. Each definition is a form which begins with a *name*, followed by *macro-style-params* which is a macro parameter list, and zero or more *macro-body-form*s. These macro definitions are similar to those globally defined by the `defmacro` operator, except that they are in a local environment.

The macro definitions are followed by optional *body-forms*. The macros specified in the definitions are visible to these forms.

Forms inside the macro definitions such as the *macro-body-form*s, and initializer forms appearing in the *macro-style-params* are subject to macro-expansion in a scope in which none of the new macros being defined are yet visible. Once the macro definitions are themselves macro-expanded, they are placed into a new macro environment, which is then used for macro expanding the *body-form*s.

A `macrolet` form is fully processed in the expansion phase of a form, and is effectively replaced by `progn` form which contains expanded versions of *body-form*s. This expanded structure shows no evidence that any macrolet forms ever existed in it. Therefore, it is impossible for the code evaluated in the bodies and parameter lists of `macrolet` macros to have any visibility to any surrounding lexical variable bindings, which are only instantiated in the evaluation phase, after expansion is done and macros no longer exist.

A local macro defined using `defmacro` may decline to expand a macro form. Declining to

expand is achieved by returning the original unexpanded form, which may be captured using the `:form` parameter. When a local macro declines to expand a form, the macro definition is temporarily hidden, as if it didn't exist in the lexical scope. If another macro of the same name is thereby revealed (a global macro or another local macro at a shallower nesting level), then an expansion is tried with that macro. If no such macro is revealed, or if a lexical function binding of that name is revealed, then no expansion takes place; the original form is taken as-is. When another macro is tried, the process repeats, resulting in a search which proceeds as far as possible through outer lexical scopes and finally the global scope.

9.26.5 Function `macro-form-p`

Syntax:

```
(macro-form-p obj [env])
```

Description:

The `macro-form-p` function returns `t` if *obj* represents the syntax of a form which is a macro form: either a compound macro or a symbol macro. Otherwise it returns `nil`.

A macro form is one that will transform under `macroexpand-1` or `macroexpand`; an object which isn't a macro form will not undergo expansion.

The optional *env* parameter is a macroexpansion environment. A macroexpansion environment is passed down to macros and can be received via their special `:env` parameter.

env is used by `macro-form-p` to determine whether *obj* is a macro in a lexical macro environment.

If *env* is not specified or is `nil`, then `macro-form-p` only recognizes global macros.

Example:

```
;; macro which translates to 'yes if its
;; argument is a macro from, or otherwise
;; transforms to the form 'no.

(defmacro ismacro (:env menv form)
  (if (macro-form-p form menv)
   ''yes ''no))

(macrolet ((local ()))
  (ismacro (local)))    ;; yields yes

(ismacro (local))       ;; yields no

(ismacro (ismacro foo)) ;; yields yes
```

During macro expansion, the global macro `ismacro` is handed the macro-expansion environment via `:env menv`.

When the macro is invoked within the macrolet, this environment includes the macro-time lexical scope in which the `local` macro is defined. So when global checks whether the argument form `(local)` is a macro, the conclusion is yes: the (local) form is a macro call in that environment: `macro-form-p` yields `t`.

When `(global (local))` is invoked outside of the macrolet, no local macro is visible is there, and so `macro-form-p` yields `nil`.

9.26.6 Functions `macroexpand-1` and `macroexpand`

Syntax:

```
(macroexpand-1 obj [env])
(macroexpand obj [env])
```

Description:

If *obj* is a macro form (an object for which `macro-form-p` returns `t`), these functions expand the macro form and return the expanded form. Otherwise, they return *obj*.

`macroexpand-1` performs a single expansion, expanding just the macro that is referenced by the symbol in the first position of *obj*, and returns the expansion. That expansion may itself be a macro form.

`macroexpand` performs an expansion similar to `macroexpand-1`. If the result is a macro form, then it expands that form, and keeps repeating this process until the expansion yields a non-macro-form. That non-macro-form is then returned.

The optional *env* parameter is a macroexpansion environment. A macroexpansion environment is passed down to macros and can be received via their special `:env` parameter. The environment they receive is their lexically apparent macro-time environment in which local macros may be visible. A macro can use this environment to "manually" expand some form in the context of that environment.

Example:

```
;; (rem-num x) expands x, and if x begins with a number,
;; it removes  the number and returns the resulting
;; form. Otherwise, it returns the entire form.

(defmacro rem-num (:env menv some-form)
  (let ((expanded (macroexpand some-form menv)))
    (if (numberp (car expanded))
      (cdr expanded)
      some-form)))

(macrolet ((foo () '(1 list 42))
           (bar () '(list 'a)))
  (list (rem-num (foo)) (rem-num (bar))))

--> ((42) (a))
```

The `rem-num` macro is able to expand the `(foo)` and `(bar)` forms it receives as the `some-form` argument, even though these forms use local macro that are only visible in their local scope. This is thanks to the macro environment passed to `rem-num`. It is correctly able to work with the expansions `(1 list 42)` and `(list 'a)` to produce `(list 42)` and `(list 'a)` which evaluate to `42` and `a` respectively.

9.26.7 Functions `macroexpand-1-lisp1` and `macroexpand-lisp1`

Syntax:

```
(macroexpand-1-lisp1 obj [env])
(macroexpand-lisp1 obj [env])
```

Description:

The `macroexpand-1-lisp1` and `macroexpand-lisp1` functions closely resemble, respectively, `macroexpand-1` and `macroexpand`.

The argument and return value syntax and semantics is almost identical, except for one difference. These functions consider argument *obj* to be syntax in a Lisp-1 evaluation context, such as any argument position of the `dwim` operator, or the equivalent DWIM Brackets notation.

This makes a difference because in a Lisp-1 evaluation context, an inner function binding is able to shadow an outer symbol macro binding of the same name.

The requirements about this language area are given in more detail in the description of the `dwim` operator.

Note: the `macroexpand-lisp1` function is useful to the implementor of a macro whose semantics requires one or more argument forms to be treated in a Lisp-1 context, in situations when such a macro needs to itself expand the material, rather than merely insert it as-is into the output code template.

9.26.8 Functions `expand` and `expand*`

Syntax:

```
(expand form [env])
(expand* form [env])
```

Description:

The functions `expand` and `expand*` both perform a complete expansion of *form* in the macro-environment *env*, and return that expansion.

If *env* is omitted, the expansion takes place in the global environment in which only global macros are visible.

The returned object is a structure that is devoid of any macro calls. Also, all `macrolet` and `symacrolet` blocks in form *form* are removed in the returned structure, replaced by their fully expanded bodies.

The difference between `expand` and `expand*` is that `expand` suppresses expansion-time deferred warnings (exceptions of type `defr-warning`), issued for unbound variables or functions. To suppress a warning means to intercept the warning exception with a handler which throws a `continue` exception to resume processing. What this requirement means is that if unbound functions or variables occur in the *form* being expanded by expand, the warning is effectively squelched. Rationale: `expand` is may be used by macros for expanding fragments which contain references to variables or functions which are not defined in those fragments.

9.26.9 Function `expand-with-free-refs`

Syntax:

```
(expand-with-free-refs form [inner-env [outer-env]])
```

Description:

The `expand-with-free-refs` form performs a full expansion of *form*, as if by the `expand` function and returns a list containing that expansion, plus four additional items which provide information about variable and function references which occur in *form*.

If both *inner-env* and *outer-env* are provided, then it is expected that *inner-env* is lexically nested within *outer-env*.

Note: it is not required that *outer-env* be the immediate parent of *inner-env*.

Note: a common usage situation is that *outer-env* is the environment of the invocation of a "parent" macro which generates a form that contains local macros. The bodies of those local macros use `expand-with-free-refs`, specifying their own environment as *inner-env* and that of their generating "parent" as *outer-env*.

In detail, the five items of the returned list are (*expansion fv-inner ff-inner fv-outer ff-outer*) whose descriptions are:

expansion
: The full expansion of *form*, containing no macro invocations, or `symacrolet` or `macrolet` forms.

fv-inner
: A list of the free variables which occur in *form* relative to the *inner-env* environment. That is to say, variables that are not bound inside *form* and are not also bound in *inner-env*. If *inner-env* is omitted, then these are the absolutely free variables occurring in *form*.

ff-inner
: Exactly like *fv-inner* but informing about function bindings rather than variables.

fv-outer
: A list of the variables which which occur in *form* which would be free if the environments between *inner-env* and *outer-env* (including the former, excluding the latter) were removed from consideration. A more detailed description of this semantics is given below. If *outer-env* is omitted, then these are the absolutely free variables occurring in *form*, ignoring the *inner-env*.

ff-outer
: Exactly like *fv-outer* but informing about function bindings rather than variables.

The semantics of the treatment of *inner-env* and *outer-env* in the calculation of *fv-outer* and *ff-outer* is as follows. A new environment *diff-env* is calculated from these two environments, and *form* is expanded in this environment. Variables and functions occurring in *form* which are not bound in *diff-env* are listed as *fv-outer* and *ff-outer*.

This *diff-env* is calculated as follows. First *diff-env* is initialized as a copy of *outer-env*. Then, all environments below *outer-env* down to *inner-env* are examined for bindings which shadow bindings in *diff-env*. Those shadows are removed from *diff-env*. Therefore, what remains in *diff-env* are those bindings from *outer-env* that are *not* shadowed by the environments between *inner-env* and *outer-env*.

Within each of the lists of variables returned by `expand-with-free-refs`, the order of the variables is not specified.

Example:

Suppose that `mac` is a macro which somehow has access to the two indicated lexical environments in the following code snippet:

```
(let (a c) ;; <- outer-env
  (let (b)
    (let (c) ;; <- inner-env
      (mac (list a b c d)))))
```

Suppose that `mac` invokes the `expand-with-free-refs` function, passing in the `(list a b c d)` argument form as `form` and two macro-time environment objects corresponding to the indicated environments.

Then the following object shall be a correct return value of `expand-with-free-refs`:

```
((list a b c d) (d) nil (d c b) nil)
```

A complete code example of this is given below.

Other correct return values are possible due to permitted variations in the order of the variables within the four lists. For instance, instead of `(d c b)` the list `(c b d)` may appear.

The *`fv-inner`* list is `(d)` because this is the only variable that occurs in `(list a b c d)` which is free with regard to *`inner-env`*. The `a`, `b` and `c` variables are not listed because they appear bound inside *`inner-env`*.

The reported *`fv-outer`* list is `(b c d)` because the form is considered against *`diff-env`* which is formed by removing the shadowing bindings from *`outer-env`*. The difference between `(a c)` and `(b c)` is `a` and so the form is considered in an environment containing the binding `a` which leaves `(b c d)` free.

The following is a complete code sample demonstrating the above descriptions:

```
;; Given this macro:
(defmacro bigmac (:env out-env big-form)
  ^(macrolet ((mac (:env in-env little-form)
                   ^',(expand-with-free-refs
                         little-form in-env ,out-env)))
    ,big-form))

(let (a c) ;; <- outer-env, surrounding bigmac
  (bigmac
    (let (b)
      (let (c) ;; <- inner-env, surrounding mac
        (mac (list a b c d))))))

--> ((list a b c d) (d) nil (d c b) nil)
```

Note: this information is useful because a set difference can be calculated between the two reported sets. The set difference between the *`fv-outer`* variables `(b c d)` and the *`fv-inner`* variables `(d)` is `(b c)`.

That set difference `(b c)` is significant because it precisely informs about the *bound* variables which occur in `(list a b c d)` which appear bound in *`inner-env`*, but are not bound due

to a binding coming from *outer-env*. In the above example, these are the variables enclosed in the `bigmac` macro, but external to the inner `mac` macro.

The variable `d` is not listed in `(b c)` because it is not a bound variable. The variable `a` is not in `(b c)` because though it is bound in *inner-env*, that binding comes from *outer-env*.

The upshot of this logic is that it allows a macro to inspect a form in order to discover the identities of the variables and functions which are used inside that form, whose definitions come from a specific, bounded scope surrounding that form.

9.26.10 Functions `lexical-var-p` and `lexical-fun-p`

Syntax:

```
(lexical-var-p env form)
(lexical-fun-p env form)
```

Description:

These two functions are useful to macro writers. They are intended to be called from the bodies of macro expanders, such as the bodies of `defmacro` or `macrolet` forms. The *env* argument is a macro-time environment, which is available to macros via the special `:env` parameter. Using these functions, a macro can enquire whether a given *form* is a symbol which has a variable binding or a function binding in the local lexical environment. This information is known during macro expansion. The macro expander recognizes lexical function and variable bindings, because these bindings can shadow macros.

Special variables are not lexical. The function `lexical-var-p` returns `nil` if *form* satisfies `special-var-p` function, indicating that it is the name of a special variable.

The `lexical-var-p` function also returns `nil` for global lexical variables. If *form* is a symbol for which only a global lexical variable binding is apparent, `lexical-var-p` returns `nil`. Testing for the existence for a global variable can be done using `boundp`; if a symbol is `boundp` but not `special-var-p`, then it is a global lexical variable.

Similarly, `lexical-fun-p` returns `nil` for global functions.

Example:

```
;;
;; this macro replaces itself with :lexical-var if its
;; argument is a lexical variable, :lexical-fun if
;; its argument is a lexical function, or with
;; :not-lex-fun-var if neither is the case.
;;
(defmacro classify (sym :env e)
  (cond
    ((lexical-var-p e sym) :lexical-var)
    ((lexical-fun-p e sym) :lexical-fun)
    (t :not-lex-fun-var)))

;;
;; Use classify macro above to report classification
;; of the x, y and f symbols in the given scope
;;
(let ((x 1) (y 2))
```

```
  (symacrolet ((y x))
    (flet ((f () (+ 2 2)))
      (list (classify x) (classify y) (classify f)))))
--> (:lexical-var :not-lex-fun-var :lexical-fun)

;; Locally bound specials are not lexical

(let ((*stdout* *stdnull*))
  (classify *stdout*))
--> :not-lex-fun-var
```

Note:

These functions do not call `macroexpand` on the form. In most cases, it is necessary for the macro writers to do so. Not that in the above example, symbol `y` is classified as neither a lexical function nor variable. However, it can be macro-expanded to `x` which is a lexical variable.

9.26.11 Function `lexical-lisp1-binding`

Syntax:

```
(lexical-lisp1-binding env symbol)
```

Description:

The `lexical-lisp1-binding` function inspects the macro-time environment *`env`* to determine what kind of binding, if any, does *`symbol`* have in that environment, from a Lisp-1 perspective.

That is to say, it considers function bindings, variable bindings and symbol macro bindings to be in a single name space and finds the innermost binding of one of these types for *`symbol`*.

If such a binding is found, then the function returns one of the three keyword symbols `:var`, `:fun`, or `:symacro`.

If no such lexical binding is found, then the function returns `nil`.

Note that a `nil` return doesn't mean that the symbol doesn't have a lexical binding. It could have an operator macro lexical binding (a macro binding in the function namespace established by `macrolet`).

9.26.12 Operator `defsymacro`

Syntax:

```
(defsymacro sym form)
```

Description:

A `defsymacro` form introduces a symbol macro. A symbol macro consists of a binding between a symbol *`sym`* and and a *`form`*. The binding denotes the form itself, rather than its value.

The *`form`* argument is not subject to macro expansion; it is associated with *`sym`* in its unexpanded state, as it appears in the `defmacro` form.

The `defsymacro` form must be evaluated for its defining to take place; therefore, the definition is not available in the top-level form which contains the `defsymacro` invocation; it becomes available to a subsequent top-level form.

Subsequent to the evaluation of the `defsymacro` definition, whenever the macro expander encounters *sym* sym as a form, it replaces it by *form*. After this replacement takes place, *form* itself is then processed for further replacement of macros and symbol macros.

Symbol macros are also recognized in contexts where *sym* denotes a place which is the target of an assignment operation like `set` and similar.

Note: if a symbol macro expands to itself directly, expansion stops. However, if a symbol macro expands to itself through a chain of expansions, runaway expansion-time recursion will occur.

If a global variable exists by the name *sym*, then `defsymacro` first removes that variable from the global environment, and if that variable is special, the symbol's special marking is removed. `defsymacro` doesn't alter the dynamic binding of a special variable. Any such a binding remains intact. If `defsymacro` is evaluated in a scope in which there is any lexical or dynamic binding of *sym* in the variable namespace, whether as a variable or macro, the global symbol macro is shadowed by that binding.

9.26.13 Operator `symacrolet`

Syntax:

```
(symacrolet ({(sym form)}*) body-form*)
```

Description:

The `symacrolet` operator binds local, lexically scoped macros that are similar to the global symbol macros introduced by `defsymacro`.

Each *sym* in the bindings list is bound to its corresponding form, creating a new extension of the expansion-time lexical macro environment.

Each *body-form* is subsequently macro-expanded in this new environment in which the new symbol macros are visible.

Note: ordinary lexical bindings such as those introduced by let or by function parameters lists shadow symbol macros. If a symbol `x` is bound by nested instances of `macrolet` and a `let`, then the scope enclosed by both constructs will see whichever of the two bindings is more inner, even though the bindings are active in completely separate phases of processing.

From the perspective of the arguments of a `dwim` form, lexical function bindings also shadow symbol macros. This is consistent with the Lisp-1-style name resolution which applies inside a `dwim` form. Lexical operator macros do not shadow symbol macros under any circumstances.

9.26.14 Macros `placelet` and `placelet*`

Syntax:

```
(placelet ({(sym place)}*) body-form*)
(placelet* ({(sym place)}*) body-form*)
```

Description:

The `placelet` macro binds lexically scoped symbol macros in such a way that they behave as aliases for places denoted by place forms.

Each *place* must be an expression denoting a syntactic place. The corresponding *sym* is established as an alias for the storage location which that place denotes, over the scope of the *body-forms*.

This binding takes place in such a way that each *place* is evaluated exactly once, only in order to determine its storage location. The corresponding *sym* then serves as an alias for that location, over the scope of the *body-form*s. This means that whenever *sym* is evaluated, it stands for the value of the storage location, and whenever a value is apparently stored into *sym*, it is actually the storage location which receives it.

The `placelet*` variant implements an alternative scoping rule, which allows a later *place* form to refer to a *sym* bound to an earlier *place* form. In other words, a given *sym* binding is visible not only to the *body-form*s but also to *place* forms which occur later.

Note: certain kinds of places, notably `(force` *promise*`)` expressions, must be accessed before they can be stored, and this restriction continues to hold when those places are accessed through `placelet` aliases.

Note: `placelet` differs from `symacrolet` in that the forms themselves are not aliased, but the storage locations which they denote. `(symacrolet ((x y)) z)` performs the syntactic substitution of symbol `x` by form `y`, wherever `x` appears inside `z` as an evaluated form, and is not shadowed by any inner binding. Whereas `(placelet ((x y)) z)` generates code which arranges for `y` to be evaluated to a storage location, and syntactically replaces occurrences of `x` with a form which directly denotes that storage location, wherever `x` appears inside `z` as an evaluated form, and is not shadowed by any inner binding. Also, `x` is not necessarily substituted by a single, fixed form, as in the case of `symacrolet`. Rather it may be substituted by one kind of form when it is treated as a pure value, and another kind of form when it is treated as a place.

Note: multiple accesses to an alias created by `placelet` denote multiple accesses to the aliased storage location. That can mean multiple function calls or array indexing operations and such. If the target of the alias is `(read-once` *place*`)` instead of *place*, then a single access occurs to fetch the prior value of *place* and stored into a hidden variable. All of the multiple occurrences of the alias then simply retrieve this cached prior value from the hidden variable, rather than accessing the place. The `read-once` macro is independent of `placelet` and separately documented.

Example:

Implementation of `inc` using `placelet`:

```
(defmacro inc (place : (delta 1))
  (with-gensyms (p)
    ^(placelet ((,p ,place))
       (set ,p (+ ,p ,delta)))))
```

The gensym `p` is used to avoid accidental capture of references emanating from the `delta` form.

9.26.15 Macro `macro-time`

Syntax:

```
(macro-time form*)
```

Description:

The `macro-time` macro evaluates its arguments immediately during macro expansion.

The *form* arguments are processed from left to right. Each *form* is fully expanded and evaluated in the top-level environment before the next form is considered.

The value of the last *form*, or else `nil` if there aren't any arguments, is converted into a literal expression which denotes that value, and the resulting literal is produced as the expansion of *macro-time*.

Note: `macro-time` supports techniques that require a calculation to be performed in the environment where the program is being compiled, and inserting the result of that calculation as a literal into the program source. Possibly, the calculation can have some useful effect in that environment, or use as an input information that is available in that environment. The `load-time` operator also inserts a calculated value as a de facto literal into the program, but it performs that calculation in the environment where the compiled file is being loaded. The two operators may be considered complementary in this sense.

Consider the source file:

```
(defun host-name-c () (macro-time (uname).nodename))

(defun host-name-l () (load-time (uname).nodename))
```

If this is compiled via `compile-file`, the `uname` call in `host-name-c` takes place when it is macro-expanded. Thereafter, the compiled version of the function returns the name of the machine where the compilation took place, no matter in what environment it is subsequently loaded and called.

In contrast, the compilation of `host-name-l` arranges for that function's `uname` call to take place just one time, whenever the compiled file is loaded. Each time the function is subsequently called, it will return the name of the machine where it was loaded, without making any additional calls to `uname`.

Note: `macro-time` can be understood in terms of the following implementation. Note that this implementation always produces a `quote` expression, which `macro-time` is not required to do if *val* is self-evaluating:

```
(defmacro macro-time (. forms)
  (let (val)
    (each ((f forms))
      (set val (eval f)))
    ^(quote ,val)))
```

Because `eval` treats a top-level `progn` specially, this implementation is also possible:

```
(defmacro macro-time (. forms)
  ^(quote ,(eval ^(progn ,*forms))))
```

Examples:

```
;; The (1 2 3) object is produced at macro-expansion time, becoming
;; a quoted literal which evaluates to (1 2 3).
(macro-time (list 1 2 3)) -> (1 2 3)

;; The above fact is revealed by macroexpand: the list form was
;; evaluated, and then quote was inserted to produce (quote (1 2 3))
;; which is notated '(1 2 3):
(macroexpand '(macro-time (list 1 2 3))) -> '(1 2 3)

;; Quote isn't required on a self-evaluating object; it serves
```

```
;; as a literal expression denoting itself:
(macroexpand '(macro-time (join-with "-" "a" "b"))) -> "a-b"
```

9.26.16 Macro `equot`

Syntax:

```
(equot form)
```

Description:

The `equot` macro ("expand and quote") performs a full expansion of `form` in the surrounding macro environment. Then it constructs a `quote` form whose argument is the expansion. This quote form is then returned as the macro replacement for the original `equot` form.

Example:

```
(symacrolet ((a (+ 2 2)))
  (list (quote a) (equot a) a))
--> (a (+ 2 2) 4)
```

Above, the expansion of `a` is `(+ 2 2)`. Thus the macro call `(equot a)` expands to `(quote (+ 2 2))`. When that is evaluated, it yields `(+ 2 2)`.

If `a` is quoted, then the result is `a`: no expansion or evaluation takes place. Whereas if `a` is presented for evaluation, then not only is it expanded to `(+ 2 2)`, but that expansion is reduced to 4.

The `equot` operator is a an intermediate point between these two semantics: it permits expansion to proceed, but then suppresses evaluation of the result.

9.26.17 Operators `tree-bind`, `mac-param-bind` and `mac-env-param-bind`

Syntax:

```
(tree-bind macro-style-params expr form*)
(mac-param-bind context-expr
   macro-style-params expr form*)
(mac-env-param-bind context-expr env-expr
   macro-style-params expr form*)
```

Description:

The `tree-bind` operator evaluates `expr`, and then uses the resulting value as a counterpart to a macro-style parameter list. If the value has a tree structure which matches the parameters, then those parameters are established as bindings, and the *`form`*s, if any, are evaluated in the scope of those bindings. The value of the last *`form`* is returned. If there are no forms, `nil` is returned. Under `tree-bind`, the value of the `:form` available to *`macro-style-params`* is the `tree-bind` form itself.

The `mac-param-bind` operator is similar to `tree-bind` except that it takes an extra argument, *`context-expr`*. This argument is an expression which is evaluated. It is expected to evaluate to a compound form. If an error occurs during binding, the error diagnostic message is based on information obtained from this form. By contrast, the `tree-bind` operator's error diagnostic refers to the `tree-bind` form, which is cryptic if the binding is used for the implementation of some other construct, hidden from the user of that construct. In addition, *`context-expr`* specifies the value for the `:form` parameter that *`macro-style-params`* may refer to.

The `mac-env-param-bind` is an extension of `mac-param-bind` which takes one more

argument, `env-expr`, before the macro parameters. This expression is evaluated, and becomes the value of the `:env` parameter that *`macro-style-params`* may refer to.

Under `tree-bind` and `mac-param-bind`, the `:env` parameter takes on the value `nil`.

Under all three operators, the `:whole` parameter takes on the value of *`expr`*.

These operators throw an exception if there is a structural mismatch between the parameters and the value of `expr`. One way to avoid this exception is to use `tree-case`, which is based on the conventions of `tree-bind`. There exists no `tree-case` analog for `mac-param-bind` or `mac-env-param-bind`.

9.26.18 Operator `tree-case`

Syntax:

```
(tree-case expr {(macro-style-params form*)}*)
```

Description:

The `tree-case` operator evaluates *`expr`* and matches it against a succession of zero or more cases. Each case defines a pattern match, expressed as a macro style parameter list *`macro-style-params`*.

If the object produced by *`expr`* matches *`macro-style-params`*, then the parameters are bound, becoming local variables, and the *`form`*s, if any, are evaluated in order in the environment in which those variables are visible. If there are forms, the value of the last *`form`* becomes the result value of the case, otherwise the result value of the case is nil.

If the result value of a case is the object `:` (the colon symbol), then processing continues with the next case. Otherwise the evaluation of `tree-case` terminates, returning the result value.

If the value of *`expr`* does not match the *`macro-style-params`* parameter list of a case, processing continues with the next case.

If no cases match, then `tree-case` terminates, returning `nil`.

Example:

```
;; reverse function implemented using tree-case

(defun tb-reverse (obj)
  (tree-case obj
    (() ())      ;; the empty list is just returned
    ((a) obj)    ;; one-element list returned
    ((a . b) ^(,*(tb-reverse b) ,a)) ;; car/cdr recursion
    (a a)))     ;; atom is just returned
```

Note that in this example, the atom case is placed last, because an argument list which consists of a symbol is a "catch all" match that matches any object. We know that it matches an atom, because the previous `(a . b)` case matches conses. In general, the order of the cases in `tree-case` is important: even more so than the order of cases in a `cond` or `caseql`. The one-element list case is unnecessary; it can be removed.

9.26.19 Macro `tb`

Syntax:

```
(tb macro-style-params form*)
```

Description:

The `tb` macro is similar to the `lambda` operator but its argument binding is based on a macro-style parameter list. The name is an abbreviation of `tree-bind`.

A `tb` form evaluates to a function which takes a variable number of arguments.

When that function is called, those arguments are taken as a list object which is matched against *`macro-style-params`* as if by *`tree-bind`*. If the match is successful, then the parameters are bound to the corresponding elements from the argument structure and each successive *`form`* is evaluated in an environment in which those bindings are visible. The value of the last *`form`* is the return value of the function. If there are no forms, the function's return value is `nil`.

The following equivalence holds, where `args` should be understood to be a globally unique symbol:

```
(tb pattern body ...) <--> (lambda (. args)
                              (tree-bind pattern args body ...))
```

9.26.20 Macro `tc`

Syntax:

```
(tc {(macro-style-params form*)}*)
```

Description:

The `tc` macro produces an anonymous function whose behavior is closely based on the `tree-case` operator. Its name is an abbreviation of `tree-case`.

The anonymous function takes a variable number of arguments. Its argument list is taken to be the value macro is tested against the multiple pattern clauses of an implicit `tree-case`. The return value of the function is that of the implied `tree-case`.

The following equivalence holds, where `args` should be understood to be a globally unique symbol:

```
(tc clause1 clause2 ...) <--> (lambda (. args)
                                 (tree-case args
                                    clause1 clause2 ...))
```

9.26.21 Macro `with-gensyms`

Syntax:

```
(with-gensyms (sym*) body-form*)
```

Description:

The `with-gensyms` evaluates the *`body-form`*s in an environment in which each variable name symbol *`sym`* is bound to a new uninterned symbol ("gensym").

Example:

The code:

```
(let ((x (gensym))
      (y (gensym))
      (z (gensym)))
  ^(,x ,y ,z))
```

may be expressed more conveniently using the `with-gensyms` shorthand:

```
(with-gensyms (x y z)
  ^(,x ,y ,z))
```

9.27 Parameter List Macros

Parameter list macros, also more briefly called *parameter macros* are an original feature of **TXR Lisp**.

If the first element of a function or macro parameter list is a keyword symbol other than `:env`, `:whole`, `:form` or `:` (the colon symbol), it denotes a parameter macro. This keyword symbol is expected to have a binding in the parameter macro namespace: a global namespace which associates keyword symbols with parameter list expander functions.

Expansion of a parameter list macro occurs at macro-expansion time, when a function's parameter list is traversed by the macro expander. It takes place as follows. First, the keyword is removed from the parameter list. The keyword's binding in the parameter macro namespace is retrieved. If it doesn't exist, an exception is thrown. Otherwise, the remaining parameter list is first recursively processed for more occurrences of parameter macros. This expansion produces a transformed parameter list, along with a transformed function body. These two artifacts are then passed to the transformer function retrieved from the keyword symbol's binding. The function returns a further transformed version of the parameter list and body. These are processed for more parameter macros. The process terminates when no more expansion is possible, because a parameter list has been produced which does not begin with a parameter macro. This final parameter list and its accompanying body are then taken in place of the original parameter list and body.

TXR Lisp provides a two built-in parameter list macros. The `:key` parameter macro endows a function keyword parameters. The `:match` parameter macro allows a function to be expressed using pattern matching, which requires the body to consist of pattern-matching clauses.

The implementation of both of these macros is written entirely using this parameter list macro mechanism, by means of the public `define-param-expander` macro.

9.27.1 Special variable `*param-macro*`

Description:

The variable `*param-macro*` holds a hash table which associates keyword symbols with parameter list expander functions.

The functions are expected to conform to the following syntax:

```
(lambda (params body env form) form*)
```

The *`params`* parameter receives the parameter list of the function which is undergoing parameter expansion. All other parameter macros have already been expanded.

The *`body`* parameter receives the list of body forms. The function is expected to return a `cons`

cell whose car contains the transformed parameter list, and whose cdr contains the transformed list of body forms. Parameter expansion takes place at macro expansion time.

The *env* parameter receives the macro-expansion-time environment which surrounds the function being expanded. Note that this environment doesn't take into account the parameters themselves; therefore, it is not the correct environment for expanding macros among the *body* forms. For that purpose, it must be extended with shadowing entries, the manner of doing which is undocumented. However *env* may be used directly for expanding init forms for optional parameters occurring in *params*.

The *form* parameter receives the overall function-defining form that is being processes, such as a defun or lambda form. This is intended for error reporting.

A parameter transformer returns the transformed parameter list and body as a single object: a list whose first element is the parameter list, and whose remaining elements are the forms of the body. Thus, the following is a correct null transformer:

```
(lambda (params body env form)
  (cons params body))
```

9.27.2 Macro define-param-expander

Syntax:

```
(define-param-expander name (pvar bvar : evar fvar)
   form*)
```

Description:

The define-param-expander macro provides syntax for defining parameter macros. Invocations of this macro expand to code which constructs an anonymous function and installs it into the *param-macro* hash table, under the key given by *name*.

The *name* parameter's argument should be a keyword symbol that is valid for use as a parameter macro name.

The *pvar*, *bvar*, *evar* and *fvar* arguments must be symbols suitable for variable binding. These symbols define the parameters of the expander function which shall, respectively, receive the parameter list, body forms, macro environment and function form. If *evar* is omitted, a symbol generated by the gensym function is used. Likewise if *fvar* is omitted.

The *form* arguments constitute the body of the expander.

The define-param-expander form returns *name*.

The parameter macro returns the transformed parameter list and body as a single object: a list whose first element is the parameter list, and whose remaining elements are the forms of the body.

Example:

The following example shows the implementation of a parameter macro :memo which provides rudimentary memoization. Using the macro is extremely easy. It is a matter of simply inserting the :memo keyword at the front of a function's parameter list. The function is then memoized.

```
(defvarl %memo% (hash :weak-keys))
```

```
(defun ensure-memo (sym)
  (or (gethash %memo% sym)
      (sethash %memo% sym (hash))))

(define-param-expander :memo (param body)
  (let* ((memo-parm [param 0..(posq : param)])
         (hash (gensym))
         (key (gensym)))
    ^(,param (let  ((,hash (ensure-memo ',hash))
                    (,key (list ,*memo-parm)))
                (or (gethash ,hash ,key)
                    (sethash ,hash ,key (progn ,*body)))))))
```

The above `:memo` macro may be used to define a memoized Fibonacci function as follows:

```
(defun fib (:memo n)
  (if (< n 2)
    (clamp 0 1 n)
    (+ (fib (pred n)) (fib (ppred n)))))
```

All that is required is the insertion of the `:memo` keyword.

9.27.3 Parameter list macro `:key`

Syntax:

```
(:key non-key-param*
   [ -- {sym | (sym [init-form [p-sym]])}* ]
   [ . rest-param ])
```

Description:

Parameter list macro `:key` injects keyword parameter support into functions and macros.

When `:key` appears as the first item in a function parameter list, a special syntax is recognized in the parameter list. After any required and optional parameters, the symbol `--` (two dashes) may appear. Parameters after this symbol are interpreted as keyword parameters. After the keyword parameters, a rest parameter may appear in the usual way as a symbol in the dotted position.

Keyword parameters use the same syntax as optional parameters, except that if used in a macro parameter list, they do not support destructuring whereas optional parameters do. That is to say, regardless whether `:key` is used in a function or macro, keyword parameters are symbols.

A keyword parameter takes three possible forms:

sym A keyword parameter may be specified as a simple symbol *sym*. If the argument for such a keyword parameter is missing, it takes on the value `nil`.

(*sym init-form*)
If the keyword parameter symbol *sym* is enclosed in a list, then the second element of that list specifies a default value, similarly to the default value for an optional argument. If the function is called in such a way that the argument for the parameter is missing, the *init-form* is evaluated and the resulting value is bound to the keyword parameter. The evaluation takes place in a lexical scope in which the required and optional parameters are are already visible, and their values are bound. If there is a *rest-param* it is also visible in this scope, even though in the parameter list it appears to the left.

(*sym init-form p-sym*)

The three-element form of the keyword parameter specifies an additional symbol *p-sym*, which names an argument that implicitly receives a Boolean argument indicating the presence of the keyword argument. If an argument is not passed for the keyword parameter *sym*, then parameter *sym-p* takes on the value `nil`. If an argument is given for *sym*, then the *sym-p* argument takes on the value `t`. This mechanism also closely resembles the analogous one supported in optional arguments. See the previous paragraph regarding the evaluation scope of *init-form*.

In a call to a `:key`-enabled function, keyword arguments begin after those arguments which satisfy all of the required and optional parameters. Keyword arguments consist of interleaved indicators and values, which are separate arguments. Thus passing a keyword argument actually requires the passing of two function arguments: an indicator keyword symbol, followed by the associated value. The indicator keywords are expected to have the same symbol name as the defined keyword parameters. For instance, the indicator-value pair `:xyz 42` passes the value `42` to a keyword parameter that may be named `xyz` in any package: it may be `usr:xyz` or `mypackage:xyz` and so forth. Arguments specifying unrecognized keywords are ignored.

If the function has a *rest-param*, then that parameter receives the keyword arguments as a list. Since that list contains indicators and values, it is a de facto property list. In detail, the `:key` mechanism generates a regular variadic function which receives the keyword arguments as the trailing argument list. That function parses the recognized keyword arguments out of the trailing list, and binds them to the keyword parameter symbols as local variables. If a *rest-param* parameter is defined, then the entire keyword argument list is available through that parameter, and the keyword argument parsing logic also refers to the value of that parameter to gain access to the keyword arguments. If there is no *rest-param* specified, then the `:key` macro adds a *rest-param* using a machine-generated symbol. The argument parsing logic then refers to the value of that symbol.

Example:

Define a function `fun` with two required arguments `a b`, one optional argument `c`, two keyword arguments `foo` and `bar`, and a rest parameter `klist`:

```
(defun fun (:key a b : c -- foo bar . klist)
  (list a b c foo bar klist))

(fun 1 2 3 :bar 4) -> (1 2 3 nil 4 (:bar 4))
```

Define a function with only keyword arguments, with default expressions and Boolean indicator params:

```
(defun keyfun (:key -- (a 10 a-p) (b 20 b-p))
  (list a a-p b b-p))

(keyfun :a 3) -> (3 t 20 nil)

(keyfun :b 4) -> (10 nil 4 t)

(keyfun :c 4) -> (10 nil 20 nil)

(keyfun) -> (10 nil 20 nil)
```

9.27.4 Function `macroexpand-params`

Syntax:

```
(expand-params proto-form [env])
```

Description:

The `expand-param` function expands all of the parameter list macros expressed in the *prototype form* `proto-form`, returning an expanded version of the form.

The `proto-form` is a compound for which has a shape very similar to a lambda expression, and may be a lambda expression.

The first element of `proto-form` is a name, which is an arbitrary object, though the use of a symbol is strongly recommended. This object plays no role in `expand-params` other than for composing diagnostic messages if errors occur.

The second element of `proto-form` is the parameter list.

The remaining elements of `proto-form` are zero or more body forms.

If `proto-form` contains no parameter macro invocations, then it is returned.

The optional `env` parameter specifies the macro environment which is passed to the parameter macro expanders, which they can receive via the `:env` parameter. The default value `nil` specifies the top-level environment.

Examples:

```
;; No expansion: argument is returned
(macroexpand-params '(foo (arg) body)) -> (foo (arg) body)

;; Expand :key macro
(macroexpand-params '(bar (:key a b c -- d (e 1234 f-p)) body))
--> (bar (a b c . #:g0014)
      (let (d e f-p)
        (let ((#:g0015 (memp :d #:g0014)))
          (when #:g0015
            (set d (cadr #:g0015))))
        (let ((#:g0015 (memp :e #:g0014)))
          (cond
            (#:g0015 (set e (cadr #:g0015))
                     (set f-p t))
            (t (set e 1234))))
       body))
```

9.28 Mutation of Syntactic Places

9.28.1 Macro `set`

Syntax:

```
(set {place new-value}*)
```

Description:

The `set` operator stores the values of expressions in places. It must be given an even number of arguments.

If there are no arguments, then `set` does nothing and returns `nil`.

If there are two arguments, *`place`* and *`new-value`*, then *`place`* is evaluated to determine its storage location, then *`new-value`* is evaluated to determine the value to be stored there, and then the value is stored in that location. Finally, the value is also returned as the result value.

If there are more than two arguments, then `set` performs multiple assignments in left-to-right order. Effectively, `(set v1 e1 v2 e2 ... vn en)` is precisely equivalent to `(progn (set v1 e1) (set v2 e2) ... (set vn en))`.

9.28.2 Macro `pset`

Syntax:

```
(pset {place new-value}*)
```

Description:

The syntax of `pset` is similar to that of `set`, and the semantics is similar also in that zero or more places are assigned zero or more values. In fact, if there are no arguments, or if there is exactly one pair of arguments, `pset` is equivalent to `set`.

If there are two or more argument pairs, then all of the arguments are evaluated first, in left-to-right order. No store takes place until after every *`place`* is determined, and every *`new-value`* is calculated. During the calculation, the values to be stored are retained in hidden, temporary locations. Finally, these values are moved into the determined places. The rightmost value is returned as the form's value.

The assignments thus appear to take place in parallel, and `pset` is capable of exchanging the values of a pair of places, or rotating the values among three or more places. (However, there are more convenient operators for this, namely `rotate` and `swap`).

Example:

```
;; exchange x and y
(pset x y y x)

;; exchange elements 0 and 1; and 2 and 3 of vector v:
(let ((v (vec 0 10 20 30))
      (i -1))
  (pset [vec (inc i)] [vec (inc i)]
        [vec (inc i)] [vec (inc i)])
   vec)
-> #(10 0 30 20)
```

9.28.3 Macro `zap`

Syntax:

```
(zap place [new-value])
```

Description:

The `zap` macro assigns *`new-value`* to *`place`* and returns the previous value of *`place`*.

If *`new-value`* is missing, then `nil` is used.

In more detail, first `place` is evaluated to determine the storage location. Then, the location is

accessed to retrieve the previous value. Then, the `new-value` expression is evaluated, and that value is placed into the storage location. Finally, the previously retrieved value is returned.

9.28.4 Macro `flip`

Syntax:

```
(flip place)
```

Description:

The `flip` macro toggles the Boolean value stored in *place*.

If *place* previously held `nil`, it is set to `t`, and if it previously held a value other than `nil`, it is set to `nil`.

9.28.5 Macros `test-set` and `test-clear`

Syntax:

```
(test-set place)
(test-clear place)
```

Description:

The `test-set` macro examines the value of *place*. If it is `nil` then it stores `t` into the place, and returns `t`. Otherwise it leaves *place* unchanged and returns `nil`.

The `test-clear` macro examines the value of *place*. If it is Boolean true (any value except `nil`) then it stores `nil` into the place, and returns `t`. Otherwise it leaves *place* unchanged and returns `nil`.

9.28.6 Macro `compare-swap`

Syntax:

```
(compare-swap place cmp-fun cmp-val store-val)
```

Description:

The `compare-swap` macro examines the value of *place* and compares it to *cmp-val* using the comparison function given by the function name *cmp-fun*.

This comparison takes places as if by evaluating the expression `(`*cmp-fun value cmp-val*`)` where *value* denotes the current value of *place*.

If the comparison is false, *place* is not modified, the *store-val* expression is not evaluated, and the macro returns `nil`.

If the comparison is true, then `compare-swap` evaluates the *store-val* expression, stores the resulting value into *place* and returns `t`.

9.28.7 Macros `inc` and `dec`

Syntax:

```
(inc place [delta])
(dec place [delta])
```

Description:

The `inc` macro increments *place* by adding *delta* to its value. If *delta* is missing, the value used in its place the integer 1.

First the *place* argument is evaluated as a syntactic place to determine the location. Then, the value currently stored in that location is retrieved. Next, the *delta* expression is evaluated. Its value is added to the previously retrieved value as if by the + function. The resulting value is stored in the place, and returned.

The macro `dec` works exactly like `inc` except that addition is replaced by subtraction. The similarly defaulted *delta* value is subtracted from the previous value of the place.

9.28.8 Macros `pinc` and `pdec`

Syntax:

```
(pinc place [delta])
(pdec place [delta])
```

Description:

The macros `pinc` and `pdec` are similar to `inc` and `dec`.

The only difference is that they return the previous value of *place* rather than the incremented value.

9.28.9 Macros `test-inc` and `test-dec`

Syntax:

```
(test-inc place [delta [from-val]])
(test-dec place [delta [to-val]])
```

Description:

The `test-inc` and `test-dec` macros provide combined operations which change the value of a place and provide a test whether, respectively, a certain previous value was overwritten, or a certain new value was attained. By default, this tested value is zero.

The `test-inc` macro notes the prior value of *place* and then updates it with that value, plus *delta*, which defaults to 1. If the prior value is `eql` to *from-val* then it returns `t`, otherwise `nil`. The default value of *from-val* is zero.

The `test-dec` macro produces a new value by subtracting *delta* from the value of *place*. The argument *delta* defaults to 1. The new value is stored into *place*. If the new value is `eql` to *to-val* then `t` is returned, otherwise `nil`.

9.28.10 Macro `swap`

Syntax:

```
(swap left-place right-place)
```

Description:

The `swap` macro exchanges the values of *left-place* and *right-place* and returns the value which is thereby transferred to *right-place*.

First, *left-place* and *right-place* are evaluated, in that order, to determine their locations.

Then the prior values are retrieved, exchanged and stored back. The value stored in *right-place* is also returned.

If *left-place* and *right-place* are ranges of the same sequence, the behavior is not specified if the ranges overlap or are of unequal length.

Note: the `rotate` macro's behavior is somewhat more specified in this regard. Thus, although any correct `swap` expression can be expressed using `rotate`, but the reverse isn't true.

9.28.11 Macro `push`

Syntax:

```
(push item place)
```

Description:

The `push` macro places *item* at the head of the list stored in *place* and returns the updated list which is stored back in *place*.

First, the expression *item* is evaluated to produce the push value. Then, *place* is evaluated to determine its storage location. Next, the storage location is accessed to retrieve the list value which is stored there. A new object is produced as if by invoking `cons` function on the push value and list value. This object is stored into the location, and returned.

9.28.12 Macro `pop`

Syntax:

```
(pop place)
```

Description:

The `pop` macro removes an element from the list stored in *place* and returns it.

First, *place* is evaluated to determine the place. The place is accessed to retrieve the original value. Then a new value is calculated, as if by applying the `cdr` function to the old value. This new value is stored. Finally, a return value is calculated and returned, as if by applying the `car` function to the original value.

9.28.13 Macro `pushnew`

Syntax:

```
(pushnew item place [testfun [keyfun]])
```

Description:

The `pushnew` macro inspects the list stored in *place*. If the list already contains the item, then it returns the list. Otherwise it creates a new list with the item at the front and stores it back into *place*, and returns it.

First, the expression *item* is evaluated to produce the push value. Then, *place* is evaluated to determine its storage location. Next, the storage location is accessed to retrieve the list value which is stored there. The list is inspected to check whether it already contains the push value, as if using the `member` function. If that is the case, the list is returned and the operation finishes. Otherwise, a new object is produced as if by invoking `cons` function on the push value and list value. This object is stored into the location and returned.

9.28.14 Macro `shift`

Syntax:

```
(shift place+ shift-in-value)
```

Description:

The `shift` macro treats one or more places as a "multi-place shift register". The values of the places are shifted one place to the left. The first (leftmost) place receives the value of the second place, the second receives that of the third, and so on. The last (rightmost) place receives *shift-in-value* (which is not treated as a place, even if it is a syntactic place form). The previous value of the first place is returned.

More precisely, all of the argument forms are evaluated left to right, in the process of which the storage locations of the places are determined, *shift-in-value* is reduced to its value.

The values stored in the places are sampled and saved.

Note that it is not specified whether the places are sampled in a separate pass after the evaluation of the argument forms, or whether the sampling is interleaved into the argument evaluation. This affects the behavior in situations in which the evaluation of any of the *place* forms, or of *shift-in-value*, has the side effect of modifying later places.

Next, the places are updated by storing the saved value of the second place into the first place, the third place into the second and so forth, and the value of *shift-in-value* into the last place.

Finally, the saved original value of the first place is returned.

If any of the places are ranges which index into the same sequence, and the behavior is not otherwise unspecified due to the issue noted in an earlier paragraph, the effect upon the multiply-stored sequence can be inferred from the above-described storage order. Note that even if stores take place which change the length of the sequence and move some elements, not-yet-processed stores whose ranges to refer to these elements are not adjusted.

With regard to the foregoing paragraph, a recommended practice is that if subranges of the same sequence object are shifted, they be given to the macro in ascending order of starting index. Furthermore, the semantics is simpler if the ranges do not overlap.

9.28.15 Macro `rotate`

Syntax:

```
(rotate place*)
```

Description:

Treats zero or more places as a "multi-place rotate register". If there are no arguments, there is no effect and `nil` is returned. Otherwise, the last (rightmost) place receives the value of the first (leftmost) place. The leftmost place receives the value of the second place, and so on. If there are two arguments, this equivalent to `swap`. The prior value of the first place, which is the value rotated into the last place, is returned.

More precisely, the *place* arguments are evaluated left to right, and the storage locations are thereby determined. The storage locations are sampled, and then the sampled values are stored back into the locations, but rotated by one place as described above. The saved original value of the leftmost *place* is returned.

It is not specified whether the sampling of the original values is a separate pass which takes place after the arguments are evaluated, or whether this sampling it is interleaved into argument evaluation. This affects the behavior in situations in which the evaluation of any of the *place* forms has the side effect of modifying the value stored in a later *place* form.

If any of the places are ranges which index into the same sequence, and the behavior is not otherwise unspecified due to the issue noted in the preceding paragraph, the effect upon the multiply-stored sequence can be inferred from the above-described storage order. Note that even if stores take place which change the length of the sequence and move some elements, not-yet-processed stores whose ranges to refer to these elements are not adjusted.

With regard to the foregoing paragraph, a recommended practice is that if subranges of the same sequence object are shifted, they be given to the macro in ascending order of starting index. Furthermore, the semantics is simpler if the ranges do not overlap.

9.28.16 Macro `del`

Syntax:

```
(del place)
```

Description:

The `del` macro requests the deletion of `place`. If `place` doesn't support deletion, an exception is thrown.

First `place` is evaluated, thereby determining its location. Then the place is accessed to retrieve its value. The place is then subject to deletion. Finally, the previously retrieved value is returned.

Precisely what deletion means depends on the kind of place. The built-in places in **TXR Lisp** have deletion semantics which are intended to be unsurprising to the programmer familiar with the data structure which holds the place.

Generally, if a place denotes the element of a sequence, then deletion of the place implies deletion of the element, and deletion of the element implies that the gap produced by the element is closed. The deleted element is effectively replaced by its successor, that successor by its successor and so on. If a place denotes a value stored in a dynamic data set such as a hash table, then deletion of that place implies deletion of the entry which holds that value. If the entry is identified by a key, that key is also removed.

9.28.17 Macro `lset`

Syntax:

```
(lset {place}+ sequence-expr)
```

Description:

The `lset` operator's parameter list consists of one or more places followed by an expression *sequence-expr*.

The macro evaluates `sequence-expr`, which is expected to produce a sequence.

Successive elements of the resulting list are then assigned to each successive `place`.

If there are fewer elements in the sequence than places, the unmatched places receive the value `nil`.

Excess elements in the sequence are ignored.

An error exception occurs if the sequence is an improper list with fewer elements than places.

A `lset` form produces the value of *`sequence-expr`* as its result value.

9.28.18 Macro `upd`

Syntax:

```
(upd place opip-arg*)
```

Description:

The `upd` macro evaluates *`place`* and passes the value as an argument to the operational pipeline function formed, as if by the `opip` macro, from the *`opip-arg`* arguments. The result of this function is then stored back into *`place`*.

The following equivalence holds, except that place `p` is evaluated only once:

```
(upd p x y z ...)  <-->  (set p (call (opip x y z ...) p))
```

9.29 User-Defined Places and Place Operators

TXR Lisp provides a number of place-modifying operators such as `set`, `push`, and `inc`. It also provides a variety of kinds of syntactic places which may be used with these operators.

Both of these categories are open-ended: **TXR Lisp** programs may extend the set of place-modifying operators, as well as the vocabulary of forms which are recognized as syntactic places.

Regarding place operators, it might seem obvious that new place operators can be developed, since they are macros, and macros can expand to uses of existing place operators. As an example, it may seem that `inc` operator could be written as a macro which uses `set`:

```
(defmacro new-inc (place : (delta 1))
  ^(set ,place (+ ,place ,delta)))
```

However, the above `new-inc` macro has a problem: the `place` argument form is inserted into two places in the expansion, which leads to two evaluations. This is visibly incorrect if the place form contains any side effects. It is also potentially inefficient.

TXR Lisp provides a framework for writing place update macros which evaluate their argument forms once, even if they have to access and update the same places.

The framework also supports the development of new kinds of place forms as capsules of code which introduce the right kind of material into the lexical environment of the body of an update macro, to enable this special evaluation.

9.29.1 Place-Expander Functions

The central design concept in **TXR Lisp** syntactic places are *place-expander functions*. Each compound place is defined by up to three place-expander functions, which are associated with the place via the leftmost operator symbol of the place form. One place-expander, the *update expander*, is mandatory. Optionally, a place may also provide a *clobber expander* as well as a *delete expander*. An update expander provides the expertise for evaluating a place form once in its proper run-time context to determine its actual run-time storage location, and to access and modify the storage location. A clobber expander provides an

optimized mechanism for uses that perform a one-time store to a place without requiring its prior value. If a place definition does not supply a clobber expander, then the syntactic places framework uses the update expander to achieve the functionality. A delete expander provides the expertise for determining the actual run-time storage location corresponding to a place, and obliterating it, returning its prior value. If a place does not supply a delete expander, then the place does not support deletion. Operators which require deletion, such as `del` will raise an error when applied to that place.

The expanders operate independently, and it is expected that place-modifying operators choose one of the three, and use only that expander. For example, accessing a place with an update expander and then overwriting its value with a clobber expander may result in incorrect code which contains multiple evaluations of the place form.

The programmer who implements a new place does not write expanders directly, but rather defines them via the `defplace`, `define-accessor` or `defset` macro.

The programmer who implements a new place update macro likewise does not call the expanders directly. Usually, they are invoked via the macros `with-update-expander`, `with-clobber-expander` and `with-delete-expander`. These are sufficient for most kind of macros. In certain complicated cases, expanders may be invoked using the wrapper functions `call-update-expander`, `call-clobber-expander` and `call-delete-expander`. These convenience macros and functions perform certain common chores, like macro-expanding the place in the correct environment, and choosing the appropriate function.

The expanders are described in the following sections.

9.29.2 The Update Expander

Syntax:

```
(lambda (getter-sym setter-sym place-form
         body-form) ...)
```

Description:

The update expander is a code-writer. It takes a *`body-form`* argument, representing code, and returns a larger form which surrounds this code with additional code.

This larger form returned by the update expander can be regarded as having two abstract actions, when it is substituted and evaluated in the context where *`place-form`* occurs. The first abstract action is to evaluate *`place-form`* exactly one time, in order to determine the actual run-time location to which that form refers. The second abstract action is to evaluate the caller's *`body-form`*s, in a lexical environment in which bindings exist for some lexical functions or (more usually) lexical macros. These lexical macros are explicitly referenced by the *`body-form`*; the update expander just provides their definition, under the names it is given via the *`getter-sym`* and *`setter-sym`* arguments.

The update expander writes local functions or macros under these names: a getter function and a setter function. Usually, update expanders write macros rather than functions, possibly in combination with some lexical anonymous variables which hold temporary objects. Therefore the getter and setter are henceforth referred to as macros.

The code being generated is with regard to some concrete instance of *`place-form`*. This argument is the actual form which occurs in a program. For instance, the update expander for the `car` place might be called with an arbitrary variant of the *`place-form`* which might look like `(car (inc (third some-list)))`.

In the abstract semantics, upfront code wrapped around the *body-form* by the update expander provides the logic to evaluate this place to a location, which is retained in some hidden local context.

The getter local macro named by *getter-sym* must provide the logic for retrieving the value of this place. The getter macro takes no arguments. The *body-form* makes free use of the getter function; they may call it multiple times, which must not trigger multiple evaluations of the original place form.

The setter local macro named by *setter-sym* must generate the logic for storing a new value into the once-evaluated version of *place-form*. The setter function takes exactly one argument, whose value specifies the value to be stored into the place. It is the caller's responsibility to ensure that the argument form which produces the value to be stored via the setter is evaluated only once, and in the correct order. The setter does not concern itself with this form. Multiple calls to the setter can be expected to result in multiple evaluations of its argument. Thus, if necessary, the caller must supply the code to evaluate the new value form to a temporary variable, and then pass the temporary variable to the setter. This code can be embedded in the *body-form* or can be added to the code returned by a call to the update expander.

The setter local macro or function must return the new value which is stored. That is to say, when *body-form* invokes this local macro or function, it may rely on it yielding the new value which was stored, as part of achieving its own semantics.

The update expander does not macro-expand `place-form`. It is assumed that the expander is invoked in such a way that the place has been expanded in the correct environment. In other words, the form matches the type of place which the expander handles. If the expander had to macro-expand the place form, it would sometimes have to come to the conclusion that the place form must be handled by a different expander. No such consideration is the case: when an expander is called on a form, that is final; it is certain that it is the correct expander, which matches the symbol in the `car` position of the form, which is not a macro in the context where it occurs.

An update expander is free to assume that any place which is stored (the setter local macro is invoked on it) is accessed at least once by an invocation of the getter. A place update macro which relies on an update expander, but uses only the store macro, might not work properly. An example of an update expander which relies on this assumption is the expander for the `(force` *promise*`)` place type. If *promise* has not yet been forced, and only the setter is used, then *promise* might remain unforced as its internal value location is updated. A subsequent access to the place will incorrectly trigger a force, which will overwrite the value. The expected behavior is that storing a value in an unforced `force` place changes the place to forced state, preempting the evaluation of the delayed form. Afterward, the promise exhibits the value which was thus assigned.

The update expander is not responsible for all issues of evaluation order. A place update macro may consist of numerous places, as well as numerous value-producing forms which are not places. Each of the places can provide its registered update expander which provides code for evaluating just that place, and a means of accessing and storing the values. The place update macro must call the place expanders in the correct order, and generate any additional code in the correct order, so that the macro achieves its required documented evaluation order.

Example Update Expander Call:

```
;; First, capture the update expander
;; function for (car ...) places
;; in a variable, for clarity.

(defvar car-update-expander [*place-update-expander* 'car])
```

```
;; Next, call it for the place (car [a 0]).
;; The body form specifies logic for
;; incrementing the place by one and
;; returning the new value.

(call car-update-expander 'getit 'setit '(car [a 0])
  '(setit (+ (getit) 1)))

;; --> Resulting code:

(rlet ((#:g0032 [a 0]))
  (macrolet ((getit nil
               (append (list 'car) (list '#:g0032)))
             (setit (val)
               (append (list 'sys:rplaca)
                       (list '#:g0032) (list val))))
    (setit (+ (getit) 1))))

;; Same expander call as above, with a call to expand added
;; to show the fully expanded version of the returned code,
;; in which the ;; setit and getit calls have disappeared,
;; replaced by their macro-expansions.

(expand
  (call car-update-expander 'getit 'setit '(car [a 0])
    '(setit (+ (getit) 1))))

;; --> Resulting code:

(let ((#:g0032 [a 0]))
  (sys:rplaca #:g0032 (+ (car #:g0032) 1)))
```

The main noteworthy points about the generated code are:

- the `(car [a 0])` place is evaluated by evaluating the embedded form `[a 0]` and storing storing the resulting object into a hidden local variable. That's as close a reference as we can make to the `car` field.
- the getter macro expands to code which simply calls the `car` function on the cell.
- the setter uses a system function called `sys:rplaca`, which differs from `rplaca` in that it returns the stored value, rather than the cell.

9.29.3 The Clobber Expander

Syntax:

```
(lambda (simple-setter-sym place-form
         body-form) ...)
```

Description:

The clobber expander is a code-writer similar to the update expander. It takes a *`body-form`* argument, and returns a larger form which surrounds this form with additional program code.

The returned block of code has one main abstract action. It must arrange for the evaluation of *`body-form`* in a lexical environment in which a lexical macro or lexical function exists which has the name requested by the *`simple-setter-sym`* argument.

The simple setter local macro written by the clobber expander is similar to the local setter written by the update expander. It has exactly the same interface, performs the same action of storing a value into the place, and returns the new value.

The difference is that its logic may be considerably simplified by the assumption that the place is being subject to exactly one store, and no access.

A place update macro which uses a clobber expander, and calls it more than once, break the assumption; doing so may result in multiple evaluations of the *place-form*.

9.29.4 The Delete Expander

Syntax:

```
(lambda (deleter-sym place-form
         body-form) ...)
```

Description:

The delete expander is a code-writer similar to clobber expander. It takes a *body-form* arguments, and returns a larger form which surrounds this form with additional program code.

The returned block of code has one main abstract action. It must arrange for the evaluation of *body-form* in a lexical environment in which a lexical macro or lexical function exists which has the name requested by the *deleter-sym* argument.

The deleter macro written by the clobber expander takes no arguments. It may be called at most once. It returns the previous value of the place, and arranges for its obliteration, whatever that means for that particular kind of place.

9.29.5 Macro `with-update-expander`

Syntax:

```
(with-update-expander (getter setter) place env
  body-form)
```

Description:

The `with-update-expander` macro evaluates the *body-form* argument, whose result is expected to be a Lisp form. The macro adds additional code around this code, and the result is returned. This additional code is called the *place-access code*.

The *getter* and *setter* arguments must be symbols. Over the evaluation of the *body-form*, these symbols are bound to the names of local functions which are provided in the place-access code.

The *place* argument is a form which evaluates to a syntactic place. The generated place-access code is based on this place.

The *env* argument is a form which evaluates to a macro-expansion-time environment. The `with-update-expander` macro uses this environment to perform macro-expansion on the value of the *place* form, to obtain the correct update expander function for the fully macro-expanded place.

The place-access code is generated by calling the update expander for the expanded version of `place`.

Example:

The following is an implementation of the `swap` macro, which exchanges the contents of two places.

Two places are involved, and, correspondingly, the `with-update-expander` macro is used twice, to add two instances of place-update code to the macro's body.

```
(defmacro swap (place-0 place-1 :env env)
  (with-gensyms (tmp)
    (with-update-expander (getter-0 setter-0) place-0 env
      (with-update-expander (getter-1 setter-1) place-1 env
        ^(let ((,tmp (,getter-0)))
           (,setter-0 (,getter-1))
           (,setter-1 ,tmp))))))
```

The basic logic for swapping two places is contained in the code template:

```
^(let ((,tmp (,getter-0)))
   (,setter-0 (,getter-1))
   (,setter-1 ,tmp))
```

The temporary variable named by the `gensym` symbol `tmp` is initialized by calling the getter function for *`place-0`*. Then the setter function of *`place-0`* is called in order to store the value of *`place-1`* into *`place-0`*. Finally, the setter for *`place-1`* is invoked to store the previously saved temporary value into that place.

The name for the temporary variable is provided by the `with-gensyms` macro, but establishing the variable is the caller's responsibility; this is seen as an explicit `let` binding in the code template.

The names of the getter and setter functions are similarly provided by the `with-update-expander` macros. However, binding those functions is the responsibility of that macro. To achieve this, it adds the place-access code to the code generated by the `^(let ...)` backquote template. In the following example macro-expansion, the additional code added around the template is seen. It takes the form of two `macrolet` binding blocks, each added by an invocation of `with-update-expander`:

```
(macroexpand '(swap a b))

-->

(macrolet ((#:g0036 () 'a)      ;; getter macro for a
           (#:g0037 (val-expr)  ;; setter macro for a
             (append (list 'sys:setq) (list 'a)
                     (list val-expr))))
  (macrolet ((#:g0038 () 'b)     ;; getter macro for b
             (#:g0039 (val-expr) ;; setter macro for b
               (append (list 'sys:setq) (list 'b)
                       (list val-expr))))
    (let ((#:g0035 (#:g0036)))  ;; temp <- a
      (#:g0037 (#:g0038))       ;; a <- b
      (#:g0039 #:g0035))))      ;; b <- temp
```

In this expansion, for example `#:g0036` is the generated symbol which forms the value of the

`getter-0` variable in the `swap` macro. The getter is a macro which simply expands to a `a`: straightforward access to the variable a. The `#:g0035` symbol is the value of the `tmp` variable. Thus the swap macro's `^(let ((,tmp (,getter-0))) ...)` has turned into `^(let ((#:g0035 (#:g0036))) ...)`

A full expansion, with the `macrolet` local macros expanded out:

```
(expand '(swap a b))

-->

(let ((#:g0035 a))
  (sys:setq a b)
  (sys:setq b #:g0035))
```

In other words, the original syntax `(,getter-0)` became `(#:g0036)` and finally just `a`.

Similarly, `(,setter-0 (,getter-1))` became the `macrolet` invocations `(#:g0037 (#:g0038))` which finally turned into: `(sys:setq a b)`.

9.29.6 Macro `with-clobber-expander`

Syntax:

```
(with-clobber-expander (simple-setter) place env
  body-form)
```

Description:

The `with-clobber-expander` macro evaluates *`body-form`*, whose result is expected to be a Lisp form. The macro adds additional code around this form, and the result is returned. This additional code is called the *place-access code*.

The *`simple-setter`* argument must be a symbol. Over the evaluation of the *`body-form`*, this symbol is bound to the name of a functions which are provided in the place-access code.

The *`place`* argument is a form which evaluates to a syntactic place. The generated place-access code is based on this place.

The *`env`* argument is a form which evaluates to a macro-expansion-time environment. The `with-clobber-expander` macro uses this environment to perform macro-expansion on the value of the *`place`* form, to obtain the correct update expander function for the fully macro-expanded place.

The place-access code is generated by calling the update expander for the expanded version of `place`.

Example:

The following implements a simple assignment statement, similar to `set` except that it only handles exactly two arguments:

```
(defmacro assign (place new-value :env env)
  (with-clobber-expander (setter) place env
    ^(,setter ,new-value)))
```

Note that the correct evaluation order of `place` and `new-value` is taken care of, because `with-clobber-expander` generates the code which performs all the necessary evaluations of `place`. This evaluation occurs before the code which is generated by `^(,setter ,new-value)` part is evaluated, and that code is what evaluates `new-value`.

Suppose that a macro were desired which allows assignment to be notated in a right to left style, as in:

```
(assign 42 a)  ;; store 42 in variable a
```

Now, the new value must be evaluated prior to the place, if left-to-right evaluation order is to be maintained. The standard `push` macro has this property: the push value is on the left, and the place is on the right.

Now, the code has to explicitly take care of the order, like this:

```
;; WRONG! We can't just swap the parameters;
;; place is still evaluated first, then new-value:

(defmacro assign (new-value place :env env)
  (with-clobber-expander (setter) place env
    ^(,setter ,new-value)))

;; Correct: arrange for evaluation of new-value first,
;; then place:

(defmacro assign (new-value place :env env)
  (with-gensym (tmp)
    ^(let ((,tmp ,new-value))
       ,(with-clobber-expander (setter) place env
         ^(,setter ,tmp)))))
```

9.29.7 Macro `with-delete-expander`

Syntax:

```
(with-delete-expander (deleter) place env
  body-form)
```

Description:

The `with-delete-expander` macro evaluates *`body-form`*, whose result is expected to be a Lisp form. The macro adds additional code around this code, and the resulting code is returned. This additional code is called the *place-access code*.

The *`deleter`* argument must be a symbol. Over the evaluation of the *`body-form`*, this symbol is bound to the name of a functions which are provided in the place-access code.

The *`place`* argument is a form which evaluates to a syntactic place. The generated place-access code is based on this place.

The *`env`* argument is a form which evaluates to a macro-expansion-time environment. The `with-delete-expander` macro uses this environment to perform macro-expansion on the value of the *`place`* form, to obtain the correct update expander function for the fully macro-expanded place.

The place-access code is generated by calling the update expander for the expanded version of `place`.

Example:

The following implements the `del` macro:

```
(defmacro del (place :env env)
  (with-delete-expander (deleter) place env
    ^(,deleter)))
```

9.29.8 Function `call-update-expander`

Syntax:

```
(call-update-expander getter setter place env
                      body-form)
```

Description:

The `call-update-expander` function provides an alternative interface for making use of an update expander, complementary to `with-update-expander`.

Arguments *getter* and *setter* are symbols, provided by the caller. These are passed to the update expander function, and are used for naming local functions in the generated code which the update expander adds to *body-form*.

The *place* argument is a place which has not been subject to macro-expansion. The `call-update-expander` function takes on the responsibility for macro-expanding the place.

The *env* parameter is the macro-expansion environment object required to correctly expand `place` in its original environment.

The *body-form* argument represents the source code of a place update operation. This code makes references to the local functions whose names are given by *getter* and *setter*. Those arguments allow the update expander to write these functions with the matching names expected by *body-form*.

The return value is an object representing source code which incorporates the *body-form*, augmenting it with additional code which evaluates `place` to determine its location, and provides place accessor local functions expected by the *body-form*.

Example:

The following shows how to implement a `with-update-expander` macro using `call-update-expander`:

```
(defmacro with-update-expander ((getter setter)
                                unex-place env body)
  ^(with-gensyms (,getter ,setter)
     (call-update-expander ,getter ,setter
                           ,unex-place ,env ,body)))
```

Essentially, all that `with-update-expander` does is to choose the names for the local functions, and bind them to the local variable names it is given as arguments. Then it calls `call-update-expander`.

Example:

Implement the swap macro using `call-update-expander`:

```
(defmacro swap (place-0 place-1 :env env)
  (with-gensyms (tmp getter-0 setter-0 getter-1 setter-1)
    (call-update-expander getter-0 setter-0 place-0 env
      (call-update-expander getter-1 setter-1 place-1 env
        ^(let ((,tmp (,getter-0)))
           (,setter-0 (,getter-1))
           (,setter-1 ,tmp))))))
```

9.29.9 Function `call-clobber-expander`

Syntax:

```
(call-clobber-expander simple-setter place env
                       body-form)
```

Description:

The `call-clobber-expander` function provides an alternative interface for making use of a clobber expander, complementary to `with-clobber-expander`.

Argument *`simple-setter`* is a symbol, provided by the caller. It is passed to the clobber expander function, and is used for naming a local function in the generated code which the update expander adds to *`body-form`*.

The *`place`* argument is a place which has not been subject to macro-expansion. The `call-clobber-expander` function takes on the responsibility for macro-expanding the place.

The *`env`* parameter is the macro-expansion environment object required to correctly expand `place` in its original environment.

The *`body-form`* argument represents the source code of a place update operation. This code makes references to the local function whose name is given by *`simple-setter`*. That argument allows the update expander to write this function with the matching name expected by *`body-form`*.

The return value is an object representing source code which incorporates the *`body-form`*, augmenting it with additional code which evaluates `place` to determine its location, and provides the clobber local function to the *`body-form`*.

9.29.10 Function `call-delete-expander`

Syntax:

```
(call-delete-expander deleter place env body-form)
```

Description:

The `call-delete-expander` function provides an alternative interface for making use of a delete expander, complementary to `with-delete-expander`.

Argument *`deleter`* is a symbol, provided by the caller. It is passed to the delete expander function, and is used for naming a local function in the generated code which the update expander adds to *`body-form`*.

The *place* argument is a place which has not been subject to macro-expansion. The `call-delete-expander` function takes on the responsibility for macro-expanding the place.

The *env* parameter is the macro-expansion environment object required to correctly expand `place` in its original environment.

The *body-form* argument represents the source code of a place delete operation. This code makes references to the local function whose name is given by *deleter*. That argument allows the update expander to write this function with the matching name expected by *body-form*.

The return value is an object representing source code which incorporates the *body-form*, augmenting it with additional code which evaluates `place` to determine its location, and provides the delete local function to the *body-form*.

9.29.11 Macro `define-modify-macro`

Syntax:

```
(define-modify-macro name parameter-list function-name)
```

Description:

The `define-modify-macro` macro provides a simplified way to write certain kinds of place update macros. Specifically, it provides a way to write place update macros which modify a place by retrieving the previous value, pass it through a function (perhaps together with some additional arguments), and then store the resulting value back into the place and return it.

The *name* parameter specifies the name for the place update macro to be written.

The *function-name* parameter must specify a symbol: the name of the update function.

The update macro and update function both take at least one parameter: the place to be updated, and its value, respectively.

The *parameter-list* specifies the additional parameters for the update function, which will also become additional parameters of the macro. Because it is a function parameter list, it cannot use the special destructuring features of macro parameter lists, or the `:env` or `:whole` special parameters. It can use optional parameters, and may be empty.

The `define-modify-macro` macro writes a macro called *name*. The leftmost parameter of this macro is a place, followed by the additional arguments specified by *parameter-list*. The macro will arrange for the evaluation of the place argument to determine the place location. It will then retrieve and save the prior value of the place, and evaluate the remaining arguments. The prior value of the place, and the values of the additional arguments, are all passed to *function* and the resulting value is then stored back into the location previously determined for *place*.

Example:

Some standard place update macros are implementable using `define-modify-macro`, such as `inc`.

The `inc` macro reads the old value of the place, then passes it through the + (plus) function, along with an extra argument: the delta value, which defaults to one. The `inc` macro could be written using `define-modify-macro` as follows:

```
(define-modify-macro inc (: (delta 1)) +)
```

Note that the argument list `(: (delta 1))` doesn't specify the place, because the place is the implicit leftmost argument of the macro which isn't given a name. With the above definition in place, when `(inc (car a))` is invoked, then `(car a)` is first reduced to a location, and that location's value is retrieved and saved. Then the `delta` parameter s evaluated to its value, which has defaulted to 1, since the argument was omitted. Then these two values are passed to the + function, and so 1 is added to the value previously retrieved from `(car a)`. The resulting sum is then stored back `(car a)` without evaluating `(car a)` again.

9.29.12 Macro `defplace`

Syntax:

```
(defplace place-destructuring-args body-sym
   (getter-sym setter-sym update-body)
   [(ssetter-sym clobber-body)
    [(deleter-sym delete-body)]])
```

Description:

The `defplace` macro is used to introduce a new kind of syntactic place. It writes the update expander, and optionally clobber and delete expander functions, from a simpler, more compact specification, and automatically registers the resulting functions. The compact specification of a `defplace` call contains only code fragments for the expander functions.

The name and syntax of the place is determined by the *place-destructuring-args* argument, which is macro-style parameter list whose structure mimics that of the place. In particular, its leftmost symbol gives the name under which the place is registered. The `defplace` macro provides automatic destructuring of the syntactic place, so that the expander code fragments can refer to the components of a place by name.

The *body-sym* parameter must be be a symbol. This symbol will capture the *body-forms* parameter which is passed to the update expander, clobber expander or delete expander. The code fragments then have access to the body forms via this name.

The *getter-sym*, *setter-sym*, and *update-body* parenthesized triplet specify the update expander fragment. The `defplace` macro will bind *getter-sym* and *setter-sym* to symbols. The *update-body* must then specify a template of code which evaluates the syntactic place to determine its storage location, and provides a pair of local functions, using these two symbols as their name. The template must also insert the *body-sym* forms into the scope of these local functions, and the place determining code.

The *setter-sym* and *clobber-body* arguments similarly specify an optional clobber expander fragment, as a single optional argument. If specified, the *clobber-body* must generate a local function named using *setter-sym* wrapped around *body-sym* forms.

The *deleter-sym* and *deleter-body* likewise specify a delete expander fragment. If this is omitted, then the place shall not support deletion.

Example:

Implementation of the place denoting the `car` field of `cons` cells:

```
(defplace (car cell) body

  ;; the update expander fragment
  (getter setter
```

```
      (with-gensyms (cell-sym) ;; temporary symbol for cell
        ^(let ((,cell-sym ,cell)) ;; evaluate place to cell
           ;; getter and setter access cell via temp var
           (macrolet ((,getter ()
                        ^(car ,',cell-sym))
                      (,setter (val)
                        ^(sys:rplaca ,',cell-sym ,val)))

             ;; insert body form from place update macro
             ,body))))

    ;; clobber expander fragment: simpler: no need
    ;; to evaluate cell to temporary variable.
    (ssetter
      ^(macrolet ((,ssetter (val)
                    ^(sys:rplaca ,',cell ,val)))
        ,body))

    ;; deleter: delegate to pop semantics:
    ;; (del (car a)) == (pop a).
    (deleter
      ^(macrolet ((,deleter () ^(pop ,',cell)))
         ,body)))
```

9.29.13 Macro `defset`

Syntax:

```
(defset name params new-val-sym set-form)
(defset get-fun-sym set-fun-sym)
```

Description:

The `defset` macro provides a mechanism for introducing a new kind of syntactic place. It is simpler to use than `defplace` and more concise, but not as general.

The `defset` macro is designed for situations in which a function or macro which evaluates all of its arguments is required to serve as a syntactic place. It provides two flavors of syntax: the long form, indicated by giving `defset` five arguments, and a short form, which uses two arguments.

In the long form of `defset`, the syntactic place is described by *name* and *params*. The `defset` form expresses the request that a call to the function or operator named *name* be treated as a syntactic place, which has arguments described by the parameter list *params*.

The *set-form* argument specifies an expression which generates the code for storing a new value to the place.

The `defset` macro makes the necessary arrangements such that when an operator form named by *name* is treated as a syntactic place, then at macro-expansion time, code is generated to evaluate all of its argument expressions into machine-generated variables. The names of those variables are automatically bound to the corresponding symbols given in the *params* argument list of the `defset` syntax. Code is also generated to evaluate the expression which gives the new value to be stored, and that is bound to a generated variable whose name is bound to the `new-val-sym` symbol. Then arrangements are made to invoke the operator named by *name* and to evaluate the `set-form` in an environment in which these symbol bindings are visible. The operator named *name* is invoked using an altered argument list which uses temporary symbols in place of the original

expressions. The task of `set-form` is to insert the values of the symbols from *`params`* and *`new-val-sym`* into a suitable code templates that will perform the store actions. The code generated by `set-form` must also take on the responsibility of yielding the new value as its result.

If *`params`* list contains optional parameters, the default value expressions of those parameters shall be evaluated in the scope of the `defset` definition.

The *`params`* list may specify a rest parameter. In the expansion, this parameter will capture a list of temporary symbols, corresponding to the list of variadic argument expressions. For instance if the `defset` parameter list for a place `g` is `(a b . c)`, featuring the rest parameter `c`, and its *`set-form`* is `^(s ,a ,b ,*c)` and the place is invoked as `(g (i) (j) (k) (l))` then parameter `c` will be bound to a list of gensyms such as `(#:g0123 #:g0124)` so that the evaluation of *`set-form`* will yield syntax resembling `(s #:g0121 #:g0122 #:g0123 #:g0124)`. Here, gensyms `#:g0123` and `#:g0124` are understood to be bound to the values of the expressions `(k)` and `(l)`, the two trailing parameters corresponding to the rest parameter `c`.

Syntactic places defined by `defset` that have a rest parameter may be invoked with improper syntax such as `(set (g x y . z) v)`. In this situation, that rest parameter will be bound to the name of a temporary variable which holds the value of `z` rather than to a list of temporary variable names holding the values of trailing expressions. The `set-form` must be prepared for this situation. In particular, the rest parameter's value is an atom, then it cannot be spliced in the backquote syntax, except at the last position of a list.

Although syntactic places defined by `defset` perform macro-parameter-like destructuring of the place form, binding unevaluated argument expressions to the parameter symbols, nested macro parameter lists are not supported: *`params`* specifies a function parameter list.

The parameter list may use parameter macros, keeping in mind that the parameter expansion is applied at the time the `defset` form is processed, specifying an expanded parameter list which receives unevaluated expressions. The *`set-form`* may refer to all symbols produced by parameter list expansion, other than generated symbols. For instance, if a parameter list macro `:addx` exists which adds the parameter symbol `x` to the parameter list, and this `:addx` is invoked in the *`params`* list of a `defset`, then `x` will be visible to the *`set-form`*.

The short, two-argument form of `defset` simply specifies the names of two functions or operators: `get-fun-sym` names the operator which accesses the place, and `set-fun-sym` names the operator which stores a new value into the place. It is expected that all arguments of these operators are evaluated expressions, and that the store operator takes one argument more than the access operator. The operators are otherwise assumed to be variadic: each instance of a place based on `get-fun-sym` individually determines how many arguments are passed to that operator and to the one named by `set-fun-sym`.

The definition `(defset g s)` means that `(inc (g x y))` will generate code which ensures that `x` and `y` are evaluated exactly once, and then those two values are passed as arguments to `g` which returns the current value of the place. That value is then incremented by one, and stored into the place by calling the `s` function/operator with three arguments: the two values that were passed to `g` and the new value. The exact number of arguments is determined by each individual use of `g` as a place; the `defset` form doesn't specify the arity of `g` and `s`, only that `s` must accept one more argument relative to `g`.

The following equivalence holds between the short and long forms:

```
(defset g s)  <-->  (defset g (. r) n ^(g ,*r) ^(s ,*r ,n))
```

Note: the short form of `defset` is similar to the `define-accessor` macro.

Example:

Implementation of `car` as a syntactic place using a long form `defset`:

```
(defset car (cell) new
  (let ((n (gensym)))
    ^(rlet ((,n ,new))
       (progn (rplaca ,cell ,n) ,n))))
```

Given such a definition, the expression `(inc (car (abc)))` expands to code closely resembling:

```
(let ((#:g0048 (abc)))
  (let ((#:g0050 (succ (car #:g0048))))
    (rplaca #:g0048 #:g0050)
    #:g0050))
```

The `defset` macro has arranged for the argument expression `(abc)` of `car` to be evaluated to a temporary variable `#:g0048`, a `gensym`. This, then, holds the `cons` cell being operated on. At macro-expansion time, the variable `cell` from the parameter list specified by the `defset` is bound to this symbol. The access expression `(car #:0048)` to retrieve the prior value is automatically generated by combining the name of the place `car` with the gensym to which its argument `(abc)` has been evaluated. The `new` variable was bound to the expression giving the new value, namely `(succ (car #:g0048))`. The *`set-form`* is careful to evaluate this only one time, storing its value into the temporary variable `#:g0050`, referenced by the variable `n`. The *`set-form`*'s `(rplaca ,cell ,n)` fragment thus turned into `(rplaca #:g0048 #:g0050)` where `#:g0048` references the cons cell being operated on, and `#:g0050` the calculated new value to be stored into its `car` field. The *`set-form`* is careful to arrange for the new value `#:g0050` to be returned. Those place-mutating operators which yield the new value, such as `set` and `inc` rely on this behavior.

9.29.14 Macro `define-place-macro`

Syntax:

```
(define-place-macro name macro-style-params
   body-form*)
```

Description:

In some situations, an equivalence exists between two forms, only one of which is recognized as a place. The `define-place-macro` macro can be used to establish a form as a place in terms of a translation to an equivalent form which is already a place.

The `define-place-macro` has the same syntax as `defmacro`. It specifies a macro transformation for a compound form which has the *`name`* symbol in its leftmost position.

Place macro expansion doesn't use an environment; place macros are in a single global namespace, special to place macros. There are no lexically scoped place macros. Such an effect can be achieved by having a place macro expand to an a form which is the target of a global or local macro, as necessary.

To support place macros, forms which are used as syntactic places are subject to a modified macro-expansion algorithm:

1. If a place macro exists for a form that is being used as a place, then the that place macro is invoked to expand the form, and the expansion is taken in place of the original form. This process repeats until the form can no longer be expanded as a place macro, or the place macro declines to expand the form by returning the unexpanded input.
2. A form that has been fully expanded as a place macro is then subject to a single-round of macro-expansion, as if by `macroexpand-1`, which takes place in the original form's lexical environment. If the form doesn't expand, or the result of expansion is `nil` or a non-symbolic atom, then the process terminates. Otherwise, the process is repeated from step 1.

The `define-place-macro` macro does not cause *name* to become `mboundp`.

There can exist both an ordinary macro and a place macro of the same name. In this situation, when the macro call appears as a place form, it is expanded as a place macro, according to the above steps. When the macro call appears as an evaluated form, not being used as a place, the form is expanded using the ordinary macro.

Example:

Implementation of `first` in terms of `car`:

```
(define-place-macro first (obj)
  ^(car ,obj))
```

9.29.15 Functions `macroexpand-place` and `macroexpand-1-place`

Syntax:

```
(macroexpand-1-place form [env])
(macroexpand-place form [env])
```

Description:

If *form* is a place macro form (a form whose operator symbol has been defined as a place macro using `define-place-macro`) these functions expand the place macro form and return the expanded form. Otherwise, they return *form*.

`macroexpand-1-place` performs a single expansion, expanding only the place the macro that is referenced by the symbol in the first position of *form*, and returns the expansion. Note that if *form* is an ordinary macro form, this function will not expand it, even if such an expansion would reveal a place macro form.

`macroexpand-place` performs a full place expansion of *form* by the following process. If *form* is a place macro call, it is expanded, and the result is checked again to see whether it is a place macro, and expanded. This is repeated as many times as necessary until the result is no longer a place macro call. Then, if the resulting form is an ordinary macro invocation, it is expanded once as if by `macroexpand-1`. This process is iterated until a fixed point is reached.

The optional *env* parameter is a macro environment. Note: the `macroexpand-1-place` function ignores the *env* parameter, which could change in the future.

Examples

Given this ordinary macro definition

```
(defmacro leftmost (x) ^(first ,x))
```

the following results are obtained:

```
;; ordinary macro leftmost expands to first,
;; then first place macro expands to car:
(macroexpand-place '(leftmost x)) -> (car x)

;; macroexpand-1-place won't expand ordinary macro:
(macroexpand-1-place '(leftmost x)) -> (leftmost x)

;; macroexpand-1-place expands place macro
(macroexpand-1-place '(first x)) -> (car x)
```

9.29.16 Macro `rlet`

Syntax:

```
(rlet ({(sym init-form)}*) body-form*)
```

Description:

The macro `rlet` is similar to the `let` operator. It establishes bindings for one or more *sym*s, which are initialized using the values of *init-form*s.

Note that the simplified syntax for a variable which initializes to `nil` by default is not supported by `rlet`; that is to say, the syntax *sym* cannot be used in place of the `(`*sym init-form*`)` syntax when *sym* is to be initialized to `nil`.

The `rlet` macro differs from `let` in that `rlet` assumes that those *sym*s whose *init-form*s, after macro expansion, are constant expressions (according to the `constantp` function) may be safely implemented as a symbol macro rather than a lexical variable.

Therefore `rlet` is suitable in situations in which simpler code is desired from the output of certain kinds of machine-generated code, which binds local symbols: code with fewer temporary variables.

On the other hand, `rlet` is not suitable in some situations when true variables are required, which are assignable, and provide temporary storage.

Example:

```
;; WRONG! Real storage location needed.
(rlet ((flag nil))
  (flip flag)) ;; error: flag expands to nil

;; Demonstration of constant-propagation
(let ((a 42))
  (rlet ((x 1)
         (y a))
    (+ x y)))  -->  43

(expand
  '(let ((a 42))
    (rlet ((x 1)
           (y a))
      (+ x y))))  -->  (let ((a 42))
                         (let ((y a))
```

```
(+ 1 y)))
```

The last example shows that the `x` variable has disappeared in the expansion. The `rlet` macro turned it into into a `symacrolet` denoting the constant 1, which then propagated to the use site, turning the expression `(+ x y)` into `(+ 1 y)`.

9.29.17 Macro `slet`

Syntax:

```
(slet ({(sym init-form)}*) body-form*)
```

Description:

The macro `slet` is a stronger form of the `rlet` macro. Just like `rlet`, `slet` reduces bindings initialized by constant expressions to symbol macros. In addition, unlike `rlet`, `slet` also reduces to symbol macros those bindings whose initializing expressions are simple references to lexical variables.

Examples:

```
;; reduces to let
(slet ((a (list x y)))
  a)

;; b is a free variable, so this is also let
(slet ((a b))
  a)

;; b is lexical, so a becomes a symbol macro
;; the (slet ...) form becomes b.
(let (b)
  (slet ((a b))
    a))

;; a becomes symbol macro; form transforms to 1.
(slet ((a 1))
  a)
```

9.29.18 Macro `alet`

Syntax:

```
(alet ({(sym init-form)}*) body-form*)
```

Description:

The macro `alet` ("atomic" or "all") is a stronger form of the `slet` macro. All bindings initialized by constant expressions are turned to symbol macros. Then, if all of the remaining bindings are all initialized by lexical variables, they are also turned to symbol macros. Otherwise, none of the remaining bindings are turned to symbol macros.

The `alet` macro can be used even in situations when it is possible that the initializing forms the variables may have side effects through which they affect each others' evaluations. In this situation `alet` still propagates constants via symbol macros, and can eliminate the remaining temporaries if they can all be made symbol macros for existing lexicals: i.e. there doesn't exist any initialization form with interfering side effects.

9.29.19 Macro `define-accessor`

Syntax:

```
(define-accessor get-function set-function)
```

Description:

The `define-accessor` macro is used for turning a function into an accessor, such that forms which call the function can be treated as places.

Arguments to `define-accessor` are two symbols, which must name functions. When the `define-accessor` call is evaluated, the *`get-function`* symbol is registered as a syntactic place. Stores to the place are handled via calls to *`set-function`*.

If *`get-function`* names a function which takes N arguments, *`set-function`* must name a function which takes N+1 arguments.

Moreover, in order for the accessor semantics to be correct *`set-function`* must treat its rightmost argument as the value being stored, and must also return that value.

When a function call form targeting *`get-function`* is treated as a place which is subject to an update operation (for instance an increment via the `inc` macro), the accessor definition created by `define-accessor` ensures that the arguments of *`get-function`* are evaluated only once, even though the update involves a call to *`get-function`* and *`set-function`* with the same arguments. The argument forms are evaluated to temporary variables, and these temporaries are used as the arguments in the calls.

No other assurances are provided by `define-accessor`.

In particular, if *`get-function`* and *`set-function`* internally each perform some redundant calculation over their arguments, this cannot be optimized. Moreover, if that calculation has a visible effect, that effect is observed multiple times in an update operation.

If further optimization or suppression of multiple effects is required, the more general `defplace` macro must be used to define the accessor. It may also be possible to treat the situation in a satisfactory way using a `define-place-macro` definition, which effectively then supplies inline code whenever a certain form is used as a place, and that code itself is treated as a place.

Note: `define-accessor` is similar to the short form of `defset`.

9.29.20 Accessor `read-once`

Syntax:

```
(read-once expression)
(set (read-once place) new-value)
```

Description:

When the `read-once` accessor is invoked as a function, it behaves like `identity`, simply returning the value of *`expression`*, which is not required to be a syntactic place.

If a `read-once` form is used as a syntactic place then its argument must also be a *`place`*. The `read-once` syntactic place denotes the same place as the enclosed `place` form, but with somewhat altered semantics, which is most useful in conjunction with `placelet`, and in writing place-mutating macros which make multiple accesses to a place.

Firstly, if the `read-once` place is evaluated, it accesses the existing value of *`place`* exactly once, even if it occurs in a place-mutating form which normally doesn't use the prior value, such as the `set` macro.

When `read-once` accesses *`place`*, it stores the value in a hidden variable. Then, within the same place-mutating form, multiple references to the same `read-once` form all access the value of this hidden variable. Whenever the `read-once` form is assigned, both the the hidden variable and the underlying *`place`* receive the new value.

Multiple references to the same `read-once` form can be produced using the `placelet` or `placelet*` macros, or by making multiple calls to the getter function obtained using `with-update-expander` in the implementation of a user-defined place-mutating operator, or user-defined place.

Example:

In both of the following two examples, there is no question that the `array` and `i` expressions are themselves evaluated only once; the issue is the access to the array itself; under the plain placelet, the array referencing takes place more times.

```
;; without read-once, array element [array i] is
;; accessed twice to fetch its current value: once
;; in the plusp expression, and then once again in
;; the dec expression.

(placelet ((cell [array i]))
  (if (plusp cell)
    (dec cell)))

;; with read-once, it is accessed once. plusp refers
;; to a hidden lexical variable to obtain the prior
;; value, and so does dec. dec stores the new value
;; through to [array i] and the hidden variable.

(placelet ((cell (read-once [array i]))))
  (if (plusp cell)
    (dec cell)))
```

The following is **not** an example of multiple references to the same `read-once` form:

```
(defmacro inc-positive (place)
  ^(if (plusp (read-once ,place))
     (inc (read-once ,place))))
```

Here, even though the `read-once` forms may be structurally identical, they are separate instances. The first instance isn't even a syntactic place, but a call to the `read-once` function. Multiple references to the same place can only be generated using `placelet` or else by multiple explicit calls to the same getter function or macro generated for a place by an update expander.

The following is a corrected version of `inc-positive`:

```
(defmacro inc-positive (place :env env)
  (with-update-expander (getter setter) ^(read-once ,place) env
    ^(if (plusp (,getter))
       (,setter (succ (,getter))))))
```

To write the macro without `read-once` requires that it handles the job of providing a temporary variable for the value:

```
(defmacro inc-positive (place :env e)
  (with-update-expander (getter setter) place env
    (with-gensym (value)
      ^(slet ((,value (,getter)))
        ^(if (plusp ,value)
           (,setter (succ ,value)))))))
```

The `read-once` accessor wrapped around *`place`* allows `inc-positive` to simply make multiple references to `(,getter)` which will cache the value; the macro doesn't have to introduce its own hidden caching variable.

9.29.21 Special variables `*place-update-expander*`, `*place-clobber-expander*` and `*place-delete-expander*`

Description:

These variables hold hash tables, by means of which update expanders, clobber expanders and delete expanders are registered, as associations between symbols and functions.

If `[*place-update-expander* 'sym]` yields a function, then symbol `sym` is the basis for a syntactic place. If the expression yields `nil`, then forms beginning with `sym` are not syntactic places. (The situation of a clobber accessor or delete accessor being defined without an update expander is improper).

9.29.22 Special variable `*place-macro*`

Description:

The `*place-macro*` special variable holds the hash table of associations between symbols and place macro expanders.

If the expression `[*place-macro* 'sym]` yields a function, then symbol `sym` has a binding as a place macro. If that expression yields `nil`, then there is no such binding: compound forms beginning with `sym` do not undergo place macro expansion.

9.30 Structural Pattern Matching

9.30.1 Introduction

TXR Lisp provides a structural pattern-matching system. Structural pattern matching is a syntax which allows for the succinct expression of code which classifies objects according to their shape and content, and which accesses the elements within objects, or both.

The central concept in structural pattern matching is the resolution of a pattern against an object. The pattern is specified as syntax which is part of the program code. The object is a run-time value of unknown type, shape and other properties. The primary pattern-matching decision is Boolean: does the object match the pattern? If the object matches the pattern, then it is possible to execute an associated body of code in a scope in which variables occurring in the pattern take on values from the corresponding parts of the object.

9.30.2 Pattern-Matching Operators

Structural pattern matching is available via several different macro operators, which are: `when-match`, `if-match`, `match`, `match-case`, `match-ecase`, `lambda-match` and `defun-match`. Function

and macro argument lists may also be augmented with pattern matching using the `:match` parameter macro.

The `when-match` macro is the simplest. It tests an object against a pattern, and if there is a match, evaluates zero or more forms in an environment in which the pattern variables have bindings to the corresponding elements of the object.

The `if-match` macro evaluates a single form if there is a match, in the scope of the bindings established by the pattern, otherwise an alternative form evaluated in a scope in which those bindings are absent.

The `match` macro tests and object against a pattern, expecting a match. If the match fails, an exception is thrown. Otherwise, it evaluates zero or more forms in the scope of the bindings established by the pattern.

The `match-case` macro evaluates the same object against multiple clauses, each consisting of a pattern and zero or more forms. The first case whose pattern matches the object is selected. The forms associated with a matching clause are evaluated in the scope the variables bound by that clause's pattern.

The `match-ecase` macro is similar to `match-case` except that if no matching case is identified, an exception is thrown.

The `lambda-match` macro provides a way to express an anonymous function whose argument list is matched against multiple clauses similarly to `match-case` and `defun-match` provides a way to define a top-level function using the same concept.

Additionally, there exist `each-match` and `while-match` macro families.

9.30.3 Syntax and Key Concepts

TXR Lisp's structural pattern-matching notation is template-based. With the exception of structures and hash tables, objects are matched using patterns which are based on their printed notation. For instance, the pattern `(1 2 @a)` is a pattern matching the list `(1 2 3)` binding `a` to `3`. The notation supports lists, vectors, ranges and atoms. Atoms are compared using the `equal` function. Thus, in the above pattern, the 1 and 2 in the pattern match the corresponding 1 and 2 atoms in the object using `equal`.

All parts of a pattern are static material which matches literally, except those parts introduced by the meta prefix `@`. This prefix denotes variables like `@a` as well as useful pattern-matching operators like `@(all` *`pattern`*`)` which matches a list or sublist whose elements all match *`pattern`*.

The quasiquote syntax is specially supported for expressing matching, in an alternative style. For instance the quasiquote `^(1 2 ,a)` is a pattern equivalent to the `(1 2 @a)`.

Structure objects are matched using a dedicated `@(struct name ...)` operator, or else in the quasiquote style using `^#S(name ...)` syntax. The non-quasiquoted literal syntax `#S(name ...)` cannot be used for matching.

Similarly, hash objects are matched using a `@(hash ...)` operator, or else `^#H(...)` syntax in the quasiquote style. `#H(...)` cannot be used.

Note: the non-quasiquoted `#S` and `#H` literals are not and cannot be used for matching because they produce structure and hash objects which lose important information about how they were specified in the syntax, and carry restrictions which are unacceptable for pattern matching. The order of sub-patterns is important in pattern syntax, but struct and hash objects do not preserve the order in which their elements were specified. A struct literal is required to specify the name of an existing struct type, and slot names which are valid for that type, otherwise it is erroneous. This is not acceptable for pattern matching, because patterns may appear in place of those elements. The pattern match for a hash may specify the same key pattern

more than once, which means that the key pattern cannot be an actual key in an actual hash, which requires every key to be unique. Structure and hash quasiquotes do not have these issues; they are not actually literal structure and hash objects, but list-based syntax.

9.30.4 Variables in Patterns

Patterns use meta-symbols for denoting variables. Variables must be either bindable symbols, or else `nil`, which has a special meaning: the pattern variable `@nil` matches any object, and binds no variable.

Pattern variables are ordinary Lisp variables. Whereas in ordinary non-pattern matching Lisp code, it is always unambiguous whether a variable is being bound or referenced, this is deliberately not the case in patterns. A variable occurring in a pattern may be a fresh variable, or a reference to an existing one. The difference between these situations is not apparent from the syntax of the pattern; it depends on the context established by the scope.

With one exception, if a pattern contains a variable which is already in the surrounding scope, including a global variable, then it refers to that variable. Otherwise, it freshly binds the variable. The exception is that pattern operator `@(as)` always binds a fresh variable.

When a pattern variable refers to an existing variable, then each occurrence of that variable must match an object which is `equal` to the value of that variable. For instance, the following function returns the third element of a list, if the first two elements are repetitions of the `x` argument, otherwise `nil`:

```
(defun x-x-y (list x)
  (when-match (@x @x @y) list y))

(x-x-y '(1 1 2) 1) -> 2
(x-x-y '(1 2 3) 1) -> nil    ;; no @x @x match
(x-x-y '(1 1 2 r2) 1) -> nil ;; list too long
```

If the variable does not exist in the scope surrounding the pattern, then the leftmost occurrence of the variable establishes a binding, taking the value from is corresponding object being matched by that occurrence of the variable. The remaining occurrences of the variable, if any, must correspond to objects which are `equal` to that value, or else there is no match. For instance, the pattern `(@a @a)` matches the list like `(1 1)` as follows. First `@a` binds to the leftmost `1` and then the second `1` matches the existing value of that `a`. An input such as `(1 2)` fails to match because the second occurrence of `@a` retrieves an object that is not `equal` to that variable's existing value.

A pattern can contain multiple occurrences of the same symbol as a variable. These may or may not refer to the same variable. Two occurrences of the same symbol refer to distinct variables if:

1. they are freshly bound in separate branches of the `@(or)` operator; or
2. one of the two variables is freshly bound by the `@(as)` operator and the other variable occurs outside of that `@(as)`; or
3. or both of the variables are freshly bound using `@(as)`.

Any other two or more occurrences same symbol occurring in the same pattern refer to the same variable.

9.30.5 Comparison to Macro Parameter Lists

TXR Lisp's macro-style parameter lists, appearing in `tree-bind` and related macros, also provide a form of structural pattern matching. Macro-style parameter list pattern matching is limited to objects of one kind: tree structures made of `cons` cells. It is only useful for matching on shape, not content. For

example, `tree-bind` cannot express the idea of matching a list whose first element is the symbol `a` and whose third element is `42`. Moreover, every position in the tree pattern much specify a variable which captures the corresponding element of the structure. For instance, a pattern which matches a three-element list must specify three variables, one for each list position. This is because macro-style parameter lists are oriented toward writing macros, and macros usually make use of every parameter position.

9.30.6 User-defined Patterns

User-defined pattern operators are possible. When the *operator* symbol in the `@(`*operator argument**`)` syntax doesn't match any built-in operator, a search takes place to determine whether *operator* is a pattern macro. If so, the pattern macro is expanded, and its result of the expansion treated as a pattern to process recursively, unless it is the original macro form, in which case it is treated as a predicate pattern. User-defined pattern macros are defined using the `defmatch` macro.

9.31 Pattern-Matching Notation

The pattern-matching notation is documented in the following sections; a section describing the pattern-matching macros follow.

9.31.1 Atom match

A pattern consisting of an atom other than a vector matches a similar object. The similarity is determined using the `equal` function.

The atom is not subject to evaluation, which means that a symbolic atom stands for itself, and not the value of a variable.

Examples:

```
;; the pattern 1 matches the object 1
(if-match 1 1 'yes 'no) --> yes

;; the object 0 does not match
(if-match 1 0 'yes 'no) --> no

;; a matches a, does not match b
(let ((sym 'a))
  (list (if-match a sym 'yes 'no)
        (if-match b sym 'yes 'no)))
--> (yes no)
```

9.31.2 Variable match

Syntax:

`@`*symbol*

Description:

A meta-symbol can be used as a pattern expression. This pattern unconditionally matches an object of any kind.

The *symbol* is required to be a either a bindable symbol according to the `bindable` function, or else the symbol `nil`.

If *symbol* is a bindable symbol, which has not binding in scope, then a variable by that name is freshly bound, and takes on the corresponding object as its value.

If *symbol* is a bindable symbol with an existing binding, then the corresponding object must be `equal` to that variable's existing value, or else the match fails.

If *symbol* is `nil`, then the match succeeds unconditionally, without binding a variable.

Examples:

```
(when-match @a 42 (list a)) -> (42)

(when-match (@a @b @c) '(1 2 3) (list c b a)) -> (3 2 1)

;; No match: list is longer than pattern
(when-match (@a @b) '(1 2 3) (list a b)) -> nil

;; Use of nil in dot position to match longer list
(when-match (@a @b . @nil) '(1 2 3) (list a b)) -> (1 2)
```

9.31.3 List match

Syntax:

```
(pattern+)
(pattern+ . pattern)
```

Description:

Pattern syntax consisting of a nonempty, possibly improper list matches list structure. A pattern expression may be specified in the dotted position. If it is omitted, then there is an implicit terminating `nil` which constitutes an atom expression matching `nil`.

A list pattern matches a list of the same shape. For each *pattern* expressions, there must exist an item in the list.

A match occurs when every *pattern* matches the corresponding element of the list, including the *pattern* in the dotted position.

Because the dotted position *pattern* matches a list, it is possible for a short pattern to match a longer list.

The syntax is indicated as requiring at least one *pattern* because otherwise the list is empty, which corresponds to the atom pattern `nil`.

The syntax `(. pattern)` is valid, but indistinguishable from *pattern* and therefore is not a list pattern.

Examples:

```
(if-match (@a @b @c . @d) '(1 2 3 . 4) (list d c b a))
--> (4 3 2 1)

;; 2 doesn't satisfy oddp
(if-match (@(oddp @a) @b @c . @d) '(2 x y z)
  (list a b c d)
  :no-match)
--> :no-match
```

```
;; 1 and 2 match, a takes (3 4)
(if-match (1 2 . @a) '(1 2 3 4) a) --> (3 4)

;; nesting
(if-match ((1 2 @a) @b) '((1 2 3) 4) (list a b)) -> (3 4)
```

9.31.4 Vector match

Syntax:

#(*pattern**)

Description:

A pattern match for a vector is expressed using vector notation enclosing pattern expressions. This pattern matches a vector object which contains exactly as many elements as there are patterns. Each pattern is applied against the corresponding vector element.

Examples:

```
;; empty vector pattern matches empty vector
(if-match #() #() :yes :no) -> :yes

;; empty vector pattern fails to match nonempty vector
(if-match #() #(1) :yes :no) -> :no

;; match with nested list and vector
(if-match #((1 @a) #(3 @b)) #((1 2) #(3 4)) (list a b))
--> (2 4)
```

9.31.5 Range match

Syntax:

#R(*from-pattern to-pattern*)

Description:

A pattern match for a range can be expressed by embedding pattern expressions in the `#R` notation. The resulting pattern requires the corresponding object to be a range, otherwise the match fails. If the corresponding object is a range, then the *from-pattern* is matched against its `from` and the *to-pattern* is matched against its `to` part.

Note that if the range expression notation `a..b` is used as a pattern, that is actually a list pattern, due to that being a syntactic sugar for `(rcons a b)`.

Examples:

```
(if-match #R(10 20) 10..20 :yes :no) -> :yes
(if-match #R(10 20) #R(10 20) :yes :no) -> :yes
(if-match #R(10 20) #R(1 2) :yes :no) -> :no

(when-match #R(@a @b) 1..2 (list a b)) -> (1 2)

;; not a range match! rcons syntax match
(when-match @a..@b '1..2 (list a b)) -> (1 2)

;; above, de-sugared:
```

```
(when-match (rcons @a @b) '(rcons 1 2) (list a b)) -> (1 2)
```

9.31.6 Quasiliteral match

Syntax:

```
`...@var...`
@`...@var...`
```

Description:

The quasiliteral syntax is supported as a pattern-matching operator. The corresponding object is required to be a character string, which is analyzed according to the structure of the quasiliteral pattern, and portions of the string are captured by variables. If the corresponding object isn't a string according to `stringp` then the match fails. The quasiliteral pattern must match the entire input string.

In order that the quasiliteral's syntactic structure is not misinterpreted as a predicate pattern, and in order to make certain situations work in quasiquoted pattern matching, a quasiliteral pattern may be specified as either `` `...` `` or `` @`...` ``. The latter form, which is structurally `(sys:expr (sys:quasi ...))` is specially recognized and treated as equivalent to the unadorned quasiliteral pattern.

A quasiliteral pattern matches in a linear fashion, from left to right. Variables bound earlier in the pattern can be referenced later in the pattern as bound variables.

With one exception, bound variables denote character strings in accordance with the usual quasiliteral conversion and formatting rules. All of the modifier notations may be used. For instance, if `x` is a bound variable, then `@{x -40}` denotes the value of `x` converted to a string, and right-aligned in a forty-character-wide field. Consequently, the notation matches exactly such a forty-character text. The exception is that if a bound variable has a regular expression modifier, as in `@{x #/re/}` then it has a special meaning as a pattern. Moreover, this syntax has no meaning in a quasiliteral.

In the following description of the quasiliteral pattern-matching rules, the symbols *uv*, *uv0* and *uv1* represent to unbound variables: variables which have no apparent lexical binding and are not defined as global variables. Unless indicated otherwise, `@`*uv* refers to a plain variable syntax such as `@abc` or else to braced syntax without modifiers, such as `@{abc}`. The same remarks apply to *uv0* and *uv1*. The symbol *bv* represents a bound variable: a variable which has an existing binding, which can occur in the form of the ordinary notation, or the braced notation with or without modifiers. The notation `{P}`, `{P0}`, `{P1}`... denotes a substring of the pattern, possibly empty.

`` `` `` The empty quasiliteral pattern matches an empty string.

`` `text{P}` ``
: A quasiliteral pattern which begins with a portion of text matches a string which begins with the same text. The remaining portion `{P}` of the pattern is then matched against a suffix of the input string which excludes the matched text.

`` `@uv` ``
: A simple unbound variable occurring as the last element of the pattern matches and binds the entire rest of the input string.

`` `@uvtext{P}` ``
: A simple unbound variable followed by a text element matches the input string if `"text"` occurs in that string as a substring. In that case, *uv* is bound to the possibly empty prefix of the input string consisting of the characters before the leftmost match for `"text"`. The rest of the pattern `{P}` is then matched against that suffix of the input

string which begins after the last character of the leftmost match for `"text"`.

`` `@uv@bv{P}` ``
: The bound variable *bv* is converted to text in the manner of an ordinary quasiliteral substitution. The situation then reduces to the `` `@uvtext{P}` `` pattern, where `text` denotes the character string produced by substitution of *bv*.

`` `@{uv integer}{P}` ``
: An unbound variable *uv* which uses the brace notation to specify a literal *integer* modifier denotes a match for that many characters. It is an error if the value is zero or negative. The match succeeds if the input string has at least that many characters, in which case the variable *uv* takes on those characters, and the rest of the pattern is matched against a suffix of the string with those characters removed.

`` `@{uv #/regex/}{P}` ``
: An unbound variable *uv* which carries a regular-expression modifier specifies a regular-expression match. If a prefix of the input string matches *regex*, then the match is successful and *uv* captures that prefix. The rest of the pattern `{P}` is then matched against the rest of the string after the prefix.

`` `@{bv #/regex/}{P}` ``
: A bound variable *bv* which carries a regular expression modifier specifies a regular expression match exactly like an unbound variable. This syntax produces a successful match if two conditions are met: a prefix of the input string matches *regex*, and the matched prefix is *equal* to the value of *bv*. The rest of the pattern `{P}` is then matched against the rest of the string after the prefix.

`` `@bv{P}` ``
: The bound variable *bv* is converted to text the manner of an ordinary quasiliteral substitution. The situation then reduces to the `` `text{P}` `` pattern, where `text` denotes the character string produced by substitution of *bv*.

`` `@uv0@uv1{P0}` ``
: Two consecutive unbound variables, where *uv0* is a plain variable with no modifiers, constitutes an invalid pattern. This situation is diagnosed as an error. If *uv0* is braced, carrying an integer or regular-expression modifier *mod*, then the situation is treated as the pattern `` `@{uv mod}{P}` `` where `{P}` refers to the `@uv1{P0}` portion.

No other quasiliteral syntax, or combination of variable modifiers, is supported in quasiliteral patterns.

Examples:

```
(when-match `@a-@b` "foo-bar" (list a b)) -> ("foo" "bar")

(when-match `@{a #/\d+/}@b` "123xy" (list a b)) -> ("123" "xy")

(let ((a 42))
  (when-match `[@{a -8}] @b` "[      42] packets` b))
-> "packets"
```

9.31.7 Quasiquote matching notation

Syntax:

```
^qq-syntax
```

Description:

Quasiquoting provides an alternative pattern-matching syntax. It uses a subset of the quasiquoting notation. Only specific kinds of quasiquoted objects listed in this description are supported. Within a quasiquote used for pattern-matching, unquotes indicate operators and variables instead of the `@` prefix. Splicing unquote syntax plays no role; its presence produces unspecified behavior.

The quasiquote matching notation is described, understood and implementing in terms of a translation to the standard pattern-matching syntax, according to the following rules. The `[X]` notation used here indicates that the element enclosed in brackets is subject to a recursive translation according to the rules:

`,`*`expr`* An unquoted expression occurring in the quasiquote is translated to the `@`*`expr`* pattern-matching syntax. If *`expr`* is an symbol, then this is a meta-variable: `(sys:var` *`expr`*`)` otherwise it is translated to the `(sys:expr` *`expr`*`)` syntax.

`,`` `...quasilit...` ``
An unquoted quasiliteral is treated uniformly as `,`*`expr`* and is therefore translated into `@`` `...quasilit...` ``. Since that is equivalent to `` `...quasilit...` ``, quasiliteral matching is supported within quasiquote notation in a straightforward way.

`~`*`expr`* In JSON syntax, unquotes are given the same above treatment as `,` (comma) unquotes in ordinary syntax.

`~`` `...quasilit...` ``
Similarly, quasiliterals are supported in JSON syntax.

`#H(()` *`(k0 v0) (k1 v1) ...`*`)`
Hash quasiliteral syntax is translated according to the `@(hash ([`*`k0`*`] [`*`v0`*`]) ([`*`k0`*`] [`*`v0`*`]) ...)` pattern, with each key and value recursively translated. The syntax must specify `()` for the hash construction arguments part, otherwise an error is diagnosed. That is to say, it must be of the form `#H(() ...)`. where the first element is `()`.

`#S(`*`type e0 e1 ...`*`)`
Structure quasiliteral syntax is translated according to the `@(struct [`*`type`*`] [`*`e0`*`] [`*`e1`*`] ...)` pattern.

`#(`*`e0 e1 ...`*`)`
Vector quasiliteral syntax is translated according to the `#([`*`e0`*`] [`*`e1`*`] ...)` pattern: it becomes a vector object containing embedded patterns.

`#J[`*`e0, e1, ...`*`]`
A JSON array quasiquote is translated into `#([`*`e0`*`] [`*`e1`*`] ...)` exactly like a vector. Here, the `[X]` transformation recognizes JSON `~` (tilde) unquotes, and recursively recognizes and transform JSON syntax not prefixed by `#J`.

`#J{`*`k0 : v0, k1 : v1, ...`*`}`
A JSON hash quasiquote is translated into `@(hash ([`*`k0`*`] [`*`v0`*`]) ([`*`k0`*`] [`*`v0`*`]) ...)` exactly like a hash.

`(`*`car . cdr`*`)`
Tree structure is translated according to the `([`*`car`*`] . [`*`cdr`*`])` pattern: it is recursively examined for translations.

`^`*`nested-qq-syntax`*
A nested quasiquote pattern is diagnosed as an error.

`,*`*`expr`*
Splicing syntax is diagnosed as an error.

~*expr
: Splicing JSON syntax is diagnosed as an error inside a JSON quasiliteral.

~*expr

obj Any other quasiquoted object is left untranslated.

Examples:

```
;; basic unquote: variables embedded via unquote,
;; not requiring @ prefix.
(when-match ^(,a ,b) '(1 2) (list a b))
--> (1 2)

;; operators embedded via unquote; interior of operators
;; is regular non-quasiquoting pattern syntax.
(when-match ^(,(oddp @a) ,(evenp @b)) '(1 2) (list a b))
--> (1 2)

(when-match ^#(,a ,b) #(1 2) (list a b))
--> (1 2)

(when-match ^#S(,type year ,y) #S(time year 2021)
  (list (struct-type-name type) y))
--> (time 2021)

(when-match ^#H(() (x ,y) (,(symbolp @y) ,datum))
            #H(() (x k) (k 42))
            datum)
--> (42)

;; JSON syntax

(when-match ^#J~a 42.0 a) --> 42.0

(when-match ^#J[~a, ~b] #J[true, false] (list a b)) --> (t nil)

(when-match ^#J{"x" : ~y, ~(symbolp @y) : ~datum}
            #J{"x" : true, true : 42}
            datum)
--> (42.0)

(when-match ^#J{"foo" : {"x" : ~val}}
            #J{"foo" : {"x" : "y"}} val)
--> "y"
```

9.31.8 Pattern operator `struct`

Syntax:

```
@(struct name {slot-name pattern}*)
@(struct pattern {slot-name pattern}*)
```

Description:

The `struct` pattern operator matches a structure object. The operator supports two modes of matching, the choice of which depends on whether the first argument is a *name* or a *pattern*.

The first argument is considered a *name* if it is a bindable symbol according to the `bindable` function. In this situation, the operator operates in strict mode. Otherwise, the operator is in loose mode.

The *name* or *pattern* argument is followed by zero or more *slot-name pattern* pairs, which are not enclosed in lists, similarly to the way slots are presented in the `#S` struct syntax and in the argument conventions of the `new` macro.

In strict mode, *name* is assumed to be the name of an existing struct type. The object being matched is tested whether it is a subtype of this type, as if using the `subtypep` function. If it isn't, the match fails.

In loose mode, the object being matched is tested whether it is a structure object of any structure type. If it isn't, the match fails.

In strict mode, each *slot-name pattern* pair requires that the object's slot of that name contain a value which matches *pattern*. The operator assumes that all the *slot-name*s are slots of the struct type indicated by *name*.

In loose mode, no assumption is made that the object actually has the slots specified by the *slot-name* arguments. The object's structure type is inquired to determine whether it has each of those slots. If it doesn't, the match fails. If the object has the required slots, then the values of those slots are matched against the patterns.

In loose mode, the *pattern* given in the first argument position of the syntax is matched against the object's structure type: the type itself, rather than its symbolic name.

Examples:

```
;; extract the month from a time structure
;; that is required to have a year of 2021.

(when-match @(struct time year 2021 month @m)
            #S(time year 2021 month 1)
  m) -> 1

;; match any structure with name and value slots,
;; whose name is foo, and extract the value.

(defstruct widget ()
  name
  value)

(defstruct grommet ()
  name
  value)

(append-each ((obj (list (new grommet name "foo" value :grom)
                         (new widget name "foo" value :widg))))
  (when-match @(struct @type name "foo" value @v) obj
    (list (list type v))))

--> ((#<struct-type grommet> :grom)
     (#<struct-type widget> :widg))
```

9.31.9 Pattern operator `hash`

Syntax:

```
@(hash {(key-pattern [value-pattern])}*)
```

Description:

The `hash` pattern operator matches a hash-table object by means of patterns which target keys, values or both.

An important concept in the requirements governing the operation of the `hash` operator is that of a trivial pattern.

A pattern is nontrivial if it is a variable or operator pattern. A pattern is also nontrivial if it is a list, vector or range pattern containing at least one nontrivial pattern. Otherwise, it is trivial.

The `hash` operator requires the corresponding object to be a hash table, otherwise the match fails.

If the corresponding object is a hash table, then matches each *`key-pattern`* and *`value-pattern`* pair against that object as described below. Each of the pairs must successfully match, otherwise the overall match fails.

The following requirements apply to key-value pattern pairs in which the value pattern is specified.

If *`key-pattern`* is a trivial pattern, then the semantics of the match is that *`key-pattern`* is taken as a literal object representing a hash key. The hash table is searched for that key. If the key is not found, the match fails. Otherwise, the value corresponding to that key is matched against the *`value-pattern`* which may be trivial or nontrivial.

If *`key-pattern`* is a simple variable pattern `@`*`sym`* and if *`sym`* has an existing binding, then the value of *`sym`* is looked up in the hash table. If it is not found, then the match fails, otherwise the corresponding value is matched against *`value-pattern`*, which may be trivial or nontrivial.

If *`key-pattern`* is a nontrivial pattern other than a variable pattern for a variable which has an existing binding, and if *`value-pattern`* is trivial, then *`value-pattern`* is taken as a literal object, which is used for searching the hash table for one or more keys, as if it were the *`value`* argument in a call to the `hash-keys-of` function, to find all keys which have a value `equal` to that value. If no keys are found, then the match fails. Otherwise, the `key-pattern` is then matched against the retrieved list of hash keys.

Finally, if both *`key-pattern`* and *`value-pattern`* are nontrivial, then an exhaustive search is performed of the hash table. Every key in the hash table is matched against *`key-pattern`* and if it matches, the value is matched against *`value-pattern`*. If both match, then the values from the matches are collected into lists. At least one matching key-value pair must be found, otherwise the overall match fails. Note: this situation can be understood as if the hash table were an association list of `cons` cells of the form

```
(key . value)
```

and as if the two patterns were combined into a `coll` operator against this list in the following way:

```
@(coll (key-pattern . value-pattern))
```

such that the semantics can then be understood in terms of the `coll` operator matching against an association list.

The following requirements apply when the *value-pattern* is omitted.

If *key-pattern* is a nontrivial pattern other than a variable pattern for a variable which has an existing binding, then the pattern is applied against the list of keys from the hash table, which are retrieved as if using the `hash-keys` function.

If *key-pattern* is a variable pattern referring to an existing binding, then that pattern is taken as a literal object. The match is successful if that object occurs as a key in the hash table.

Example:

```
;; First, (x @y) has a trivial key pattern so the x
;; entry from the hash table is retrieved, the
;; value being the symbol k. This k is bound to @y.
;; Because y now a bound variable the pattern (@y @datum)
;; is interpreted as search of the hash table for
;; a single entry matching the value of @y. This
;; is the k entry, whose value is 42. The @datum
;; value match takes this 42.
(when-match @(hash (x @y) (@y @datum))
            #H(() (x k) (k 42)) datum)
--> 42

;; Again, (x @y) has a trivial key pattern so the x
;; entry from the hash table is retrieved, the
;; value being the symbol k. This k is bound to @y.
;; This time the second pattern has a @(symbolp)
;; predicate operator. This is not a variable, and
;; so the pattern searches the entire
;; hash table. The @y variable has a binding to k,
;; so only the (k 42) entry is matched. The 42
;; value matches @datum, and is collected into a list.
(when-match @(hash (x @y) (@(symbolp @y) @datum))
            #H(() (x k) (k 42)) datum)
--> (42)
```

9.31.10 Pattern operator `as`

Syntax:

```
@(as name pattern)
```

Description:

The `as` pattern operator binds the corresponding object to a fresh variable given by *name*, similarly to the Lisp `let` operator. If another variable called *name* exists, it is shadowed; thus, no back-referencing is performed.

The *name* argument must be a bindable symbol, or else `nil`. If *name* is `nil`, then no name is bound. Thus `@(as nil pattern)` is equivalent to *pattern*. Otherwise, *pattern* processed in a scope in which the new *name* binding is already visible.

The `as` operator succeeds if *pattern* matches.

Note: in a situation when it is necessary to bind a variable to an object in parallel with one or more patterns, such that the variable can back-reference to an existing occurrence, the `and` pattern

operator can be used.

Example:

```
;; w captures the entire (1 2 3) list:

(when-match @(as w (@a @b @c)) '(1 2 3) (list w a b c))
--> ((1 2 3) 1 2 3)

;; match a list which has itself as the third element
(when-match @(as a (1 2 @a 4)) '#1=(1 2 #1# 4) :yes)
--> :yes
```

9.31.11 Pattern operator `with`

Syntax:

```
@(with [main-pattern] {side-pattern | name} expr)
```

Description:

The `with` pattern operator matches the optional *`main-pattern`* against a corresponding object, while matching a *`side-pattern`* or *`name`* against the value of the expression *`expr`* which is embedded in the syntax.

First, if *`main-pattern`* is present in the syntax, it is matched against its corresponding object. This match must succeed, or else the `with` operator fails to match, in which case *`expr`* is not evaluated.

Next, if *`main-pattern`* successfully matched, or is absent, *`expr`* is evaluated in the scope of earlier pattern variables, including any which that emanate from *`main-pattern`*. It is unspecified whether later pattern variables are visible.

Finally, *`side-pattern`* is matched against the value of *`expr`*. If that succeeds, then the operator has successfully matched.

If a *`name`* is specified instead of a *`side-pattern`*, it must be a bindable symbol or else *`nil`*.

Examples:

```
(when-match (@(with @a x 42) @b @c) '(1 2 3) (list a b c x))
--> (1 2 3 42)

(let ((o 3))
  (when-match (@(evenp @x) @(with @z @(oddp y) o)) '(4 6)
    (list x y z)))
--> (4 3 6)
```

9.31.12 Pattern operator `require`

Syntax:

```
@(require pattern condition*)
```

Description:

The pattern operator `require` applies the specified *`pattern`* to the corresponding object. If the *`pattern`* matches, the operator then imposes the additional constraints specified by zero or more

condition forms. Each *condition* is evaluated in a scope in which the variables from *pattern* have already been established.

For the `require` operator to be a successful match, every *condition* must evaluate true, otherwise the match fails.

The *condition* forms behave as if they were the arguments of an implicit `and` operator, which implies left-to-right evaluation behavior, stopping evaluation on the first *condition* which produces `nil`, and defaulting to a result of `t` when no *condition* forms are specified.

Examples:

```
;; Match a (+ a b) expression where a and b are similar:

(when-match @(require (+ @a @b) (equal a b)) '(+ z z) (list a b))
--> (z z)

;; Mismatched case
(if-match @(require (+ @a @b) (equal a b)) '(+ y z)
   (list a b)
   :no-match)
--> :no-match
```

9.31.13 Pattern operators `all` and `all*`

Syntax:

```
@(all pattern)
@(all* pattern)
```

Description:

The `all` and `all*` pattern operators require the corresponding object to be a sequence.

The specified *pattern* is applied against every element of the sequence. The match is successful if *pattern* matches every element.

Furthermore, in the case of a successful match, each variable that is freshly bound by *pattern* is converted into a list of all of the objects which that variable encounters from all elements of the sequence. Those variables which already have a binding from another *pattern* are not converted to lists. Their existing values are merely required to match each corresponding object they encounter.

The difference between `all` and `all*` is as follows. The `all` operator respects the vacuous truth of the match when the sequence is empty. In that case, the match is successful, and the variables are all bound to the empty list `nil`. In contrast, the alternative `all*` operator behaves like a failed match when the sequence is empty.

Examples:

```
;; all elements of list match the pattern (x @a @b)
;; a is bound to (1 2 3); b to (a b c)

(when-match @(all (x @a @b))
           '((x 1 a) (x 2 b) (x 3 c))
  (list a b))
```

```
--> ((1 2 3) (a b c))

;; Match a two element list whose second element
;; consists of nothing but zero or more repetitions
;; of the first element. x is not turned into a list
;; because it has a binding due to @x.
(when-match @(@x @(all x)) '(1 (1 1 1 1)) x) -> 1

;; no match because of the 2
(when-match @(@x @(all x)) '(1 (1 1 1 2)) x) -> nil
```

9.31.14 Pattern operator `some`

Syntax:

```
@(some pattern)
```

Description:

The `some` pattern operator requires the corresponding object to be a sequence. The specified *pattern* is applied against every element of the sequence. The match is successful if *pattern* matches at least one element.

Variables are extracted from the first matching which is found.

Example:

```
;; the second (x 2 b) element is the leftmost one
;; which matches the (x @a @b) pattern

(when-match @(some (x @a @b))
           '((y 1 a) (x 2 b) (z 3 c))
  (list a b))
-> (2 b)
```

9.31.15 Pattern operator `coll`

Syntax:

```
@(coll pattern)
```

Description:

The `coll` pattern operator requires the corresponding object to be a sequence. The specified *pattern* is applied against every element of the sequence. The match is successful if *pattern* matches at least one element.

Each variable that is freshly bound by the *pattern* is converted into a list of all of the objects which that variable encounters from the matching elements of the sequence. Those variables which already have a binding from another *pattern* are not converted to lists. Their existing values are merely required to match each corresponding object they encounter.

Variables are extracted from all matching elements, and collected into parallel lists, just like with the `@(all)` operator.

Example:

```
(when-match @(coll (x @a @b))
```

```
            '((y 1 a) (x 2 b) (z 3 c) (x 4 d))
  (list a b))
-> ((2 4) (b d))
```

9.31.16 Pattern operator `scan`

Syntax:

```
@(scan pattern)
```

Description:

The `scan` operator matches *`pattern`* against the corresponding object. If the match fails, and the object is a `cons` cell, the match is tried on the `cdr` of the cons cell. The `cdr` traversal repeats until a successful match is found, or a match failure occurs against against an atom.

Thus, a list object, possibly improper, matches *`pattern`* under `scan` if any suffix of that object matches.

Examples:

```
;; mismatch: 1 doesn't match 2
(when-match @(scan 2) 1 t) -> t

;; simple atom match: 42 matches 42
(when-match @(scan 42) 42 t) -> t

;; (2 3) is a sublist of (1 2 3 4)
(when-match @(scan (2 3 . @nil)) '(1 2 3 4) t) -> t

;; (2 @x 4 . @nil) matches (2 3 4), binding x to 3:
(when-match @(scan (2 @x 4 . @nil)) '(1 2 3 4 5) x) -> 3

;; The entire matching suffix can be captured.
(when-match @(scan @(as sfx (2 @x 4 . @nil)))
             '(1 2 3 4 5)
  sfx)
-> (2 3 4 5)

;; Missing . @nil in pattern anchors search to end:
(when-match @(scan (@x 2))
             '(1 2 3 2 4 2)
  x)

;; Terminating atom anchors to improper end:
(when-match @(scan (@x . 4))
             '(1 2 3 . 4)
  x)
-> 3

;; Atom pattern matches only terminating atom
(when-match @(scan #(@x @y))
             '(1 2 3 . #(4 5))
  (list x y))
-> (4 5)
```

9.31.17 Pattern operators `and` and `or`

Syntax:

```
@(and pattern*)
@(or pattern*)
```

Description:

The `and` and `or` operators match multiple patterns in parallel, against the same object. The `and` operator matches if every *pattern* matches the object, otherwise there is no match. The `or` operator requires one *pattern* to match. It tries the patterns in left-to-right order, and stops at the first matching one, declaring failure if none match.

The `and` and `or` operators have different scoping rules. Under `and`, later patterns are processed in the scopes of earlier patterns, just like with other pattern operators. Duplicate variables back-reference. Under `or`, the patterns are processed in separate, parallel scopes. No back-referencing takes place among same-named variables introduced in separate patterns of the same `or`.

When the `and` matches, the variables from all of the patterns are bound. When the `or` operator matches, the variables from all of the patterns are also bound. However, only the variables from the matching *pattern* take on the values implied by that pattern. The variables from the non-matching patterns that do not have the same names as variables in the matching *pattern*, and that have been newly introduced in the `or` operator, take on `nil` values.

Examples

```
(if-match @(and (@x 2 3) (1 @y 3) (1 2 @z)) '(1 2 3)
  (list x y z)) -> (1 2 3)

(if-match @(or (@x 3 3) (1 @x 3) (1 2 @x)) '(1 2 3)
  x) -> 2
```

9.31.18 Pattern operator `not`

Syntax:

```
@(not pattern)
```

Description:

The pattern operator `not` provides logical inverse semantics. It matches if and only if the *pattern* does not match.

Whether or not the `not` operator matches, no variables are bound. If the embedded *pattern* matches, the variables which it binds are suppressed by the `not` operator.

Examples:

```
;; @a matches unconditionally, so @(not @a) always fails:
(if-match @(not @a) 1 :yes :no) -> :no

;; error: a is not bound
(if-match @(not @a) 1 :yes a) -> error

(match-case '(1 2 3)
  ((@(not 1) @b @c) (list :case1 b c))
  ((@(not 0) @b @c) (list :case2 c b)))
```

```
--> (:case2 3 2)
```

9.31.19 Pattern predicate operator

Syntax:

```
@(function arg*)
@(function arg* @avar arg*)
@(function arg* . @avar)
@(@rvar (function arg*))
@(@rvar (function arg* @avar arg*))
@(@rvar (function arg* . @avar))
```

Description:

Whenever the operator position of a pattern consists of a symbol which is neither the name of a pattern operator, nor the name of a macro, the expression denotes a predicate pattern. An expression is also a predicate pattern if it is handled by a pattern macro which declines to expand it by yielding the original expression.

An operator pattern is expected to conform to one of the first three syntactic variations above. Together, these three variations constitute the *first form* of the pattern predicate operator. Whenever the operator position of a pattern consists of a meta-symbol, it is also a predicate pattern, expected to conform to one of the second three syntax variations above. These three variations constitute the *second form* of the operator.

The first form of the predicate pattern consists of a compound form consisting of an operator and arguments. Exactly one of the arguments may be a pattern variable *avar* ("argument variable") which must be a bindable symbol or else `nil`. The pattern variable may also appear in the dot position, rather than as an argument. The role of *avar* and the consequences of omitting it are described below.

The second form of the predicate pattern consists of a meta-symbol *rvar* ("result variable") which must be a bindable symbol or else `nil`. This is followed by a compound form which consists of an operator symbol, followed by arguments, one of which may be a pattern `avar` as in the simple form. If *rvar* is `nil`, then the predicate pattern is equivalent to the first form. That is to say, the following are equivalent:

```
@(@nil (f ...))  <-->  @(f ...)
```

The matching of the predicate pattern is processed as follows. If the *avar* variable is present, then the predicate pattern first binds the corresponding object to the *avar* variable, performing an ordinary variable match with the potential back-referencing which that implies. If that succeeds, then the object is inserted into the compound form, substituted in the position indicated by the `@`*avar* variable, either an ordinary argument position or the dot position. This form is then evaluated. If it yields true, then the match is successful, otherwise the match fails.

If the *avar* variable is absent, then no initial variable matching takes place. The corresponding object is added as an extra rightmost argument into the compound form, which is evaluated. Its truth value then determines the success of the match, just like in the case with *avar*.

If the second form is being processed, and specifies a *rvar* that is not `nil`, and if the predicate has succeeded, then then an extra processing step takes place. A variable match is performed to bind the *rvar* variable to the result of the predicate, with potential back-referencing. If that match succeeds, then the predicate pattern succeeds.

The compound form may be headed by the `dwim` operator, and therefore the DWIM bracket notation may be used. For instance `@[f @x]` is equivalent to `@(dwim f @x)` and is processed accordingly. Similarly, `@(@y [f @x])` is equivalent to `@(@y (dwim f @x))`.

The dot position of *avar* in the predicate syntax denotes function application. So that is to say, the pattern predicate form `(f . @a)` where `@a` is in the dotted position invokes the function `f` as if by evaluation of the form `(f . x)` where `x` is hidden temporary variable holding the object corresponding to the pattern. The form `(f . x)` is a standard **TXR Lisp** notation with the same meaning as `(apply (fun f) x)`.

If *avar* is the `nil` symbol, then no variable is bound. The matched object is substituted into the predicate expression at the position indicated by `@nil`.

Examples:

```
(when-match (@(evenp) @(oddp @x)) '(2 3) x) -> 3

(when-match @(<= 1 @x 10) 4 x) -> 4

(when-match @(@d (chr-digit @c)) #\5 (list d c)) -> (5 #\5)

(when-match @(<= 1 @x 10) 11 x) -> nil

;; use hash table as predicate:
(let ((h #H(() (a 1) (b 2))))
  (when-match @[h @x] 'a x))
-> a

;; as above, also capture hash value
(let ((h #H(() (a 1) (b 2))))
  (when-match @(@y [h @x]) 'a (list x y)))
-> (a 1)

;; apply (1 2 3) to < using dot position
(when-match @(@x (< . @sym)) '(1 2 3) (list x sym))
-> (t (1 2 3))

;; Match three-element list whose middle element
;; is a number in the range 10 20, without
;; binding any variables:
(when-match (@nil @(<= 10 @nil 20) @nil) obj
  (prinl "obj matches"))
```

9.31.20 Pattern macro `sme`

Syntax:

```
@(sme spat mpat epat [mvar [evar]])
```

Description:

The pattern macro `sme` (start, middle, end) is a notation defined using the `defmatch` macro.

The `sme` macro generates a complex pattern which matches three non-overlapping parts of a list object using three patterns. The *spat* pattern is required to match a prefix of the input list. If that match is successful, then the remainder of the list is searched for a match for *mpat*, using the

`scan` operator. If that match, in turn, is successful, then the suffix of the remainder of the list is required to match `epat`.

The optional *mvar* and *evar* arguments must be bindable symbols, if they are specified. These symbols specify lexical variables which are bound to, respectively, the object matched by *mpat* and *epat*, using the fresh binding semantics of the `as` pattern operator.

The first two patterns, *spat* and *mpat*, must be possibly dotted list patterns. The last pattern, *epat*, must be either an atom or a possibly dotted list pattern.

Important to the semantics of `sme` is the concept of the length of a list pattern.

The length of a pattern with a pattern variable or operator in the dotted position is the number of items before that variable or operator. The length of `(1 2 . @(and a b))` is 2; likewise the length of `(1 2 . @nil)` is also 2. The length of a pattern which does not have a variable or operator in the dotted position is simply its list length. For instance, the pattern `(1 2 3)` has length 3, and so does the pattern `(1 2 3 . 4)`. The length is determined by the list object structure of the pattern, and not the printed syntax used to express it. Thus, `(1 . (2 3))` is still a length 3 pattern, because it denotes the same `(1 2 3)` object, using the dot notation unnecessarily.

The non-overlapping semantics of `sme` evolves as follows. In the following description, it is understood that a match is required at every step. If that match fails, then the entire `sme` operator fails:

1. First, *spat* is required to match a a prefix of the input list. If the match succeeds, then a *middle suffix* of the input is calculated by dropping from it leading elements. The number of elements dropped is equal to the length of *spat*.
2. The middle suffix is then searched for an occurrence of the middle pattern *mpat*, as if using the `scan` pattern operator. All elements skipped by the search are dropped, until a match is found.
3. At that point, if *mvar* has been specified, it is bound to the remaining input, which still includes the part which just matched *mpat*.
4. Next, a number of elements equal to the length of *mpat*, are dropped from the middle suffix, leaving a residue comprising the *final suffix*.
5. The end pattern *epat* must then match a suffix of the final suffix.
6. If the *evar* variable has been specified, it is bound to the entire suffix that was matched by *epat*.

Examples:

```
(when-match @(sme (1 2) (3 4) (5 . 6) m e)
            '(1 2 3 4 5 . 6)
  (list m e))
-> ((3 4 5 . 6) (5 . 6))

(when-match @(sme (1 2) (3 4) (5 . 6) m e)
            '(1 2 abc 3 4 def 5 . 6)
  (list m e))
((3 4 def 5 . 6) (5 . 6))

;; backreferencing
```

```
(when-match @(sme (1 @y) (@z @x @y @z) (@x @y)) '(1 2 3 1 2 3 1 2)
  (list x y z))
-> (1 2 3))

;; collect odd items starting at 3, before 7
(when-match @(and @(sme (1 @x) (3) (7) m e)
                  @(with @(coll @(oddp @y)) (ldiff m e)))
            '(1 2 3 4 5 6 7)
  (list x y))
-> (2 (3 5)))

;; no overlap
(when-match @(sme (1 2) (2 3) (3 4)) '(1 2 3 4) t) -> nil

;; The atom 5 is like a "zero-length improper list".
(when-match @(sme () () 5) 5 t) -> t
```

9.31.21 Pattern macro `end`

Syntax:

```
@(end pattern [var])
```

Description:

The pattern macro `end` is a notation defined using the `defmatch` macro, which matches *pattern* against the suffix of a corresponding list object, which may be an improper list or atom.

The optional argument *var* specifies the name of a variable which captures the matched portion of the object.

The `end` macro is related to the `sme` macro according to the following equivalence:

```
@(end pat var)  <-->  @(sme () () pat : var)
```

All of the requirements given for `sme` apply accordingly.

Examples:

```
;; atom match
(when-match @(end 3 x) 3 x) -> 3

;; y captures (2 3)
(when-match @(end (2 @x) y)
           '(1 2 3)
  (list x y))
-> (3 (2 3))

;; variable in dot position
(when-match @(end (2 . @x) y)
            '(1 2 . 3)
  (list x y))
-> (3 (2 . 3))

;; z captures entire object
(when-match @(as z @(end (2 @x) y))
```

```
            '(1 2 3)
  (list x y z))
-> (3 (2 3) (1 2 3)))
```

9.32 Pattern-Matching Macros

9.32.1 Macros `when-match`, `match` and `if-match`

Syntax:

```
(when-match pattern expr form*)
(match pattern expr form*)
(if-match pattern expr then-form [else-form])
```

Description:

The `when-match`, `match` and `if-match` macros conditionally evaluate code based on whether the value of *expr* matches *pattern*.

The `when-match` macro arranges for every *form* to be evaluated in the scope of the variables established by *pattern* when it matches the object produced by *expr*. The value of the last *form* is returned, or else `nil` if there are no forms. If the match fails, the forms are not evaluated, and `nil` is produced.

The `match` macro behaves exactly like `when-match` when the match is successful. When the match fails, `match` throws an exception of type `match-error`.

The `if-match` macro evaluates *then-form* in the scope of the variables established by *pattern* if the match is successful, and yields the value of that form. Otherwise, it evaluates *else-form*, which defaults to `nil` if it is not specified.

9.32.2 Macros `match-case` and `match-ecase`

Syntax:

```
(match-case expr {(pattern form*)}*)
(match-ecase expr {(pattern form*)}*)
```

Description:

The `match-case` macro successively matches the value of *expr* against zero or more patterns.

The syntax of `match-case` consists of an expression *expr* followed by zero or more clauses. Each clause is a compound expression whose first element is *pattern*, which is followed by zero or more forms.

First, *expr* is evaluated. Then, the value is matched against each *pattern* in succession, stopping at the first pattern which provides a successful match. If no pattern provides a successful match, then `match-case` terminates and returns `nil`.

If a *pattern* matches successfully, then each *form* associated with the pattern is evaluated in the scope of the variable bindings established by that *pattern*. Then `match-case` terminates, returning the value of the last *form* or else `nil` if there are no forms.

The `match-ecase` macro differs from `match-case` as follows. When none of the clauses match under `match-case`, then that form terminates with a value of `nil`. In the same situation, the `match-ecase` form throws an exception of type `match-error`.

Examples:

```
;; classify sequence of objects by pattern matching,
;; returning a list of the results

(collect-each ((obj (list '(1 2 3)
                          '(4 5)
                          '(3 5)
                          #S(time year 2021 month 1 day 1)
                          #(vec tor))))
  (match-case obj
    (@(struct time year @y) y)
    (#(@x @y) (list x y))
    ((@nil @nil @x) x)
    ((4 @x) x)
    ((@x 5) x)))

 --> (3 5 3 2021 (vec tor))

 ;; default case can be represented by a guaranteed match

 (match-case 1
   (2 :two)
   (@x :default)) --> :default
```

9.32.3 Macro `lambda-match`

Syntax:

```
(lambda-match {(pattern form*)}*)
```

Description:

The `lambda-match` is conceptually similar to `match-case`.

The arguments of `lambda-match` are zero or more clauses similar to those of `match-case`, each consisting of a compound expression headed by a *`pattern`* followed by zero or more *`form`*s.

The macro generates a `lambda` expression which evaluates to an anonymous function in the usual way.

When the anonymous function is called, each clause's *`pattern`* is matched against the function's actual arguments. When a match occurs, each *`form`* associated with the *`pattern`* is evaluated, and the value of the last *`form`* becomes the return value of the function. If none of the clauses match, then `nil` is returned.

Whenever *`pattern`* is a list-like pattern, it is not matched against a list object, as is the usual case with a list-like pattern, but against the actual arguments. For instance, the pattern `(@a @b @c)` expects that the function was called with exactly three arguments. If that is the case, the patterns are then matched to the arguments. The pattern `@a` takes the first argument, binding it to variable `a` and so forth.

If *`pattern`* is a dotted list-like pattern, then the dot position is matched against the remaining arguments. For instance, the pattern `(@a @b . @c)` requires at least two arguments. The first two are bound to `a` and `b`, respectively. The list of remaining arguments, if any, is bound to `c`, which will be `nil` if there are no remaining arguments.

Any non-list-like *pattern* `P` is analyzed as an equivalent list-like dotted pattern due to `P` syntax being equivalent to `(. P)` syntax. Such a pattern matches the list of all arguments. Thus, the following are all equivalent:

```
(lambda-match (@a a))
(lambda-match ((. @a) a))
(lambda a a)
(lambda (. a) a)
```

The characteristics of the resulting anonymous function are determined as follows.

If at least one *pattern* specified in a *lambda-match* is a dotted pattern, the function is variadic.

The arity of the resulting anonymous function is determined as follows, from the lengths of the patterns. The length of a pattern is the number of elements, not including the dotted element.

The length of the longest pattern determines the number of fixed arguments. Unless the function is variadic, it may not be called with more arguments than can be matched by the longest pattern.

The length of the shortest pattern determines the number of required arguments. The function may not be called with fewer arguments than can be matched by the shortest pattern.

If these two lengths are unequal, then the function has a number of optional arguments, equal to the difference.

Note: an anonymous function which takes one argument and matches that object against clauses using `match-case` can be obtained with the `do` operator, using the pattern: `(do match @1 ...)`.

Note: the parameter macro `:match` can also define a `lambda` with pattern matching. Any `(lambda-match clauses ...)` form can be written as `(lambda (:match) clauses ...)`. The parameter macro offers the additional ability of defining named arguments which are inserted before the implicit arguments generated from the clauses, and combining with other parameter macros.

Examples:

```
(let ((f (lambda-match
           (() (list 0 :args))
           ((@a) (list 1 :arg a))
           ((@a @b) (list 2 :args a b))
           ((@a @b . @c) (list* '> 2 :args a b c)))))
  (list [f] [f 1] [f 1 2] [f 1 2 3]))
-->
((0 :args) (1 :arg 1) (2 :args 1 2) (> 2 :args 1 2 3))

[(lambda-match
   ((0 1) :zero-one)
   ((1 0) :one-zero)
   ((@x @y) :no-match)) 1 0] --> :one-zero

[(lambda-match
   ((0 1) :zero-one)
   ((1 0) :one-zero)
```

```
   ((@x @y) :no-match)) 1 1] --> :no-match

[(lambda-match
   ((0 1) :zero-one)
   ((1 0) :one-zero)
   ((@x @y) :no-match)) 1 2 3] --> ;; error
```

9.32.4 Macro `defun-match`

Syntax:

```
(defun-match name {(pattern form*)}*)
```

Description:

The `defun-match` macro can be used to define a top-level function in the style of `lambda-match`.

It produces a form which has all of the properties of `defun`, such as a block of the same *name* being established around the implicit `match-case` so that `return-from` is possible.

The `(pattern form*)` clauses of `defun-match` have exactly the same syntax and semantics as those of `lambda-match`.

Note: instead of `defun-match`, the parameter macro `:match` may be used. The following equivalence holds:

```
(defun name (:match) ...)  <-->  (defun-match ...)
```

The parameter macro offers the additional ability of defining named arguments which are inserted before the implicit arguments generated from the clauses, and combining with other parameter macros.

Examples:

```
;; Fibonacci
(defun-match fib
  ((0) 1)
  ((1) 1)
  ((@x) (+ (fib (pred x)) (fib (ppred x)))))

(fib 0) -> 1
(fib 1) -> 1
(fib 2) -> 2
(fib 3) -> 3
(fib 4) -> 5
(fib 5) -> 8

;; Ackermann
(defun-match ack
  ((0 @n) (+ n 1))
  ((@m 0) (ack (- m 1) 1))
  ((@m @n) (ack (- m 1) (ack m (- n 1)))))

(ack 3 7) -> 1021
(ack 1 1) -> 3
```

```
(ack 2 2) -> 7
```

9.32.5 Parameter list macro `:match`

Syntax:

```
(:match left-param* [-- extra-param*]) clause*
```

Description:

Parameter list macro `:match` allows any function to be expressed in the style of `lambda-match`, with extra features.

The `:match` macro expects the body of the function to consist of `lambda-match` clauses, which are semantically treated in exactly the same manner as under `lambda-match`.

The following restrictions apply. The parameter list may not include optional parameters delimited by `:` (the colon keyword symbol). The parameter list may not be dotted.

The macro produces a function which the *`left-param`* parameters, if any, are inserted to the left of the implicit parameters generated by the `lambda-match` transformation.

Furthermore, the `:match` parameter macro supports integration with the `:key` parameter macro, or any other macro which uses a compatible `--` convention for delimiting special arguments. If the parameter list includes the symbol `--` then that portion of the parameter list is set aside and not included in the `lambda-match` transformation. Then, that list is integrated into the resulting lambda.

A complete transformation can be described by the following diagram:

```
(lambda (:match a b c ... -- s t u ...) clauses ...)

-->

(lambda (a b c ... m n p ... -- s t u ... . z) body ...)
```

In this diagram, `a b c ...` denote the *`left-param`* parameters. The `m n p ...` symbols denote the fixed parameters generated by the `lambda-match` transformation from the semantic analysis of *`clauses`*. The `s t u ...` symbols denote the original *`extra-param`* parameters. Finally, `z` denotes the dotted parameter generated by the `lambda-match` transform. If the transform produces no dotted parameter, then this is `nil`. The dotted parameter is thus separated from the `m n p ...` group to which it belongs.

When no `--` and *`extra-params`* are present, the transformation reduces to:

```
(lambda (:match a b c ...) clauses ...)

-->

(lambda (a b c ... m n p ... . z) body ...)
```

Note: these requirements harmonize with the `:key` parameter macro. If that is present to the left of `:match` it removes the `--` and the `s t u ...` keyword parameters, reuniting the `z` parameter with the `m n p` group. Furthermore, the `:key` macro generates code which refers to the existing `z` dotted parameter as the source for the keyword parameters, unless `z` is `nil`, in which case it inserts its own generated symbol.

Examples:

```
;; Match-style cond-like macro with unreachability diagnosis.
;; Demonstrates usefulness of :match, which allows the :form
;; parameter to be promoted through to the macro definition.

(defmacro my-cond (:match :form f)
  (() nil)
  ((((@(and @(constantp @test) @(eval))) . @rest)
    (when rest
      (compile-error f "unreachable code after ~s" test))
    test)
  ((((@(and @(constantp @test) @(eval)) . @forms) . @rest)
    (when (and rest)
      (compile-error f "unreachable code after ~s" test))
    ^(progn ,*forms))
  (((@test) . @rest)
    ^(or ,test (my-cond ,*rest)))
  (((@test . @forms) . @rest)
    ^(if ,test (progn ,*forms)
       (my-cond ,*rest)))
  ((@else . @rest) (compile-error f "bad syntax")))

(my-cond (3)) --> 3
(my-cond (3 4)) --> 4
(my-cond (3 4) (5)) --> ;; my-cond: unreachable code after 3
(my-cond 42) --> ;; my-cond: bad syntax

;; Keyword parameter example.

(defstruct simple-widget ()
  name)

(defstruct widget (simple-widget)
  frobosity
  luminance)

(defstruct simple-point-widget (simple-widget)
  (:static width 0)
  (:static height 0))

(defstruct point-widget (widget)
  (:static width 0)
  (:static height 0))

(defstruct general-widget (widget)
  width
  height)

;; Note that in clauses with no . @rest parameter, there
;; is a mismatch if keyword arguments are present. The (0 0)
;; clause exploits this to match only when keywords are absent.

(defun make-widget (:key :match name -- frob lum)
  ((0 0) (new simple-point-widget name name))
```

```
  ((0 0 . @rest) (new point-widget name name
                                    frobosity frob
                                    luminance lum))
  ((@x @y . @rest) (new general-widget name name
                                       width x
                                       height x
                                       frobosity frob
                                       luminance lum)))

(make-widget "abc" 0 0) --> #S(simple-point-widget name "abc")

(make-widget "abc" 0 0 :frob 42)
--> #S(point-widget name "abc" frobosity 42 luminance nil)

(make-widget "abc" 0 0 :lum 9)
--> #S(point-widget name "abc" frobosity nil luminance 9)

(make-widget "abc" 0 1 :lum 9)
--> #S(general-widget name "abc" frobosity nil luminance 9
                      width 0 height 0)
```

9.32.6 Macro `defmatch`

Syntax:

```
(defmatch name macro-style-params
   body-form*)
```

Description:

The `defmatch` macro allows for the definition of pattern macros: user-defined pattern operators which are implemented via expansion into existing operator syntax.

The `defmatch` macro has the same syntax as `defmacro`. It specifies a macro transformation for a compound form which has the *`name`* symbol in its leftmost position.

This macro transformation is performed when *`name`* is used as a pattern operator: an expression of the form `@(`*`name argument*`*`)` occurring in pattern-matching syntax.

The behavior is unspecified if *`name`* is the name a built-in pattern operator, or a predefined pattern macro.

The pattern macro bindings are stored in a hash table held by the variable `*match-macro*` whose keys are symbols, and whose values are expander functions. There are no lexically scoped pattern macros.

Pattern macros defined with `defmatch` may specify the special macro parameters `:form` and `:env` in their parameter lists. The values of these parameters are determined in a manner particular to `defmatch`.

The `:form` parameter captures the pattern-matching form, or a constituent thereof, in which the macro is being invoked. For instance, if the operator is being used inside a pattern given to a `when-match` macro invocation, then the form will be that entire `when-match` form.

The `:env` parameter captures a specially constructed macro-time environment object in which all of the variables to the left of the pattern appear as lexical variables. The parent of this environment

is the surrounding macro environment. If the pattern macro needs to treat a variable which already has a binding differently from an unbound variable, it can look up the variable in this environment.

Example:

```
;; Create an alias called let for the @(as var pattern) operator:
;; Note that the macro produces @(as ...) and not just (as ...)

(defmatch let (var pattern)
  ^@(as ,var ,pattern))

;; use the macro in matching:
(when-match @(let x @(or foo bar)) 'foo x)

;; Error reporting example using :form

(defmatch foo (sym)
  (unless (bindable sym)
    (compile-error *match-form*
                   "~s: bindable symbol expected, not ~s"
                   'foo sym))
  ...)

;; Pattern macro which uses = equality to backreference
;; an existing lexical binding, or else binds the variable
;; if it has no existing lexical binding.
(defmatch var= (sym :env e)
  (if (lexical-var-p e sym)
    (with-gensyms (obj)
      ^@(require (sys:var ,obj)
          (= ,sym ,obj)))
    ^(sys:var ,sym)))

;; example use:
(when-match (@(var= a) @(var= a)) '(1 1.0) a)
-> 1

;; no match: (equal 1 1.0) is false
(when-match (@a @a) '(1 1.0) a)
-> nil
```

9.32.7 Function `macroexpand-match`

Syntax:

(macroexpand-match *pattern* [*env*])

Description:

If `pattern` is a compound form whose operator symbol has been defined as a macro pattern using `defmatch`, then `macroexpand-match` will expand that pattern and return the expansion. Otherwise it returns the `pattern` argument.

In order to be recognized by `macroexpand-match` the *pattern* argument must not include the `@` prefix that would normally be used to invoke it. The expansion, however, will include that syntax.

The `env` parameter specifies the macro-time environment for the expander. Note: pattern expanders, like built-in patterns, may use the macro environment for deciding whether a variable is an existing lexical variable, or a free variable, based on which a pattern may be expanded differently.

Example:

Given:

```
(defmatch point (x y)
  ^@(struct point x @,x y @,y))
```

a result similar to the following may be obtained:

```
(macroexpand-match '(point a b)) -> @(struct point x @a y @b)
```

Note that the pattern is specified plainly as `(point a b)` rather than `@(point a b)`, yet the expansion is `@(struct ...)`.

9.32.8 Special variable `*match-macro*`

Description:

The `*match-macro*` special variable holds the hash table of associations between symbols and pattern macro expanders.

If the expression `[*match-macro* 'sym]` yields a function, then symbol `sym` has a binding as a pattern macro. If that expression yields `nil`, then there is no such binding: pattern operator forms based on `sym` do not undergo place macro expansion.

The macro expanders in `*match-macro*` are two-parameter functions. The first argument passes the operator syntax to be expanded. The second argument is used for passing the environment object which the expander can capture using `:env` in its macro parameter list.

9.32.9 Macros `each-match` and `each-match-product`

Syntax:

```
(each-match ({pattern seq-form}*) body-form*)
(each-match-product ({pattern seq-form}*) body-form*)
```

Description:

The `each-match` macro arranges for elements from multiple sequences to be visited in parallel, and each to be matched against respective patterns. For each matching tuple of parallel elements, a body of forms is evaluated in the scope of the variables bound in the patterns.

The first argument of `each-match` specifies a list of alternating *`pattern`* and *`seq-form`* expressions. Each *`pattern`* is associated with the sequence which results from evaluating the immediately following *`seq-form`*. Items coming from that sequence correspond with that pattern.

The remaining arguments are *`body-form`*s to evaluated for successful matches.

The *`body-form`*s are surrounded by an implicit anonymous block. If any of the forms `return` invoke a return out of this block, then the iteration terminates, and the result value of the block becomes the result value of the loop.

The processing takes place as follows:

1. Every *seq-form* is evaluated in left-to-right order and is expected to produce an iterable sequence or object that would be a suitable argument to `mapcar` or `iter-begin`. This evaluation takes place in the scope surrounding the macro form, in which none of the variables that are bound in the *pattern* expressions are yet visible.
2. The next available item is taken from each of the sequences. If any of the sequences has no more items available, then `each-match` terminates and returns `nil`.
3. Each item taken in step 2 is matched against the *pattern* which is corresponds with its sequence. Each successive pattern can refer to the variables bound in the previous patterns in the same iteration. If any pattern match fails, then the process continues with step 2.
4. If all the matches are successful, then *body-form*s, if any, are executed in the scope of variables bound in the *pattern*s. Processing then continues at step 2.

The `each-match-product` differs from `each-match` in that instead of taking parallel tuples of items from the sequences, it iterates over the tuples of the Cartesian product of the sequences similarly to the `maprod` function. The product tuples are ordered in such a way that the rightmost element, which always coming coming from sequence produced by the last *seq-form*, varies the fastest. If there are two sequences `(1 2)` and `(a b)`, then `each-match` iterates over the tuples `(1 a)` and `(2 b)`, whereas `each-match-product` iterates over `(1 a)`, `(1 b)`, `(2 a)` and `(2 b)`.

Examples:

```
;; Number all the .JPG files in the current directory.
;; For instance foo.jpg becomes foo-0001.jpg, if it is
;; the first file.
(each-match (@(as name `@base.jpg`) (glob "*.jpg")
             @(@num (fmt "~,04a")) 1)
  (rename-path name `@base-@num.jpg`))

;; Iterate over combinations of matching phone
;; numbers and odd integers from the (1 2 3) list
(build
  (each-match-product (`(@a) @b-@c` '("x"
                                      ""
                                      "(311) 555-5353"
                                      "(604) 923-2323"
                                      "133"
                                      "4-5-6-7")
                      @(oddp @x) '(1 2 3))
   (add (list x a b c))))
-->
((1 "311" "555" "5353") (3 "311" "555" "5353")
 (1 "604" "923" "2323") (3 "604" "923" "2323")))
```

9.32.10 Macros `append-matches` and `append-match-products`

Syntax:

```
(append-matches ({pattern seq-form}*) body-form*)
(append-match-products ({pattern seq-form}*) body-form*)
```

Description:

The macro `append-matches` is subject to all of the requirements specified for `each-match` in regard to the argument conventions and semantics, and the presence of the implicit anonymous

block around the *body-form*s.

Whereas each-match returns nil, the append-matches macro requires, in each iteration which produces a match for each *pattern*, that the last *body-form* evaluated must produce a list.

These lists are catenated together as if by the append function and returned.

It is unspecified whether the nonmatching iterations produce empty lists which are included in the append operation.

If the last tuple of items which produces a match is absolutely the the last tuple, the corresponding *body-form* evaluation may yield an atom which then becomes the terminator for the returned list, in keeping with the semantics of append. an atom.

The append-match-products macro differs from append-matches in that it iterates over the Cartesian product tuples of the sequences, rather than parallel tuples. The difference is exactly like that between each-match and each-match-product.

Examples:

```
(append-matches
  ((:foo @y) '((:foo a) (:bar b) (:foo c) (:foo d))
   (@x :bar) '((1 :bar)  (2 :bar) (3 :bar) (4 :foo)))
  (list x y))
--> (1 a 3 c)

(append-matches (@x '((1) (2) (3) 4)) x)
--> (1 2 3 . 4)

(append-match-products (@(oddp @x) (range 1 5)
                        @(evenp @y) (range 1 5))
  (list x y))
--> (1 2 1 4 3 2 3 4 5 2 5 4)
```

9.32.11 Macros keep-matches and keep-match-products

Syntax:

```
(keep-matches ({pattern seq-form}*) body-form*)
(keep-match-products ({pattern seq-form}*) body-form*)
```

Description:

The macro keep-matches is subject to all of the requirements specified for each-match in regard to the argument conventions and semantics, and the presence of the implicit anonymous block around the *body-form*s.

Whereas each-match returns nil, the keep-matches macro returns a list of the values produced by all matching iterations which led to the execution of the *body-form*s.

The keep-match-products macro differs from keep-matches in that it iterates over the Cartesian product tuples of the sequences, rather than parallel tuples. The difference is exactly like that between each-match and each-match-product.

Examples:

```
(keep-matches ((:foo @y) '((:foo a) (:bar b) (:foo c) (:foo d))
              (@x :bar) '((1 :bar)  (2 :bar) (3 :bar) (4 :foo)))
  (list x y))
--> ((1 a) (3 c))

(keep-match-products (@(oddp @x) (range 1 5)
                     @(evenp @y) (range 1 5))
  (list x y))
--> ((1 2) (1 4) (3 2) (3 4) (5 2) (5 4))
```

9.32.12 Macro `while-match`

Syntax:

```
(while-match pattern expr form*)
```

Description:

The `while-match` macro evaluates *expr* and matches it against *pattern* similarly to `when-match`.

If the match is successful, every *form* is evaluated in an environment in which new bindings from *pattern* are visible. In this case, the process repeats: *expr* is evaluated again, and tested against *pattern*.

If the match fails, `while-match` terminates and produces `nil` as its result value.

Each iteration produces fresh bindings for any variables that are implicated for binding in *pattern*.

The *expr* and *form* expressions are surrounded by an anonymous block.

9.32.13 Macros `while-match-case` and `while-true-match-case`

Syntax:

```
(while-match-case expr {(pattern form*)}*)
(while-true-match-case expr {(pattern form*)}*)
```

Description:

The macros `while-match-case` and `while-true-match-case` combine iteration with the semantics of `match-case`.

The `while-match-case` evaluates *expr* and matches it against zero or more clauses in the manner of `match-case`. If there is a match, this process is repeated. If there is no match, `while-match-case` terminates, and returns `nil`.

In each iteration, the matching clause produces fresh bindings for any variables implicated for binding in its respective *pattern*.

The *expr* and *form* expressions are surrounded by an anonymous block.

The `while-true-match-case` macro is identical in almost every respect to `while-match-case`, except that it terminates the loop if *expr* evaluates to `nil`, without attempting to match that value against the clauses.

Note: the semantics of `while-true-match-case` can be obtained in `while-match-case` by inserting a `return` clause. That is to say, a construct of the form

```
(while-true-match-case expr
  ...)
```

may be rewritten into

```
(while-match-case expr
  (nil (return)) ;; match nil and return
  ...)
```

except that `while-true-match-case` isn't required to rely on performing a block return.

9.33 Quasiquote Operator Syntax

9.33.1 Macro `qquote`

Syntax:

(`qquote` *form*)

Description:

The `qquote` (quasi-quote) macro operator implements a notation for convenient list construction. If *form* is an atom, or a list structure which does not contain any `unquote` or `splice` operators, then (`qquote` *form*) is equivalent to (`qquote` *form*).

If *form*, however, is a list structure which contains `unquote` or `splice` operators, then the substitutions implied by those operators are performed on *form*, and the `qquote` operator returns the resulting structure.

Note: how the qquote operator actually works is that it is compiled into code. It becomes a Lisp expression which, when evaluated, computes the resulting structure.

A `qquote` can contain another `qquote`. If an `unquote` or `splice` operator occurs within a nested `qquote`, it belongs to that `qquote`, and not to the outer one.

However, an unquote operator which occurs inside another one belongs one level higher. For instance in

```
(qquote (qquote (unquote (unquote x))))
```

the leftmost `qquote` belongs with the rightmost unquote, and the inner `qquote` and `unquote` belong together. When the outer `qquote` is evaluated, it will insert the value of `x`, resulting in the object `(qquote (unquote [value-of-x]))`. If this resulting qquote value is evaluated again as Lisp syntax, then it will yield `[value-of-value-of-x]`, the value of `[value-of-x]` when treated as a Lisp expression and evaluated.

Examples:

```
(qquote a) -> a

(qquote (a b c)) -> (a b c)

(qquote (1 2 3 (unquote (+ 2 2)) (+ 2 3))) -> (1 2 3 4 (+ 2 3))
```

```
(qquote (unquote (+ 2 2))) -> 4
```

In the second-to-last example, the `1 2 3` and the `(+ 2 3)` are quoted verbatim. Whereas the `(unquote (+ 2 2))` operator caused the evaluation of `(+ 2 2)` and the substitution of the resulting value.

The last example shows that *form* can itself (the entire argument of `qquote`) can be an unquote operator. However, note: `(quote (splice form))` is not valid.

Note: a way to understand the nesting behavior is a via a possible model of quasi-quote expansion which recursively compiles any nested quasi quotes first, and then treats the result of their expansion. For instance, in the processing of

```
(qquote (qquote (unquote (unquote x))))
```

the `qquote` operator first encounters the embedded `(qquote ...)` and compiles it to code. During that recursive compilation, the syntax `(unquote (unquote x))` is encountered. The inner quote processes the outer unquote which belongs to it, and the inner `(unquote x)` becomes material that is embedded verbatim in the compilation, which will then be found when the recursion pops back to the outer quasiquote, which will then traverse the result of the inner compilation and find the `(unquote x)`.

Dialect Note:

In Lisp dialects which have a published quasiquoting operator syntax, there is the expectation that the quasiquote read syntax corresponds to it. That is to say, the read syntax `^(a b ,c)` is expected to translate to `(qquote a b (unquote c))`.

In **TXR Lisp**, this is not true! Although `^(a b ,c)` is translated to a quasiquoting macro, it is an internal one, not based on the public `qquote`, `unquote` and `splice` symbols being documented here.

This idea exists for hygiene. The quasiquote read syntax is not confused by the presence of the symbols `qquote`, `unquote` or `splice` in the template, since it doesn't treat them specially.

This also allows programmers to use the quasiquote read syntax to construct quasiquote macros. For instance

```
^(qquote (unquote ,x)) ;; does not mean ^^,,x !
```

To the quasiquote reader, the `qquote` and `unquote` symbols mean nothing special, and so this syntax simply means that if the value of `x` is `foo`, the result of evaluating this expression will be `(qquote (unquote foo))`.

The form's expansion is actually this:

```
(sys:qquote (qquote (unquote (sys:unquote x))))
```

the `sys:qquote` macro recognizes `sys:unquote` embedded in the form, and the other symbols not in the `sys:` package are just static template material.

The `sys:quote` macro and its associated `sys:unquote` and `sys:splice` operators work exactly like their ordinary counterparts. So in effect, **TXR** has two nearly identical, independent quasi-quote implementations, one of which is tied to the read syntax, and one of which isn't. This is useful for writing quasiquotes which write quasiquotes.

9.33.2 Operator `unquote`

Syntax:

```
(qquote (... (unquote form) ...))
(qquote (unquote form))
```

Description:

The `unquote` operator is not an operator *per se*. The `unquote` symbol has no binding in the global environment. It is a special syntax that is recognized within a `qquote` form, to indicate forms within the quasiquote which are to be evaluated and inserted into the resulting structure.

The syntax `(qquote (unquote form))` is equivalent to *form*: the `qquote` and `unquote` "cancel out".

9.33.3 Operator `splice`

Syntax:

```
(qquote (... (splice form) ...))
```

Description:

The `splice` operator is not an operator *per se*. The `splice` symbol has no binding in the global environment. It is a special syntax that is recognized within a `qquote` form, to indicate forms within the quasiquote which are to be evaluated and inserted into the resulting structure.

The syntax `(qquote (splice form))` is not permitted and raises an exception if evaluated. The `splice` syntax must occur within a list, and not in the dotted position.

The `splice` form differs from unquote in that `(splice form)` requires that *form* must evaluate to a list. That list is integrated into the surrounding list.

9.34 Math Library

9.34.1 Functions +, − and *

Syntax:

```
(+ number*)
(- number number*)
(* number*)
```

Description:

The +, − and * functions perform addition, subtraction and multiplication, respectively. Additionally, the − function performs additive inverse.

The + function requires zero or more arguments. When called with no arguments, it produces 0 (the identity element for addition), otherwise it produces the sum over all of the arguments.

Similarly, the * function requires zero or more arguments. When called with no arguments, it produces 1 (the identity element for multiplication). Otherwise it produces the product of all the arguments.

The semantics of − changes from subtraction to additive inverse when there is only one argument. The argument is treated as a subtrahend, against an implicit minuend of zero. When there are two or more argument, the first one is the minuend, and the remaining are subtrahends.

When there are three or more operands, these operations are performed as if by binary operations, in a left-associative way. That is to say, `(+ a b c)` means `(+ (+ a b) c)`. The sum of `a` and `b` is computed first, and then this is added to `c`. Similarly `(- a b c)` means `(- (- a b) c)`. First, `b` is subtracted from `a`, and then `c` is subtracted from that result.

The arithmetic inverse is performed as if it were subtraction from integer 0. That is, `(- x)` means the same thing as `(- 0 x)`.

The operands of +, − and `*` can be characters, integers (fixnum and bignum), and floats, in nearly any combination.

If two operands have different types, then one of them is converted to the type of the one with the higher rank, according to this ranking: character < integer < float. For instance if one operand is integer, and the other float, the integer is converted to a float.

Restrictions:

Characters are not considered numbers, and participate in these operations in limited ways. Subtraction can be used to computed the displacement between the Unicode values of characters, and an integer displacement can be added to a character, or subtracted from a character. For instance `(- #\9 #\0) is 9`. The Unicode value of a character `C` can be found using `(- C #\x0)`: the displacement from the NUL character.

The rules can be stated as a set of restrictions:

1. Two characters may not be added together.
2. A character may not be subtracted from an integer (which also rules out the possibility of computing the additive inverse of a character).
3. A character operand may not be opposite to a floating point operand in any operation.
4. A character may not be an operand of multiplication.

9.34.2 Function /

Syntax:

```
(/ divisor)
(/ dividend divisor*)
```

Description:

The `/` function performs floating-point division. Each operands is first converted to floating-point type, if necessary. In the one-argument form, the *dividend* argument is omitted. An implicit dividend is present, whose value is `1.0`, such that the one-argument form `(/ x)` is equivalent to the two-argument form `(/ 1.0 x)`.

If there are two or more arguments, explicitly or by the above equivalence, then a cumulative division is performed. The *divisor* value is taken into consideration, and divided by the first `divisor`. If another `divisor` follows, then that value is divided by that subsequent divisor. This process repeats until all divisors are exhausted, and the value of the last division is returned.

A division by zero throws an exception of type `numeric-error`.

9.34.3 Functions `sum` and `prod`

Syntax:

```
(sum sequence [keyfun])
(prod sequence [keyfun])
```

Description:

The `sum` and `prod` functions operate on an effective sequence of numbers derived from *`sequence`*, which is an object suitable for iteration according to `seq-begin`.

If the *`keyfun`* argument is omitted, then the effective sequence is the *`sequence`* argument itself. Otherwise, the effective sequence is understood to be a projection mapping of the elements of *`sequence`* through *`keyfun`* as would be calculated by the `(mapcar keyfun sequence)` expression.

The `sum` function returns the left-associative sum of the elements of the effective sequence calculated as if using the `+` function. Similarly, the `prod` function calculates the left-associative product of the elements of the sequence as if using the `*` function.

If *`sequence`* is empty then `sum` returns `0` and `prod` returns `1`.

If the effective sequence contains one number, then both functions return that number.

9.34.4 Functions `wrap` and `wrap*`

Syntax:

```
(wrap start end number)
(wrap* start end number)
```

Description:

The `wrap` and `wrap*` functions reduce *`number`* into the range specified by *`start`* and *`end`*.

Under `wrap` the range is inclusive of the *`end`* value, whereas under `wrap*` it is exclusive.

The following equivalence holds

```
(wrap a b c) <--> (wrap* a (succ b) c)
```

The expression `(wrap* x0 x1 x)` performs the following calculation:

```
(+ (mod (- x x0) (- x1 x0)) x0)
```

In other words, first *`start`* is subtracted from *`number`*. Then the result is reduced modulo the displacement between `start` and `end`. Finally, *`start`* is added back to that result, which is returned.

Example:

```
;; perform ROT13 on the string "nop"
[mapcar (opip (+ 13) (wrap #\a #\z)) "nop"] -> "abc"
```

9.34.5 Functions `gcd` and `lcm`

Syntax:

```
(gcd number*)
(lcm number*)
```

Description:

The `gcd` function computes the greatest common divisor: the largest positive integer which divides each *number*.

The `lcm` function computes the lowest common multiple: the smallest positive integer which is a multiple of each *number*.

Each *number* must be an integer.

Negative integers are replaced by their absolute values, so `(lcm -3 -4)` is `12` and `(gcd -12 -9)` yields `3`.

The value of `(gcd)` is `0` and that of `(lcm)` is 1 .

The value of `(gcd x)` and `(lcm x)` is `(abs x)`.

Any arguments of `gcd` which are zero are effectively ignored so that `(gcd 0)` and `(gcd 0 0 0)` are both the same as `(gcd)` and `(gcd 1 0 2 0 3)` is the same as `(gcd 1 2 3)`.

If `lcm` has any argument which is zero, it yields zero.

9.34.6 Function `divides`

Syntax:

```
(divides d n)
```

Description:

The `divides` function tests whether integer *d* divides integer *n*. If this is true, `t` is returned, otherwise `nil`.

The integers 1 and -1 divide every other integer and themselves. By established convention, every integer, except zero, divides zero.

For other values, *d* divides *n* if division of *n* by *d* leaves no remainder.

9.34.7 Function `abs`

Syntax:

```
(abs number)
```

Description:

The `abs` function computes the absolute value of *number*. If *number* is positive, it is returned. If *number* is negative, its additive inverse is returned: a positive number of the same type with exactly the same magnitude.

9.34.8 Function `signum`

Syntax:

```
(signum number)
```

Description:

The `signum` function calculates a representation of the sign of *number* as a numeric value.

If *number* is an integer, then signum returns -1 if the integer is negative, 1 if the integer is positive, or else 0.

If *number* is a floating-point value then signum returns -1.0 if the value is negative, 1.0 if the value is positive or else 0.0.

9.34.9 Functions trunc, floor, ceil and round

Syntax:

```
(trunc dividend [divisor])
(floor dividend [divisor])
(ceil dividend [divisor])
(round dividend [divisor])
```

Description:

The trunc, floor, ceil and round functions perform division of the *dividend* by the *divisor*, returning an integer quotient.

If the *divisor* is omitted, it defaults to 1.

A zero *divisor* results in an exception of type numeric-error.

If both inputs are integers, the result is of type integer.

If all inputs are numbers and at least one of them is floating-point, the others are converted to floating-point and the result is floating-point.

The *dividend* input may be a range. In this situation, the operation is recursively distributed over the from and to fields of the range, individually matched against the *divisor*, and the result is a range composed of these two individual quotients.

When the quotient is a scalar value, trunc returns the closest integer, in the zero direction, from the value of the quotient. The floor function returns the highest integer which does not exceed the value of the quotient. That is to say, the division is truncated to an integer value toward negative infinity. The ceil function the lowest integer which is not below the value of the quotient. does not exceed the value of *dividend*. That is to say, the division is truncated to an integer value toward positive infinity. The round function returns the nearest integer to the quotient. Exact halfway cases are rounded to the integer away from zero so that (round -1 2) yields -1 and (round 1 2) yields 1.

Note that for large floating point values, due to the limited precision, the integer value corresponding to the mathematical floor or ceiling may not be available.

Dialect Note:

In ANSI Common Lisp, the round function chooses the nearest even integer, rather than rounding halfway cases away from zero. **TXR**'s choice harmonizes with the semantics of the round function in the C language.

9.34.10 Function mod

Syntax:

```
(mod dividend divisor)
```

Description:

The `mod` function performs a modulus operation. Firstly, the absolute value of *divisor* is taken to be a modulus. Then a residue of *dividend* with respect to *modulus* is calculated. The residue's sign follows that of the sign of *divisor*. That is, it is the smallest magnitude (closest to zero) residue of *dividend* with respect to the absolute value of *divisor*, having the same sign as *divisor*. If the operands are integer, the result is an integer. If either operand is of type float, then the result is a float. The modulus operation is then generalized into the floating point domain. For instance the expression `(mod 0.75 0.5)` yields a residue of 0.25 because 0.5 "goes into" 0.75 only once, with a "remainder" of 0.25.

If *divisor* is zero, `mod` throws an exception of type `numeric-error`.

9.34.11 Functions `trunc-rem`, `floor-rem`, `ceil-rem` and `round-rem`

Syntax:

```
(trunc-rem dividend [divisor])
(floor-rem dividend [divisor])
(ceil-rem dividend [divisor])
(round-rem dividend [divisor])
```

Description:

These functions, respectively, perform the same division operation as `trunc`, `floor`, `ceil`, and `round`, referred to here as the respective target functions.

If the *divisor* is missing, it defaults to 1.

Each function returns a list of two values: a *quotient* and a *remainder*. The *quotient* is exactly the same value as what would be returned by the respective target function for the same inputs.

The *remainder* value obeys the following identity:

```
(eql remainder (- dividend (*divisor quotient)))
```

If *divisor* is zero, these functions throw an exception of type `numeric-error`.

9.34.12 Functions `sin`, `cos`, `tan`, `asin`, `acos`, `atan` and `atan2`

Syntax:

```
(sin radians)
(cos radians)
(tan radians)
(atan slope)
(atan2 y x)
(asin num)
(acos num)
```

Description:

These trigonometric functions convert their argument to floating point and return a float result. The `sin`, `cos` and `tan` functions compute the sine and cosine and tangent of the *radians* argument which represents an angle expressed in radians. The `atan`, `acos` and `asin` are their respective inverse functions. The *num* argument to `asin` and `acos` must be in the range -1.0 to 1.0. The `atan2` function converts the rectilinear coordinates *x* and *y* to an angle in polar coordinates in the

range [0, 2π).

9.34.13 Functions `sinh`, `cosh`, `tanh`, `asinh`, `acosh` and `atanh`

Syntax:

```
(sinh argument)
(cosh argument)
(tanh argument)
(atanh argument)
(asinh argument)
(acosh argument)
```

Description:

These functions are the hyperbolic analogs of the trigonometric functions `sin`, `cos` and so forth. They convert their argument to floating point and return a float result.

9.34.14 Functions `exp`, `log`, `log10` and `log2`

Syntax:

```
(exp arg)
(log arg)
(log10 arg)
(log2 arg)
```

Description:

The `exp` function calculates the value of the transcendental number e raised to the exponent *`arg`*.

The `log` function calculates the base e logarithm of *`arg`*, which must be a positive value.

The `log10` function calculates the base 10 logarithm of *`arg`*, which must be a positive value.

The `log2` function calculates the base 2 logarithm of *`arg`*, which must be a positive value.

9.34.15 Functions `expt`, `sqrt` and `isqrt`

Syntax:

```
(expt base exponent*)
(sqrt arg)
(isqrt arg)
```

Description:

The `expt` function raises *`base`* to zero or more exponents given by the *`exponent`* arguments. `(expt x)` is equivalent to `(expt x 1)`, and yields `x` for all `x`. For three or more arguments, the operation is right-associative. That is to say, `(expt x y z)` is equivalent to `(expt x (expt y z))`, similarly to the way nested exponents work in standard algebraic notation.

Exponentiation is done pairwise using a binary operation. If both operands to this binary operation are nonnegative integers, then the result is an integer.

If the exponent is negative, and the base is zero, the situation is treated as a division by zero: an exception of type `numeric-error` is thrown. Otherwise, a negative exponent is converted to floating-point, if it already isn't, and a floating-point exponentiation is performed.

If either operand is a float, then the other operand is converted to a float, and a floating point

exponentiation is performed. Exponentiation that would produce a complex number is not supported.

If the exponent is zero, then the return value is 1.0 if at least one operand is floating-point, otherwise 1.

The `sqrt` function produces a floating-point square root of *arg*, which is converted from integer to floating-point if necessary. Negative operands are not supported.

The `isqrt` function computes the integer square root of *arg*, which must be an integer. The integer square root is a value which is the greatest integer that is no greater than the real square root of *arg*. The input value must be an integer.

9.34.16 Function `exptmod`

Syntax:

```
(exptmod base exponent modulus)
```

Description:

The `exptmod` function performs modular exponentiation and accepts only integer arguments. Furthermore, *exponent* must be a nonnegative and *modulus* must be positive.

The return value is *base* raised to *exponent*, and reduced to the least positive residue modulo *modulus*.

9.34.17 Function `square`

Syntax:

```
(square argument)
```

Description:

The `square` function returns the product of *argument* with itself. The following equivalence holds, except that `x` is evaluated only once in the `square` expression:

```
(square x)  <-->  (* x x)
```

9.34.18 Function `cum-norm-dist`

Syntax:

```
(cum-norm-dist argument)
```

Description:

The `cum-norm-dist` function calculates an approximation to the cumulative normal distribution function: the integral, of the normal distribution function, from negative infinity to the *argument*.

9.34.19 Function `inv-cum-norm`

Syntax:

```
(inv-cum-norm argument)
```

Description:

The `inv-cum-norm` function calculates an approximate to the inverse of the cumulative normal

distribution function. The argument, a value expected to lie in the range [0, 1], represents the integral of the normal distribution function from negative infinity to some domain point *p*. The function calculates the approximate value of *p*. The minimum value returned is -10, and the maximum value returned is 10, regardless of how closely the argument approaches, respectively, the 0 or 1 integral endpoints. For values less than zero, or exceeding 1, the values returned, respectively, are -10 and 10.

9.34.20 Functions `n-choose-k` and `n-perm-k`

Syntax:

```
(n-choose-k n k)
(n-perm-k n k)
```

Description:

The `n-choose-k` function computes the binomial coefficient nCk which expresses the number of combinations of *k* items that can be chosen from a set of *n*, where combinations are subsets.

The `n-perm-k` function computes nPk: the number of permutations of size *k* that can be drawn from a set of *n*, where permutations are sequences, whose order is significant.

The calculations only make sense when *n* and *k* are nonnegative integers, and *k* does not exceed *n*. The behavior is not specified if these conditions are not met.

9.34.21 Functions `fixnump`, `bignump`, `integerp`, `floatp` and `numberp`

Syntax:

```
(fixnump object)
(bignump object)
(integerp object)
(floatp object)
(numberp object)
```

Description:

These functions test the type of *object*, returning `t` if it is an object of the implied type, `nil` otherwise. The `fixnump`, `bignump` and `floatp` functions return `t` if the object is of the basic type `fixnum`, `bignum` or `float`. The function `integerp` returns true of *object* is either a `fixnum` or a `bignum`. The function `numberp` returns `t` if *object* is either a `fixnum`, `bignum` or `float`.

9.34.22 Function `arithp`

Syntax:

```
(arithp object)
```

Description:

The `arithp` function returns true if *object* is a character, integer, floating-point number, range or a user-defined arithmetic object. For a range, `t` is returned without examining the values of the `from` and `to` fields. A user-defined arithmetic object is identified as a struct type which implements the + method as a static slot.

9.34.23 Functions `zerop` and `nzerop`

Syntax:

```
(zerop number)
(nzerop number)
```

Description:

The `zerop` function tests *number* for equivalence to zero. The argument must be a number or character. It returns `t` for the integer value `0` and for the floating-point value `0.0`. For other numbers, it returns `nil`. It returns `t` for the null character `#\nul` and `nil` for all other characters.

If *number* is a range, then `zerop` returns `t` if both of the range endpoints individually satisfy `zerop`.

The `nzerop` function is the logical inverse of `zerop`: it returns `t` for those arguments for which `zerop` returns `nil` and vice versa.

9.34.24 Functions `plusp` and `minusp`

Syntax:

```
(plusp number)
(minusp number)
```

Description:

These functions test whether a number is positive or negative, returning `t` or `nil`, as the case may be.

The argument may also be a character. All characters other than the null character `#\nul` are positive. No character is negative.

9.34.25 Functions `evenp` and `oddp`

Syntax:

```
(evenp integer)
(oddp integer)
```

Description:

The `evenp` and `oddp` functions require integer arguments. `evenp` returns `t` if *integer* is even (divisible by two), otherwise it returns `nil`. `oddp` returns `t` if *integer* is not divisible by two (odd), otherwise it returns `nil`.

9.34.26 Functions `succ`, `ssucc`, `sssucc`, `pred`, `ppred` and `pppred`

Syntax:

```
(succ number)
(ssucc number)
(sssucc number)
(pred number)
(ppred number)
(pppred number)
```

Description:

The `succ` function adds 1 to its argument and returns the resulting value. If the argument is an integer, then the return value is the successor of that integer, and if it is a character, then the return value is the successor of that character according to Unicode.

The `pred` function subtracts 1 from its argument, and under similar considerations as above, the

result represents the predecessor.

The `ssucc` and `sssucc` functions add 2 and 3, respectively. Similarly, `ppred` and `pppred` subtract 2 and 3 from their argument.

9.34.27 Functions >, <, >=, <= and =

Syntax:

```
(> object object*)
(< object object*)
(>= object object*)
(<= object object*)
(= object object*)
```

Description:

These relational functions compare characters, numbers, ranges and sequences of characters or numbers for numeric equality or inequality. The arguments must be one or more numbers, characters, ranges, or sequences of these objects, or, recursively, of sequences.

If just one argument is given, then these functions all return `t`.

If two arguments are given then, they are compared as follows. First, if the numbers do not have the same type, then the one which has the lower ranking type is converted to the type of the other, according to this ranking: character < integer < float. For instance if a character and integer are compared, the character is converted to its integer character code. Then a numeric comparison is applied.

Three or more arguments may be given, in which case the comparison proceeds pairwise from left to right. For instance in `(< a b c)`, the comparison `(< a b)` is performed in isolation. If the comparison is false, then `nil` is returned, otherwise the comparison `(< b c)` is performed in isolation, and if that is false, `nil` is returned, otherwise `t` is returned. Note that it is possible for `b` to undergo two different conversions. For instance in the `(< float character integer)` comparison, *character* will first convert to a floating-point representation of its Unicode value so that it can be compared to *float*, and if that comparison succeeds, then in the second comparison, *character* will be converted to integer so that it can be compared to *integer*.

Ranges may only be compared with ranges. Corresponding fields of ranges are compared for equality by = such that `#R(0 1)` and `#R(0 1.0)` are reported as equal. The inequality comparisons are lexicographic, such that the `from` field of the range is considered more major than the `to` field. For example the inequalities `(< #R(1 2) #R(2 0))` and `(< #R(1 2) #R(1 3))` hold.

Sequences may only be compared with sequences, but mixtures of any kinds of sequences may be compared: lists with vectors, vectors with strings, and so on.

The = function considers a pair of sequences of unequal length to be unequal, reporting `nil`. Sequences are equal if they have the same length and their corresponding elements are recursively equal under the = function.

The inequality functions treat sequences lexicographically. A pair of sequences is compared by comparing corresponding elements. The < function tests each successive pair of corresponding elements recursively using the < function. If this recursive comparison reports `t`, then the function immediately returns `t` without considering any more pairs of elements. Otherwise the same pair of elements is compared again using the = function. If that reports false, then the function reports

false without considering any more pairs of elements. Otherwise processing continues with the next pair, if any. If all corresponding elements are equal, but the right sequence is longer, < returns `t`, otherwise the function reports `nil`. The <= function tests each successive pair of corresponding elements recursively using the <= function. If this returns `nil` then the function returns `nil` without considering any more pairs. Otherwise processing continues with the next pair, if any. If all corresponding elements satisfy the test, but the left sequence is longer, then `nil` is returned. Otherwise `t` is returned.

The inequality relations exhibit symmetry, which means that the functions > and >= functions are equivalent, respectively, to < and <= with the order of the argument values reversed. For instance, the expression `(< a b c)` is equivalent to `(> c b a)` except for the difference in evaluation order of the `a`, `b` and `c` operands themselves. Any semantic description of < or <= applies, respectively, also to > or >= with the appropriate adjustment for argument order reversal.

9.34.28 Function `/=`

Syntax:

```
(/= number*)
```

Description:

The arguments to `/=` may be numbers or characters. The `/=` function returns `t` if no two of its arguments are numerically equal. That is to say, if there exist some `a` and `b` which are distinct arguments such that `(= a b)` is true, then the function returns `nil`. Otherwise it returns `t`.

9.34.29 Functions `max` and `min`

Syntax:

```
(max first-arg arg*)
(min first-arg arg*)
```

Description:

The `max` and `min` functions determine and return the highest or lowest value from among their arguments.

If only *`first-arg`* is given, that value is returned.

These functions are type generic, since they compare arguments using the same semantics as the `less` function.

If two or more arguments are given, then `(max a b)` is equivalent to `(if (less a b) b a)`, and `(min a b)` is equivalent to `(if (less a b) a b)`. If the operands do not have the same type, then one of them is converted to the type of the other; however, the original unconverted values are returned. For instance `(max 4 3.0)` yields the integer `4`, not `4.0`.

If three or more arguments are given, `max` and `min` reduce the arguments in a left-associative manner. Thus `(max a b c)` means `(max (max a b) c)`.

9.34.30 Function `clamp`

Syntax:

```
(clamp low high val)
```

Description:

The `clamp` function clamps value *`val`* into the range *`low`* to *`high`*.

The `clamp` function returns *`low`* if *`val`* is less than *`low`*. If *`val`* is greater than or equal to *`low`*, but less than *`high`*, then it returns *`val`*. Otherwise it returns *`high`*.

More precisely, `(clamp a b c)` is equivalent to `(max a (min b c))`.

9.34.31 Function `bracket`

Syntax:

```
(bracket value level*)
```

Description:

The `bracket` function's arguments consist of one required *`value`* followed by *n* *`level`* arguments. The *`level`* arguments are optional; in other words, *n* may be zero.

The `bracket` function calculates the *bracket* of the *`value`* argument: a zero-based positional index of the value, in relation to the *`level`* arguments.

Each of the *`level`* arguments, of which there may be none, is associated with an integer index, starting at zero, in left-to-right order. The *`level`* arguments are examined in that order. When a *`level`* argument is encountered which exceeds *`value`*, that *`level`* argument's index is returned. If *`value`* exceeds all of the *`level`* arguments, then *n* is returned.

Determining whether *`value`* exceeds a *`level`* is performed using the `less` function.

Examples:

```
(bracket 42) -> 0
(bracket 5 10) -> 0
(bracket 15 10) -> 1
(bracket 15 10 20) -> 1
(bracket 15 10 20 30) -> 1
(bracket 20 10 20 30) -> 2
(bracket 35 10 20 30) -> 3
(bracket "a" "aardvark" "zebra") -> 0
(bracket "ant" "aardvark" "zebra") -> 1
(bracket "zebu" "aardvark" "zebra") -> 2
```

9.34.32 Functions `int-str`, `flo-str` and `num-str`

Syntax:

```
(int-str string [radix])
(flo-str string)
(num-str string)
```

Description:

These functions extract numeric values from character string *`string`*. Leading whitespace in *`string`*, if any, is skipped. If no digits can be successfully extracted, then `nil` is returned. Trailing material which does not contribute to the number is ignored.

The `int-str` function converts a string of digits in the specified *`radix`* to an integer value. If

radix isn't specified, it defaults to 10. Otherwise it must be an integer in the range 2 to 36, or else the character `#\c`.

For radices above 10, letters of the alphabet are used for digits: `A` represent a digit whose value is 10, `B` represents 11 and so forth until `Z`. Uppercase and lowercase letters are recognized. Any character which is not a digit of the specified radix is regarded as the start of trailing junk at which the extraction of the digits stops.

When *radix* is specified as the character object `#\c`, this indicates that a C-language-style integer constant should be recognized. If, after any optional sign, the remainder of *string* begins with the character pair `0x` then that pair is considered removed from the string, and it is treated as base 16 (hexadecimal). If, after any optional sign, the remainder of *string* begins with a leading zero not followed by `x`, then the radix is taken to be 8 (octal). In scanning these formats, `int-str` function is not otherwise constrained by C language representational limitations. Specifically, the input values are taken to be the printed representation of arbitrary-precision integers and treated accordingly.

The `flo-str` function converts a floating-point decimal notation to a nearby floating point value. The material which contributes to the value is the longest match for optional leading space, followed by a mantissa which consists of an optional sign followed by a mixture of at least one digit, and at most one decimal point, optionally followed by an exponent part denoted by the letter `E` or `e`, an optional sign and one or more optional exponent digits. If the value specified by *string* is out of range of the floating-point representation, then `nil` is returned.

The `num-str` function converts a decimal notation to either an integer as if by a radix 10 application of `int-str`, or to a floating point value as if by `flo-str`. The floating point interpretation is chosen if the possibly empty initial sequence of digits (following any whitespace and optional sign) is followed by a period, or by `e` or `E`.

9.34.33 Functions `int-flo` and `flo-int`

Syntax:

```
(int-flo float)
(flo-int integer)
```

Description:

These functions perform numeric conversion between integer and floating point type. The `int-flo` function returns an integer by truncating toward zero. The `flo-int` function returns an exact floating point value corresponding to *integer*, if possible, otherwise an approximation using a nearby floating point value.

9.34.34 Functions `tofloat` and `toint`

Syntax:

```
(tofloat value)
(toint value [radix])
```

Description:

These functions convert *value* to floating-point or integer, respectively. The *value* can be of several types, including string.

If a floating-point value is passed into tofloat, or an integer value into toint, then that value is simply returned.

If *value* is a character, then it is treated as a string of length one containing that character.

If *value* is a string, then it is converted by `tofloat` as if by the function *flo-str*, and by `toint` as if by the function `int-str`.

If *value* is an integer, then it is converted by `tofloat` as if by the function `flo-int`.

If *value* is a floating-point number, then it is converted by `toint` as if by the function `int-flo`.

9.34.35 Variables `fixnum-min` and `fixnum-max`

Description:

These variables hold, respectively, the most negative value of the `fixnum` integer type, and its most positive value. Integer values from `fixnum-min` to `fixnum-max` are all of type `fixnum`. Integers outside of this range are `bignum` integers.

9.34.36 Functions `tofloatz` and `tointz`

Syntax:

```
(tofloatz value)
(tointz value [radix])
```

Description:

These functions are closely related to, respectively, `tofloat` and `toint`. They differ in that these functions return a floating-point or integer zero, respectively, in some situations in which those functions would return `nil` or throw an error.

Whereas those functions reject a *value* argument of `nil`, for that same argument `tofloatz` function returns 0.0 and `tointz` returns 0.

Likewise, in cases when `value` contains a string or character which cannot be converted to a number, and `tofloat` and `toint` would return `nil`, these functions return 0.0 and 0, respectively.

In other situations, these functions behave exactly like `tofloat` and `toint`.

9.34.37 Variables `flo-min`, `flo-max` and `flo-epsilon`

Description:

These variables hold, respectively: the smallest positive floating-point value; the largest positive floating-point value; and the difference between 1.0 and the smallest representable value greater than 1.0.

`flo-min` and `flo-max` define the floating-point range, which consists of three regions: values from `(- flo-max)` to `(- flo-min)`; the value 0.0, and values from `flo-min` to `flo-max`.

9.34.38 Variable `flo-dig`

Description:

This variable holds an integer representing the number of decimal digits in a decimal floating-point number such that this number can be converted to a **TXR** floating-point number, and back to decimal, without a change in any of the digits. This holds regardless of the value of the number,

provided that it does not exceed the floating-point range.

9.34.39 Variable `flo-max-dig`

Description:

This variable holds an integer representing the maximum number of decimal digits required to capture the value of a floating-point number such that the resulting decimal form will convert back to the same floating-point number. See also the `*print-flo-precision*` variable.

9.34.40 Variables `%pi%` and `%e%`

Description:

These variables hold an approximation of the mathematical constants π and e. To four digits of precision, π is 3.142 and e is 2.718. The `%pi%` and `%e%` approximations are accurate to `flo-dig` decimal digits.

9.34.41 Function `digits`

Syntax:

```
(digits number [radix])
```

Description:

The `digits` function returns a list of the digits of *`number`* represented in the base given by *`radix`*.

The *`number`* argument must be a nonnegative integer, and *`radix`* must be an integer greater than one.

If *`radix`* is omitted, it defaults to 10.

The return value is a list of the digits in descending order of significance: most significant to least significant. The digits are integers. For instance, if *`radix`* is 42, then the digits are integer values in the range 0 to 41.

The returned list always contains at least one element, and includes no leading zeros, except when *`number`* is zero. In that case, a one-element list containing zero is returned.

Examples:

```
(digits 1234) -> (1 2 3 4)
(digits 1234567 1000) -> (1 234 567)
(digits 30 2) -> (1 1 1 1 0)
(digits 0) -> (0)
```

9.34.42 Function `digpow`

Syntax:

```
(digpow number [radix])
```

Description:

The `digpow` function decomposes the *`number`* argument into a power series whose terms add up to *`number`*.

The *number* argument must be a nonnegative integer, and *radix* must be an integer greater than one.

The returned power series consists of a list of nonnegative integers. It is formed from the digits of *number* in the given *radix*, which serve as coefficients which multiply successive powers of the *radix*, starting at the zeroth power (one).

The terms are given in decreasing order of significance: the term corresponding to the most significant digit of *number*, multiplying the highest power of *radix*, is listed first.

The returned list always contains at least one element, and includes no leading zeros, except when *number* is zero. In that case, a one-element list containing zero is returned.

```
(digpow 1234) -> (1000 200 30 4)
(digpow 1234567 1000) -> (1000000 234000 567)
(digpow 30 2) -> (16 8 4 2 0)
(digpow 0) -> (0)
```

9.34.43 Functions `poly` and `rpoly`

Syntax:

```
(poly arg coeffs)
(rpoly arg coeffs)
```

Description:

The `poly` and `rpoly` functions evaluate a polynomial, for the given numeric argument value *arg* and the coefficients given by *coeffs*, a sequence of numbers.

If *coeffs* is an empty sequence, it denotes the zero polynomial, whose value is zero everywhere; the functions return zero in this case.

Otherwise, the `poly` function considers *coeffs* to hold the coefficients in the conventional order, namely in order of decreasing degree of polynomial term. The first element of *coeffs* is the leading coefficient, and the constant term appears as the last element.

The `rpoly` function takes the coefficients in opposite order: the first element of *coeffs* gives the constant term coefficient, and the last element gives the leading coefficient.

Note: except in the case of `rpoly` operating on a list or list-like sequence of coefficients, Horner's method of evaluation is used: a single result accumulator is initialized with zero, and then for each successive coefficient, in order of decreasing term degree, the accumulator is multiplied by the argument, and the coefficient is added. When `rpoly` operates on a list or list-like sequence, it makes a single pass through the coefficients in order, thus taking them in increasing term degree. It maintains two accumulators: one for successive powers of *arg* and one for the resulting value. For each coefficient, the power accumulator is updated by a multiplication by *arg* and then this value is multiplied by the coefficient, and that value is then added to the result accumulator.

Examples:

```
;;            2
;; evaluate x  +  2x + 3  for x = 10.
(poly 10 '(1 2 3)) -> 123

;;             2
```

```
;; evaluate 3x  +  2x + 1  for x = 10.
(rpoly 10 '(1 2 3)) -> 321
```

9.34.44 Function `bignum-len`

Syntax:

```
(bignum-len arg)
```

Description:

The `bignum-len` function reports the machine-specific *bignum order* of the integer or character argument *`arg`*.

If *`arg`* is a character or `fixnum` integer, the function returns zero.

Otherwise *`arg`* is expected to be a `bignum` integer, and the function returns the number of "limbs" used for its representation, a positive integer.

Note: the `bignum-len` function is intended to be of use in algorithms whose performance benefits from ordering the operations on multiple integer operands according to the magnitudes of those operands. The function provides an estimate of magnitude which trades accuracy for efficiency.

9.34.45 Function `quantile`

Syntax:

```
(quantile p [group-size [rate]])
```

Description:

The `quantile` function returns a function which estimates a specific quantile of a set of real-valued samples. The desired quantile is indicated by the *`p`* parameter, which is a number in the range 0 to 1.0. If *`p`* is specified as 0.5, then the median is estimated. The *`p`* value of 0.9 leads to the estimation of the 90th percentile: a value such that approximately 90% of the samples are below that value.

If the *`group-size`* parameter is specified, it must be a positive integer. The returned function then operates in grouped mode. The *`rate`* parameter is relevant only to grouped mode. Grouped mode is described below.

The function returned by `quantile` maintains internal state in relation to calculating the quantile. The function may be called with any number of arguments, including none. It expects every argument to be either a number, or a sequence of numbers. These numbers are accumulated into the quantile calculation, and a revised estimate of the quantile is then returned.

Note: the algorithm used is the P-Squared algorithm invented in 1985 by Raj Jain and Imrich Chlamtac, which avoids accumulating and sorting the entire data set, while still obtaining good quality estimates of the quantile. The algorithm requires an initial seed of five samples. Then additional samples input into the algorithm produce quantile estimates. To eliminate this special case from the abstract interface, the **TXR** implementation is capable of producing an estimate when five or fewer samples have been presented, including none. In this low situation, the *`p`* value is ignored in reporting the estimate. When no samples have been given, the estimate is zero. When one sample has been given, the estimate is that sample itself. When between two and five samples have been given, the estimate is their median. Using the median as the estimate ensures a smooth transition from these early estimates into the estimates produced by the P-Squared algorithm. This is because the P-Squared algorithm always reports the value of the middle height

accumulator as the estimate, and that accumulator's initial value is the median of the first five samples.

The function returned by `quantile`, though not accumulating all of the samples passed to it, nevertheless has a limited sample capacity, because the registers it uses for tracking the sample group positions are fixed-width integers. The sample capacity is approximately 4 times the value of `fixnum-max`.

Example:

```
(defparm q (quantile 0.9))   ;; create 90-th percentile accumulator

[q] -> 0.0                   ;; no samples given: estimate is 0.
[q 3.14] -> 3.14             ;; one sample: estimate is that sample
[q 13.3 7.9 5.2 6.3] -> 7.9  ;; five samples: estimate is median.
[q 6.8 7.3 9.1 4.0]          ;; more than five samples; estimate now
-> 8.44651234567901          ;; from P-Square algorithm
[q #(13.1 5 2.5)]            ;; vector argument
-> 9.68660493827161
[q] -> 9.68660493827161      ;; no arguments: repeat current estimate
```

If the *`group-size`* argument is specified, then the quantile accumulator operates in grouped mode. Grouped mode allows infinite sample operation without overflow: an unlimited number of samples can be accepted. However, old samples lose their influence over the estimated value: newer samples are considered more significant than old samples.

In grouped mode, the quantile accumulator is reset to its initial state whenever *`group-size`* samples have been accumulated, and begins freshly calculating the quantile. Prior to the reset, an estimate is obtained and retained in an internal register. Going forward, this remembered previous estimate is blended in with the newly calculated estimate values, as described below. The cycle repeats itself whenever *`group-size`* samples accumulate: the state is reset, and the current estimate is loaded into the previous estimate register, which is then blended with newly computed values.

The *`rate`* parameter, whose default value is 0.999, controls the estimate blending. It should be a value between 0 and 1.

Upon each reset, a blend value register is initialized to 1.0. Each time a new sample is accumulated, the blend register is multiplied by the rate parameter, and the product is stored back into the blend register. Thus if the rate is between 0 and 1, exclusive, then the blend register exponentially decreases as the number of samples grows. The blend register indicates the fraction of the estimate which comes from the remembered previous estimate.

For instance, if the current blend value is 0.8, then the returned estimate value is 0.8 times the remembered previous estimate, plus 0.2 times the newly computed estimate for the current sample in the new group: the previous and current estimate are blended 80:20.

The default *`rate`* value of 0.999 is chosen for a slow transition to the new estimates, which helps to conceal inaccuracies in the algorithm associated with having accumulated a small number of samples. At this rate, it requires about 290 samples before the blend value drops to 75% of the old estimate.

If `rate` is specified as 0, then no blending of the previous estimate value takes place, since the blend factor will drop to zero upon the first sample being received after the group reset, causing the newly calculated estimates to be returned without blending. The previous sample groups

therefore have no influence over newer estimates. If `rate` is specified as 1, then the blend factor will stay at 1, and so the estimate will forever remain at the previous value, ignoring the calculations driven by the new samples.

Note: it is recommended that if *`group-size`* is specified, the value should be at least several hundred. Too small a group size will prevent the estimation algorithm from settling on good results. The *`rate`* parameter should not much smaller than 1. A rate too low will cause the previous estimate's contribution to the quantile value to diminish, too quickly, before the new estimation settles.

9.34.46 Variables `flo-near`, `flo-down`, `flo-up` and `flo-zero`

Description:

These variables hold integer values suitable as arguments to the `flo-set-round-mode` function, which controls the rounding mode for the results of floating-point operations. These variables are only defined on platforms which support rounding control.

Their values have the following meanings:

`flo-near`
: Round to nearest: the result of an operation is rounded to the nearest representable value.

`flo-down`
: Round down: the result of an operation is rounded to the nearest representable value that lies in the direction of negative infinity.

`flo-up`
: Round up: the result of an operation is rounded to the nearest representable value that lies in the direction of positive infinity.

`flo-zero`
: Round to zero: the result of an operation is rounded to the nearest representable value that lies in the direction of zero.

9.34.47 Functions `flo-get-round-mode` and `flo-set-round-mode`

Syntax:

```
(flo-get-round-mode)
(flo-set-round-mode mode)
```

Description:

Sometimes floating-point operations produce a result which requires more bits of precision than the floating point representation can provide. A representable floating-point value must be substituted for the true result and yielded by the operation.

On platforms which support rounding control, these functions are provided for selecting the decision procedure by which the floating-point representation is taken.

The `flo-get-round-mode` returns the current rounding mode. The rounding mode is represented by an integer value which is either equal to one of the four variables `flo-near`, `flo-down`, `flo-up` and `flo-zero`, or else some other value specific to the host environment. Initially, the value is that of `flo-near`. Otherwise, the value returned is that which was stored by the most recent successful call to `flo-set-round-mode`.

The `flo-set-round-mode` function changes the rounding mode. The argument to its *`mode`* parameter may be the value of one of the above four variables, or else some other value supported by the host environment's `fesetround` C library function.

The `flo-set-round-mode` function returns `t` if it is successful, otherwise the return value is `nil` and the rounding mode is not changed.

If a value is passed to `flo-set-round-mode` which is not the value of one of the above four rounding mode variables, and the function succeeds anyway, then the rounding behavior of floating-point operations depends on the host environment's interpretation of that value.

9.35 Bit Operations

In **TXR Lisp**, similarly to Common Lisp, bit operations on integers are based on a concept that might be called "infinite two's complement". Under infinite two's complement, a positive number is regarded as having a binary representation prefixed by an infinite stream of zero digits (for example `1` is `...00001`). A negative number in infinite two's complement is the bitwise negation of its positive counterpart, plus one: it carries an infinite prefix of 1 digits. So for instance the number `-1` is represented by `...11111111`: an infinite sequence of 1 bits. There is no specific sign bit; any operation which produces such an infinite sequence of 1 digits on the left gives rise to a negative number. For instance, consider the operation of computing the bitwise complement of the number `1`. Since the number `1` is represented as `...0000001`, its complement is `...11111110`. Each one of the `0` digits in the infinite sequence is replaced by `1`, And this leading sequence means that the number is negative, in fact corresponding to the two's complement representation of the value `-2`. Hence, the infinite digit concept corresponds to an arithmetic interpretation.

In fact **TXR Lisp**'s bignum integers do not use a two's complement representation internally. Numbers are represented as an array which holds a pure binary number. A separate field indicates the sign: negative, or nonnegative. That negative numbers appear as two's complement under the bit operations is merely a carefully maintained illusion (which makes bit operations on negative numbers more expensive).

The `logtrunc` function, as well as a feature of the `lognot` function, allow bit manipulation code to be written which works with positive numbers only, even if complements are required. The trade off is that the application has to manage a limit on the number of bits.

9.35.1 Functions `logand`, `logior` and `logxor`

Syntax:

```
(logand integer*)
(logior integer*)
(logxor int1 int2)
```

Description:

These operations perform the familiar bitwise and, inclusive or, and exclusive or operations, respectively. Positive values inputs are treated as pure binary numbers. Negative inputs are treated as infinite-bit two's complement.

For example `(logand -2 7)` produces `6`. This is because `-2` is `...111110` in infinite-bit two's complement. And-ing this value with `7` (or `...000111`) produces `110`.

The `logand` and `logior` functions are variadic, and may be called with zero, one, two, or more input values. If `logand` is called with no arguments, it produces the value -1 (all bits 1). If `logior` is called with no arguments it produces zero. In the one-argument case, the functions just return their argument value.

In the two-argument case, one of the operands may be a character, if the other operand is a fixnum integer. The character operand is taken to be an integer corresponding to the character value's Unicode code point value. The resulting value is regarded as a Unicode code point and converted to a character value accordingly.

When three or more arguments are specified, the operation's semantics is that of a left-associative reduction through two-argument invocations, so that the three-argument case `(logand a b c)` is equivalent to the expression `(logand (logand a b) c)`, which features two two-argument cases.

9.35.2 Function `logtest`

Syntax:

```
(logtest int1 int2)
```

Description:

The `logtest` function returns true if *int1* and *int2* have bits in common. The following equivalence holds:

```
(logtest a b) <--> (not (zerop (logand a b)))
```

9.35.3 Functions `lognot` and `logtrunc`

Syntax:

```
(lognot value [bits])
(logtrunc value bits)
```

Description:

The `lognot` function performs a bitwise complement of *value*. When the one-argument form of lognot is used, then if *value* is nonnegative, then the result is negative, and vice versa, according to the infinite-bit two's complement representation. For instance `(lognot -2)` is `1`, and `(lognot 1)` is `-2`.

The two-argument form of `lognot` produces a truncated complement. Conceptually, a bitwise complement is first calculated, and then the resulting number is truncated to the number of bits given by *bits*, which must be a nonnegative integer. The following equivalence holds:

```
(lognot a b) <--> (logtrunc (lognot a) b)
```

The `logtrunc` function truncates the integer *value* to the specified number of bits. If *value* is negative, then the two's complement representation is truncated. The return value of `logtrunc` is always a nonnegative integer.

9.35.4 Function `sign-extend`

Syntax:

```
(sign-extend value bits)
```

Description:

The `sign-extend` function first truncates the infinite-bit two's complement representation of the integer *value* to the specified number of bits, similarly to the `logtrunc` function. Then, this truncated value is regarded as a *bits*-wide two's complement integer. The value of this integer is calculated and returned.

Examples:

```
(sign-extend 127 8) -> 127
(sign-extend 128 8) -> -128
(sign-extend 129 8) -> -127
```

```
(sign-extend 255 8) -> -1
(sign-extend 256 8) -> 0
(sign-extend -1  8) -> -1
(sign-extend -255 8) -> 0
```

9.35.5 Function `ash`

Syntax:

```
(ash value bits)
```

Description:

The `ash` function shifts *`value`* by the specified number of *`bits`* producing a new value. If *`bits`* is positive, then a left shift takes place. If *`bits`* is negative, then a right shift takes place. If *`bits`* is zero, then *`value`* is returned unaltered. For positive numbers, a left shift by n bits is equivalent to a multiplication by two to the power of n, or `(expt 2 n)`. A right shift by n bits of a positive integer is equivalent to integer division by `(expt 2 n)`, with truncation toward zero. For negative numbers, the bit shift is performed as if on the two's complement representation. Under the infinite two's complement representation, a right shift does not exhaust the infinite sequence of `1` digits which extends to the left. Thus if `-4` is shifted right it becomes `-2` because the bitwise representations of these values are `...111100` and `...11110`.

9.35.6 Function `bit`

Syntax:

```
(bit value bit)
```

Description:

The `bit` function tests whether the integer or character *`value`* has a 1 in bit position *`bit`*. The *`bit`* argument must be a nonnegative integer. A value of *`bit`* of zero indicates the least-significant-bit position of *`value`*.

The `bit` function has a Boolean result, returning the symbol `t` if bit *`bit`* of *`value`* is set, otherwise `nil`.

If *`value`* is negative, it is treated as if it had an infinite-bit two's complement representation. For instance, if *`value`* is `-2`, then the `bit` function returns `nil` for a *`bit`* value of zero, and `t` for all other values, since the infinite bit two's complement representation of `-2` is `...11110`.

9.35.7 Function `mask`

Syntax:

```
(mask integer*)
```

Description:

The `mask` function takes zero or more integer arguments, and produces an integer value which corresponds to a bitmask made up of the bit positions specified by the integer arguments.

If `mask` is called with no arguments, then the return value is zero.

If `mask` is called with a single argument *`integer`* then the return value is the same as that of the expression `(ash 1 <integer>)`: the value 1 shifted left by *`integer`* bit positions. If *`integer`* is zero, then the result is `1`; if *`integer`* is `1`, the result is `2` and so forth. If *`value`* is negative, then the result is zero.

If `mask` is called with two or more arguments, then the result is a bitwise or of the masks individually computed for each of the values.

In other words, the following equivalences hold:

```
(mask) <--> 0
(mask a) <--> (ash 1 a)
(mask a b c ...) <--> (logior (mask a) (mask b) (mask c) ...)
```

9.35.8 Function `bitset`

Syntax:

```
(bitset integer)
```

Description:

The `bitset` function returns a list of the positions of bits which have a value of 1 in a positive *`integer`* argument, or the positions of bits which have a value of zero in a negative *`integer`* argument. The positions are ordered from least to greatest. The least significant bit has position zero. If *`integer`* is zero, the empty list `nil` is returned.

A negative integer is treated as an infinite-bit two's complement representation.

The argument may be a character.

If *`integer`* `x` is nonnegative, the following equivalence holds:

```
x  <-->  [apply mask (bitset x)]
```

That is to say, the value of `x` may be reconstituted by applying the bit positions returned by `bitset` as arguments to the `mask` function.

The value of a negative `x` may be reconstituted from its `bitset` as follows:

```
x  <-->  (pred (- [apply mask (bitset x)]))
```

also, more trivially, thus:

```
x  <-->  (- [apply mask (bitset (- x))])
```

9.35.9 Function `width`

Syntax:

```
(width integer*)
```

Description:

A two's complement representation of an integer consists of a sign bit and a mantissa field. The `width` function computes the minimum number of bits required for the mantissa portion of the two's complement representation of the *`integer`* argument.

For a nonnegative argument, the width also corresponds to the number of bits required for a natural binary representation of that value.

Two integer values have a width of zero, namely 0 and -1. This means that these two values can be represented in a one-bit two's complement, consisting of only a sign bit: the one-bit two's

complement bitfield 1 denotes -1, and 0 denotes 0.

Similarly, two integer values have a width of 1: 1 and -2. The two-bit two's complement bitfield 01 denotes 1, and 10 denotes -2.

The argument may be a character.

9.35.10 Function `logcount`

Syntax:

```
(logcount integer)
```

Description:

The `logcount` function considers *`integer`* to have a two's complement representation. If the integer is positive, it returns the count of bits in that representation whose value is 1. If *`integer`* is negative, it returns the count of zero bits instead. If *`integer`* is zero, the value returned is zero.

The argument may be a character.

9.35.11 Macros `set-mask` and `clear-mask`

Syntax:

```
(set-mask place integer*)
(clear-mask place integer*)
```

Description:

The `set-mask` and `clear-mask` macros set to 1 and 0, respectively, the bits in *`place`* corresponding to bits that are equal to 1 in the mask resulting from applying the inclusive or operation to the *`integer`* arguments. The following equivalences hold:

```
(set-mask place integer ...)
<-->  (set place (logior place integer ...)

(clear-mask place integer ...)
<-->  (set place (logand place (lognot (logior integer ...))))
```

9.36 User-Defined Arithmetic Types

TXR Lisp makes it possible for the user application program to define structure types which can participate in arithmetic operations as if they were numbers. Under most arithmetic functions, a structure object may be used instead of a number, if that structure object implements a specific method which is required by that arithmetic function.

The following paragraphs give general remarks about the method conventions. Not all arithmetic and bit manipulation functions have a corresponding method, and a small number of functions do not follow these conventions.

In the simplest case of arithmetic functions which are unary, the method takes no argument other than the object itself. Most unary arithmetic functions expect a structure argument to have a method which has the same name as that function. For instance, if `x` is a structure, then `(cos x)` will invoke `x.(cos)`. If `x` has no `cos` method, then an `error` exception is thrown. A few unary methods are not named after the corresponding function. The unary case of the `-` function expects an object to have a method named `neg`; thus, `(- x)` invokes `x.(neg)`. Unary division requires a method called `recip`; thus, `(/ x)`, invokes `x.(recip)`.

When a structure object is used as an argument in a two-argument (binary) arithmetic function, there are several cases to consider. If the left argument to a binary function is an object, then that object is expected to support a binary method. That method is called with two arguments: the object itself, of course, and the right argument of the arithmetic operation. In this case, the method is named after the function. For instance, if `x` is an object, then `(+ x 3)` invokes `x.(+ 3)`. If the right argument, and only the right argument, of a binary operation is an object, then the situation falls into two cases depending on whether the operation is commutative. If the operation is commutative, then the same method is used as in the case when the object is the left argument. The arguments are merely reversed. Thus `(+ 3 x)` also invokes `x.(+ 3)`. If the operation is not commutative, then the object must supply an alternative method. For most functions, that method is named by a symbol whose name begins with a `r-` prefix. For instance `(mod x 5)` invokes `x.(mod 5)` whereas `(mod 5 x)` invokes `x.(r-mod 5)`. Note: the "r" may be remembered as indicating that the object is the **right** argument of the binary operation or that the arguments are **reversed**. Two functions do not follow the `r-` convention. These are `-` and `/`. For these, the methods used for the object as a right argument, respectively, are `--` and `//`. Thus `(/ 5 x)` invokes `x.(// 5)` and `(- 5 x)` invokes `x.(-- 5)`. Several binary functions do not support an object as the right argument. These are `sign-extend`, `ash` and `bit`.

Variadic arithmetic functions, when given three or more arguments, are regarded as performing a left-associative decimation of the arguments through a binary function. Thus for instance `(- 1 x 4)` is understood as `(- (- 1 x) 4)` where `x.(-- 1)` is evaluated first. If that method yields an object `o` then `o.(- 4)` is invoked.

Certain variadic arithmetic functions, if invoked with one argument, just return that argument: for instance, `+` and `*` are in this category. A special concession exists in these functions: if their one and only argument is a structure, then that structure is returned without any error checking, even if it implements no methods related to arithmetic.

The following sections describe each of the methods that must be implemented by an object for the associated arithmetic function to work with that object, either at all, or in a specific argument position, as the case may be. These methods are not provided by **TXR Lisp**; the application is required to provide them.

9.36.1 Method +

Syntax:

obj`.(+` *arg*`)`

Description:

The + method is invoked when a structure is used as an argument to the + function together with at least one other operand.

If an object *obj* is combined with an argument *arg*, either as `(+` *obj arg*`)` or as `(+` *arg obj*`)` then, effectively, the method call *obj*`.(+` *arg*`)` takes place, and its return value is taken as the result of the operation.

9.36.2 Method –

Syntax:

```
obj.(- arg)
```

Description:

The − method is invoked when the structure *obj* is used as the left argument of the − function.

If an object *obj* is combined with an argument *arg*, as `(- obj arg)` then, effectively, the method call `obj.(- arg)` takes place, and its return value is taken as the result of the operation.

9.36.3 Method `--`

Syntax:

```
obj.(-- arg)
```

Description:

The `--` method is invoked when the structure *obj* is used as the right argument of the − function.

If an object *obj* is combined with an argument *arg*, as `(- arg obj)` then, effectively, the method call `obj.(-- arg)` takes place, and its return value is taken as the result of the operation.

9.36.4 Method `neg`

Syntax:

```
obj.(neg)
```

Description:

The `neg` method is invoked when the structure *obj* is used as the sole argument to the − function.

If an object *obj* is passed to the function as `(- obj)` then, effectively, the method call `obj.(neg)` takes place, and its return value is taken as the result of the operation.

9.36.5 Method `*`

Syntax:

```
obj.(* arg)
```

Description:

The `*` method is invoked when a structure is used as an argument to the `*` function together with at least one other operand.

If an object *obj* is combined with an argument *arg*, either as `(* obj arg)` or as `(* arg obj)` then, effectively, the method call `obj.(* arg)` takes place, and its return value is taken as the result of the operation.

9.36.6 Method `/`

Syntax:

```
obj.(/ arg)
```

Description:

The `/` method is invoked when the structure *obj* is used as the left argument of the `/` function.

If an object *obj* is combined with an argument *arg*, as `(/ obj arg)` then, effectively, the method call `obj.(/ arg)` takes place, and its return value is taken as the result of the operation.

9.36.7 Method `//`

Syntax:

```
obj.(// arg)
```

Description:

The `//` method is invoked when the structure *obj* is used as the right argument of the `/` function.

If an object *obj* is combined with an argument *arg*, as `(/ arg obj)` then, effectively, the method call `obj.(// arg)` takes place, and its return value is taken as the result of the operation.

9.36.8 Method `recip`

Syntax:

```
obj.(recip)
```

Description:

The `recip` method is invoked when the structure *obj* is used as the sole argument to the `/` function.

If an object *obj* is passed to the function as `(/ obj)` then, effectively, the method call `obj.(recip)` takes place, and its return value is taken as the result of the operation.

9.36.9 Method `abs`

Syntax:

```
obj.(abs)
```

Description:

The `abs` method is invoked when a structure is used as the argument to the `abs` function.

If an object *obj* is passed to the function as `(abs obj)` then, effectively, the method call `obj.(abs)` takes place, and its return value is taken as the result of the operation.

9.36.10 Method `signum`

Syntax:

```
obj.(signum)
```

Description:

The `signum` method is invoked when a structure is used as the argument to the `signum` function.

If an object *obj* is passed to the function as `(signum obj)` then, effectively, the method call `obj.(signum)` takes place, and its return value is taken as the result of the operation.

9.36.11 Method `trunc`

Syntax:

```
obj.(trunc arg)
```

Description:

The `trunc` method is invoked when the structure *obj* is used as the left argument of the `trunc` function.

If an object *obj* is combined with an argument *arg*, as `(trunc obj arg)` then, effectively, the method call `obj.(trunc arg)` takes place, and its return value is taken as the result of the

operation.

9.36.12 Method `r-trunc`

Syntax:

```
obj.(r-trunc arg)
```

Description:

The `r-trunc` method is invoked when the structure *obj* is used as the right argument of the `trunc` function.

If an object *obj* is combined with an argument *arg*, as `(trunc arg obj)` then, effectively, the method call *obj*`.(r-trunc` *arg*`)` takes place, and its return value is taken as the result of the operation.

9.36.13 Method `trunc1`

Syntax:

```
obj.(trunc1)
```

Description:

The `trunc1` method is invoked when the structure *obj* is used as the sole argument to the `trunc` function.

If an object *obj* is passed to the function as `(trunc obj)` then, effectively, the method call *obj*`.(trunc1)` takes place, and its return value is taken as the result of the operation.

9.36.14 Method `mod`

Syntax:

```
obj.(mod arg)
```

Description:

The `mod` method is invoked when the structure *obj* is used as the left argument of the `mod` function.

If an object *obj* is combined with an argument *arg*, as `(mod obj arg)` then, effectively, the method call *obj*`.(mod` *arg*`)` takes place, and its return value is taken as the result of the operation.

9.36.15 Method `r-mod`

Syntax:

```
obj.(r-mod arg)
```

Description:

The `r-mod` method is invoked when the structure *obj* is used as the right argument of the `mod` function.

If an object *obj* is combined with an argument *arg*, as `(mod arg obj)` then, effectively, the method call *obj*`.(r-mod` *arg*`)` takes place, and its return value is taken as the result of the operation.

9.36.16 Method `expt`

Syntax:

```
obj.(expt arg)
```

Description:

The `expt` method is invoked when the structure *obj* is used as the left argument of the `expt` function.

If an object *obj* is combined with an argument *arg*, as `(expt obj arg)` then, effectively, the method call *obj*`.(expt arg)` takes place, and its return value is taken as the result of the operation.

9.36.17 Method `r-expt`

Syntax:

obj`.(r-expt arg)`

Description:

The `r-expt` method is invoked when the structure *obj* is used as the right argument of the `expt` function.

If an object *obj* is combined with an argument *arg*, as `(expt arg obj)` then, effectively, the method call *obj*`.(r-expt arg)` takes place, and its return value is taken as the result of the operation.

9.36.18 Method `exptmod`

Syntax:

obj`.(exptmod arg1 arg2)`

Description:

The `exptmod` method is invoked when the structure *obj* is used as the left argument of the `exptmod` function.

If an object *obj* is combined with arguments *arg1* and *arg2*, as `(exptmod obj arg1 arg2)` then, effectively, the method call *obj*`.(exptmod arg1 arg2)` takes place, and its return value is taken as the result of the operation.

Note: the `exptmod` function doesn't support structure objects in the second and third argument positions. The *exponent* and *base* arguments must be integers.

9.36.19 Method `isqrt`

Syntax:

obj`.(isqrt)`

Description:

The `isqrt` method is invoked when a structure is used as the argument to the `isqrt` function.

If an object *obj* is passed to the function as `(isqrt obj)` then, effectively, the method call *obj*`.(isqrt)` takes place, and its return value is taken as the result of the operation.

9.36.20 Method `square`

Syntax:

obj`.(square)`

Description:

The `square` method is invoked when a structure is used as the argument to the `square` function.

If an object *obj* is passed to the function as `(square obj)` then, effectively, the method call `obj.(square)` takes place, and its return value is taken as the result of the operation.

9.36.21 Method >

Syntax:

```
obj.(> arg)
```

Description:

The > method is invoked when the > function is invoked with two operands, and the structure *obj* is the left operand. The method is also invoked when the < function is invoked with two operands, and *obj* is the right operand.

If an object *obj* is combined with an argument *arg*, either as `(> obj arg)` or as `(< arg obj)` then, effectively, the method call `obj.(> arg)` takes place, and its return value is taken as the result of the operation.

9.36.22 Method <

Syntax:

```
obj.(< arg)
```

Description:

The < method is invoked when the < function is invoked with two operands, and the structure *obj* is the left operand. The method is also invoked when the > function is invoked with two operands, and *obj* is the right operand.

If an object *obj* is combined with an argument *arg*, either as `(< obj arg)` or as `(> arg obj)` then, effectively, the method call `obj.(< arg)` takes place, and its return value is taken as the result of the operation.

9.36.23 Method >=

Syntax:

```
obj.(>= arg)
```

Description:

The >= method is invoked when the >= function is invoked with two operands, and the structure *obj* is the left operand. The method is also invoked when the <= function is invoked with two operands, and *obj* is the right operand.

If an object *obj* is combined with an argument *arg*, either as `(>= obj arg)` or as `(<= arg obj)` then, effectively, the method call `obj.(>= arg)` takes place, and its return value is taken as the result of the operation.

9.36.24 Method <=

Syntax:

```
obj.(<= arg)
```

Description:

The <= method is invoked when the <= function is invoked with two operands, and the structure *obj* is the left operand. The method is also invoked when the >= function is invoked with two operands, and *obj* is the right operand.

If an object *obj* is combined with an argument *arg*, either as `(<= obj arg)` or as `(>= arg obj)` then, effectively, the method call `obj.(<= arg)` takes place, and its return value is

taken as the result of the operation.

9.36.25 Method =

Syntax:

```
obj.(= arg)
```

Description:

The = method is invoked when a structure is used as an argument to the = function.

If an object *obj* is combined with an argument *arg*, either as `(= obj arg)` or as `(= arg obj)` then, effectively, the method call `obj.(= arg)` takes place, and its return value is taken as the result of the operation.

9.36.26 Method `zerop`

Syntax:

```
obj.(zerop)
```

Description:

The `zerop` method is invoked when a structure is used as the argument to the `zerop` function.

If an object *obj* is passed to the function as `(zerop obj)` then, effectively, the method call `obj.(zerop)` takes place, and its return value is taken as the result of the operation.

9.36.27 Method `plusp`

Syntax:

```
obj.(plusp)
```

Description:

The `plusp` method is invoked when a structure is used as the argument to the `plusp` function.

If an object *obj* is passed to the function as `(plusp obj)` then, effectively, the method call `obj.(plusp)` takes place, and its return value is taken as the result of the operation.

9.36.28 Method `minusp`

Syntax:

```
obj.(minusp)
```

Description:

The `minusp` method is invoked when a structure is used as the argument to the `minusp` function.

If an object *obj* is passed to the function as `(minusp obj)` then, effectively, the method call `obj.(minusp)` takes place, and its return value is taken as the result of the operation.

9.36.29 Method `evenp`

Syntax:

```
obj.(evenp)
```

Description:

The `evenp` method is invoked when a structure is used as the argument to the `evenp` function.

If an object *obj* is passed to the function as `(evenp obj)` then, effectively, the method call `obj.(evenp)` takes place, and its return value is taken as the result of the operation.

9.36.30 Method `oddp`

Syntax:

```
obj.(oddp)
```

Description:

The `oddp` method is invoked when a structure is used as the argument to the `oddp` function.

If an object *obj* is passed to the function as `(oddp obj)` then, effectively, the method call `obj.(oddp)` takes place, and its return value is taken as the result of the operation.

9.36.31 Method `floor`

Syntax:

```
obj.(floor arg)
```

Description:

The `floor` method is invoked when the structure *obj* is used as the left argument of the `floor` function.

If an object *obj* is combined with an argument *arg*, as `(floor obj arg)` then, effectively, the method call `obj.(floor arg)` takes place, and its return value is taken as the result of the operation.

9.36.32 Method `r-floor`

Syntax:

```
obj.(r-floor arg)
```

Description:

The `r-floor` method is invoked when the structure *obj* is used as the right argument of the `floor` function.

If an object *obj* is combined with an argument *arg*, as `(floor arg obj)` then, effectively, the method call `obj.(r-floor arg)` takes place, and its return value is taken as the result of the operation.

9.36.33 Method `floor1`

Syntax:

```
obj.(floor1)
```

Description:

The `floor1` method is invoked when the structure *obj* is used as the sole argument to the `floor` function.

If an object *obj* is passed to the function as `(floor obj)` then, effectively, the method call `obj.(floor1)` takes place, and its return value is taken as the result of the operation.

9.36.34 Method `ceil`

Syntax:

```
obj.(ceil arg)
```

Description:

The `ceil` method is invoked when the structure *obj* is used as the left argument of the `ceil` function.

If an object *obj* is combined with an argument *arg*, as `(ceil obj arg)` then, effectively, the method call *obj*`.(ceil arg)` takes place, and its return value is taken as the result of the operation.

9.36.35 Method `r-ceil`

Syntax:

obj`.(r-ceil arg)`

Description:

The `r-ceil` method is invoked when the structure *obj* is used as the right argument of the `ceil` function.

If an object *obj* is combined with an argument *arg*, as `(ceil arg obj)` then, effectively, the method call *obj*`.(r-ceil arg)` takes place, and its return value is taken as the result of the operation.

9.36.36 Method `ceil1`

Syntax:

obj`.(ceil1)`

Description:

The `ceil1` method is invoked when the structure *obj* is used as the sole argument to the `ceil` function.

If an object *obj* is passed to the function as `(ceil obj)` then, effectively, the method call *obj*`.(ceil1)` takes place, and its return value is taken as the result of the operation.

9.36.37 Method `round`

Syntax:

obj`.(round arg)`

Description:

The `round` method is invoked when the structure *obj* is used as the left argument of the `round` function.

If an object *obj* is combined with an argument *arg*, as `(round obj arg)` then, effectively, the method call *obj*`.(round arg)` takes place, and its return value is taken as the result of the operation.

9.36.38 Method `r-round`

Syntax:

obj`.(r-round arg)`

Description:

The `r-round` method is invoked when the structure *obj* is used as the right argument of the `round` function.

If an object *obj* is combined with an argument *arg*, as `(round arg obj)` then, effectively, the method call *obj*`.(r-round arg)` takes place, and its return value is taken as the result of the operation.

9.36.39 Method `round1`

Syntax:

```
obj.(round1)
```

Description:

The `round1` method is invoked when the structure *obj* is used as the sole argument to the `round` function.

If an object *obj* is passed to the function as `(round obj)` then, effectively, the method call `obj.(round1)` takes place, and its return value is taken as the result of the operation.

9.36.40 Method `sin`

Syntax:

```
obj.(sin)
```

Description:

The `sin` method is invoked when a structure is used as the argument to the `sin` function.

If an object *obj* is passed to the function as `(sin obj)` then, effectively, the method call `obj.(sin)` takes place, and its return value is taken as the result of the operation.

9.36.41 Method `cos`

Syntax:

```
obj.(cos)
```

Description:

The `cos` method is invoked when a structure is used as the argument to the `cos` function.

If an object *obj* is passed to the function as `(cos obj)` then, effectively, the method call `obj.(cos)` takes place, and its return value is taken as the result of the operation.

9.36.42 Method `tan`

Syntax:

```
obj.(tan)
```

Description:

The `tan` method is invoked when a structure is used as the argument to the `tan` function.

If an object *obj* is passed to the function as `(tan obj)` then, effectively, the method call `obj.(tan)` takes place, and its return value is taken as the result of the operation.

9.36.43 Method `asin`

Syntax:

```
obj.(asin)
```

Description:

The `asin` method is invoked when a structure is used as the argument to the `asin` function.

If an object *obj* is passed to the function as `(asin obj)` then, effectively, the method call `obj.(asin)` takes place, and its return value is taken as the result of the operation.

9.36.44 Method `acos`

Syntax:

obj.(acos)

Description:

The acos method is invoked when a structure is used as the argument to the acos function.

If an object *obj* is passed to the function as (acos *obj*) then, effectively, the method call *obj*.(acos) takes place, and its return value is taken as the result of the operation.

9.36.45 Method atan

Syntax:

obj.(atan)

Description:

The atan method is invoked when a structure is used as the argument to the atan function.

If an object *obj* is passed to the function as (atan *obj*) then, effectively, the method call *obj*.(atan) takes place, and its return value is taken as the result of the operation.

9.36.46 Method atan2

Syntax:

obj.(atan2 *arg*)

Description:

The atan2 method is invoked when the structure *obj* is used as the left argument of the atan2 function.

If an object *obj* is combined with an argument *arg*, as (atan2 *obj arg*) then, effectively, the method call *obj*.(atan2 *arg*) takes place, and its return value is taken as the result of the operation.

9.36.47 Method r-atan2

Syntax:

obj.(r-atan2 *arg*)

Description:

The r-atan2 method is invoked when the structure *obj* is used as the right argument of the atan2 function.

If an object *obj* is combined with an argument *arg*, as (atan2 *arg obj*) then, effectively, the method call *obj*.(r-atan2 *arg*) takes place, and its return value is taken as the result of the operation.

9.36.48 Method sinh

Syntax:

obj.(sinh)

Description:

The sinh method is invoked when a structure is used as the argument to the sinh function.

If an object *obj* is passed to the function as (sinh *obj*) then, effectively, the method call *obj*.(sinh) takes place, and its return value is taken as the result of the operation.

9.36.49 Method `cosh`

Syntax:

```
obj.(cosh)
```

Description:

The `cosh` method is invoked when a structure is used as the argument to the `cosh` function.

If an object *obj* is passed to the function as `(cosh obj)` then, effectively, the method call `obj.(cosh)` takes place, and its return value is taken as the result of the operation.

9.36.50 Method `tanh`

Syntax:

```
obj.(tanh)
```

Description:

The `tanh` method is invoked when a structure is used as the argument to the `tanh` function.

If an object *obj* is passed to the function as `(tanh obj)` then, effectively, the method call `obj.(tanh)` takes place, and its return value is taken as the result of the operation.

9.36.51 Method `asinh`

Syntax:

```
obj.(asinh)
```

Description:

The `asinh` method is invoked when a structure is used as the argument to the `asinh` function.

If an object *obj* is passed to the function as `(asinh obj)` then, effectively, the method call `obj.(asinh)` takes place, and its return value is taken as the result of the operation.

9.36.52 Method `acosh`

Syntax:

```
obj.(acosh)
```

Description:

The `acosh` method is invoked when a structure is used as the argument to the `acosh` function.

If an object *obj* is passed to the function as `(acosh obj)` then, effectively, the method call `obj.(acosh)` takes place, and its return value is taken as the result of the operation.

9.36.53 Method `atanh`

Syntax:

```
obj.(atanh)
```

Description:

The `atanh` method is invoked when a structure is used as the argument to the `atanh` function.

If an object *obj* is passed to the function as `(atanh obj)` then, effectively, the method call `obj.(atanh)` takes place, and its return value is taken as the result of the operation.

9.36.54 Method `log`

Syntax:

```
obj.(log)
```

Description:

The `log` method is invoked when a structure is used as the argument to the `log` function.

If an object *obj* is passed to the function as `(log obj)` then, effectively, the method call `obj.(log)` takes place, and its return value is taken as the result of the operation.

9.36.55 Method `log2`

Syntax:

```
obj.(log2)
```

Description:

The `log2` method is invoked when a structure is used as the argument to the `log2` function.

If an object *obj* is passed to the function as `(log2 obj)` then, effectively, the method call `obj.(log2)` takes place, and its return value is taken as the result of the operation.

9.36.56 Method `log10`

Syntax:

```
obj.(log10)
```

Description:

The `log10` method is invoked when a structure is used as the argument to the `log10` function.

If an object *obj* is passed to the function as `(log10 obj)` then, effectively, the method call `obj.(log10)` takes place, and its return value is taken as the result of the operation.

9.36.57 Method `exp`

Syntax:

```
obj.(exp)
```

Description:

The `exp` method is invoked when a structure is used as the argument to the `exp` function.

If an object *obj* is passed to the function as `(exp obj)` then, effectively, the method call `obj.(exp)` takes place, and its return value is taken as the result of the operation.

9.36.58 Method `sqrt`

Syntax:

```
obj.(sqrt)
```

Description:

The `sqrt` method is invoked when a structure is used as the argument to the `sqrt` function.

If an object *obj* is passed to the function as `(sqrt obj)` then, effectively, the method call `obj.(sqrt)` takes place, and its return value is taken as the result of the operation.

9.36.59 Method `logand`

Syntax:

```
obj.(logand arg)
```

Description:

The `logand` method is invoked when a structure is used as an argument to the `logand` function together with at least one other operand.

If an object *obj* is combined with an argument *arg*, either as `(logand obj arg)` or as `(logand arg obj)` then, effectively, the method call *obj*`.(logand arg)` takes place, and its return value is taken as the result of the operation.

9.36.60 Method `logior`

Syntax:

obj`.(logior arg)`

Description:

The `logior` method is invoked when a structure is used as an argument to the `logior` function together with at least one other operand.

If an object *obj* is combined with an argument *arg*, either as `(logior obj arg)` or as `(logior arg obj)` then, effectively, the method call *obj*`.(logior arg)` takes place, and its return value is taken as the result of the operation.

9.36.61 Method `lognot`

Syntax:

obj`.(lognot arg)`

Description:

The `lognot` method is invoked when the structure *obj* is used as the left argument of the `lognot` function.

If an object *obj* is combined with an argument *arg*, as `(lognot obj arg)` then, effectively, the method call *obj*`.(lognot arg)` takes place, and its return value is taken as the result of the operation.

9.36.62 Method `r-lognot`

Syntax:

obj`.(r-lognot arg)`

Description:

The `r-lognot` method is invoked when the structure *obj* is used as the right argument of the `lognot` function.

If an object *obj* is combined with an argument *arg*, as `(lognot arg obj)` then, effectively, the method call *obj*`.(r-lognot arg)` takes place, and its return value is taken as the result of the operation.

9.36.63 Method `lognot1`

Syntax:

obj`.(lognot1)`

Description:

The `lognot1` method is invoked when the structure *obj* is used as the sole argument to the `lognot` function.

If an object *obj* is passed to the function as `(lognot obj)` then, effectively, the method call

obj.(lognot1) takes place, and its return value is taken as the result of the operation.

9.36.64 Method `logtrunc`

Syntax:

obj.(logtrunc *arg*)

Description:

The `logtrunc` method is invoked when the structure *obj* is used as the left argument of the `logtrunc` function.

If an object *obj* is combined with an argument *arg*, as (logtrunc *obj arg*) then, effectively, the method call *obj*.(logtrunc *arg*) takes place, and its return value is taken as the result of the operation.

9.36.65 Method `r-logtrunc`

Syntax:

obj.(r-logtrunc *arg*)

Description:

The `r-logtrunc` method is invoked when the structure *obj* is used as the right argument of the `logtrunc` function.

If an object *obj* is combined with an argument *arg*, as (logtrunc *arg obj*) then, effectively, the method call *obj*.(r-logtrunc *arg*) takes place, and its return value is taken as the result of the operation.

9.36.66 Method `sign-extend`

Syntax:

obj.(sign-extend *arg*)

Description:

The `sign-extend` method is invoked when the structure *obj* is used as the left argument of the `sign-extend` function.

If an object *obj* is combined with an argument *arg*, as (sign-extend *obj arg*) then, effectively, the method call *obj*.(sign-extend *arg*) takes place, and its return value is taken as the result of the operation.

Note: the `sign-extend` function doesn't support a structure as the right argument, *bits*, which must be an integer.

9.36.67 Method `ash`

Syntax:

obj.(ash *arg*)

Description:

The `ash` method is invoked when the structure *obj* is used as the left argument of the `ash` function.

If an object *obj* is combined with an argument *arg*, as (ash *obj arg*) then, effectively, the method call *obj*.(ash *arg*) takes place, and its return value is taken as the result of the operation.

Note: the `ash` function doesn't support a structure as the right argument, *`bits`*, which must be an integer.

9.36.68 Method `bit`

Syntax:

```
obj.(bit arg)
```

Description:

The `bit` method is invoked when the structure *`obj`* is used as the left argument of the `bit` function.

If an object *`obj`* is combined with an argument *`arg`*, as `(bit obj arg)` then, effectively, the method call *`obj`*`.(bit arg)` takes place, and its return value is taken as the result of the operation.

Note: the `bit` function doesn't support a structure as the right argument, *`bit`*, which must be an integer.

9.36.69 Method `width`

Syntax:

```
obj.(width)
```

Description:

The `width` method is invoked when a structure is used as the argument to the `width` function.

If an object *`obj`* is passed to the function as `(width obj)` then, effectively, the method call *`obj`*`.(width)` takes place, and its return value is taken as the result of the operation.

9.36.70 Method `logcount`

Syntax:

```
obj.(logcount)
```

Description:

The `logcount` method is invoked when a structure is used as the argument to the `logcount` function.

If an object *`obj`* is passed to the function as `(logcount obj)` then, effectively, the method call *`obj`*`.(logcount)` takes place, and its return value is taken as the result of the operation.

9.36.71 Method `bitset`

Syntax:

```
obj.(bitset)
```

Description:

The `bitset` method is invoked when a structure is used as the argument to the `bitset` function.

If an object *`obj`* is passed to the function as `(bitset obj)` then, effectively, the method call *`obj`*`.(bitset)` takes place, and its return value is taken as the result of the operation.

9.37 Exception Handling

An *exception* in **TXR** is a special event in the execution of the program which potentially results in a transfer of control. An exception is identified by a symbol, known as the *exception type*, and it carries zero or more arguments, called the *exception arguments*.

When an exception is initiated, it is said to be *thrown*. This action is initiated by the following functions: `throw`, `throwf` and `error`, and possibly other functions which invoke these. When an exception is thrown, **TXR** enters into exception processing mode. Exception processing mode terminates in one of several ways:

- A *catch* is found which matches the exception, and control is transferred to the catch by a nonlocal transfer which performs unwinding. Catches are defined by the `catch` macro.
- A *handler* is found which matches the exception, and control is transferred to the handler by invoking its function. The handler function accepts the exception by performing a nonlocal transfer to a destination of its choice, or else declines to accept the exception by returning. Handlers are defined by the `handler-bind` operator or `handle` macro.
- If no catch or accepting handler is found for an exception derived from `error` and `*unhandled-hook*` is `nil`, then a built-in strategy for handling the exception is invoked, consisting of unwinding, and then printing some informational messages and terminating. If the `*unhandled-hook*` variable contains a value that isn't `nil`, then control is transferred to the function stored in the that variable first; only if that function returns is the above built-in strategy invoked.
- If no catch or accepting handler is found for an exception derived from `warning`, then a warning diagnostic is issued on the `*stderr*` stream and a `continue` exception is thrown with no arguments. If no catch or handler is found for that exception, then control returns normally to the site which threw the warning exception.
- If no catch or accepting handler is found for an exception that is neither derived from `error` nor from `warning`, then no control transfer takes place; control returns to the `throw` or `throwf` function which returns normally, with a return value of `nil`.

9.37.1 Catches and Handlers

There are two ways by which exceptions are handled: catches and handlers. Catches and handlers are similar, but different. A catch is an exit point associated with an active scope. When an exception is handled by a catch, the form which threw the exception is abandoned, and unwinding takes place to the catch site, which receives the exception type and arguments. A handler is also associated with an active scope. However, it is a function, and not a dynamic exit point. When an exception is passed to handler, unwinding does not take place; rather, the function is called. The function then either completes the exception handling by performing a nonlocal transfer, or else declines the exception by performing an ordinary return.

Catches and handlers are identified by exception type symbols. A catch or handler is eligible to process an exception if it handles a type which is a supertype of the exception which is being processed. Handles and catches are found by means of a combined search which proceeds from the innermost nesting of dynamic scope to the outermost, without performing any unwinding. When an eligible handler is encountered, its registered function is called, thereby suspending the search. If the handler function returns, the search continues from that scope to yet unvisited outer scopes. When an eligible catch is encountered rather than a handler, the search terminates and a control transfer takes place to the catch site. That control transfer then performs unwinding, which requires it to make a second pass through the same nestings of dynamic scope that had just been traversed in order to find that catch.

9.37.2 Handlers and Sandboxing

Because handlers execute in the dynamic context of the exception origin, without any unwinding having taken place, they expose a potential route of sandbox escape via the package system, unless special steps are taken. The threat is that code at the handler site could take advantage of the current value of the

`*package*` and `*package-alist*` variables established at the exception throw site to gain inappropriate access to symbols.

For this reason, when a handler is established, the current values of `*package*` and `*package-alist*` are recorded into the handler frame. When that handler is later invoked, it executes in a dynamic environment in which those variables are bound to the previously noted values.

The catch mechanism doesn't do any such thing because the unwinding which is performed prior to the invocation of a catch implicitly restores the values of **all** special variables to the values they had at the time the frame was established.

9.37.3 Exception Type Hierarchy

Exception type symbols are arranged in an inheritance hierarchy, at whose top the symbol `t` is the supertype of every exception type, and the `nil` symbol is at the bottom, the subtype of every exception type.

Keyword symbols may be used as exception types.

Every symbol is its own supertype and subtype. Thus whenever X is known to be a subtype of Y, it is possible that X is exactly Y. The `defex` macro registers exception supertype/subtype relationships among symbols.

The following tree diagram shows the relationships among **TXR Lisp**'s built-in exception symbols. Not shown is the exception symbol `nil`, subtype of every exception type:

```
t ----+--- warning
      |
      +--- restart ---+--- continue
      |               |
      |               +--- retry
      |               |
      |               +--- skip
      |
      +--- error ---+--- type-error
                    |
                    +--- internal-error
                    |
                    +--- panic
                    |
                    +--- numeric-error
                    |
                    +--- range-error
                    |
                    +--- query-error
                    |
                    +--- file-error -------+--- path-not-found
                    |                      |
                    |                      +--- path-exists
                    |                      |
                    |                      +--- path-permission
                    |
                    +--- process-error
                    |
                    +--- socket-error
                    |
```

```
+--- system-error
|
+--- alloc-error
|
+--- timeout-error
|
+--- assert
|
+--- syntax-error
|
+--- eval-error
|
+--- match-error
|
+--- case-error
|
+--- opt-error
```

Program designers are encouraged to derive new error exceptions from the `error` type. The `restart` type is intended to be the root of a hierarchy of exception types used for denoting restart points: designers are encouraged to derive restarts from this type. A catch for the `continue` exception should be established around constructs which can throw an error from which it is possible to recover. That exception provides the entry point into the recovery which resumes execution. A catch for `retry` should be provided in situations when it is possible and makes sense for a failed operation to be tried again. A catch for `skip` should be provided in situations when it is possible and sensible to continue with subsequent operations even though an operation has failed.

9.37.4 Dialect Notes

Exception handling in **TXR Lisp** provides capabilities similar to the condition system in ANSI Common Lisp. The implementation and terminology differ.

Most obviously, ANSI CL uses the "condition" term, whereas **TXR Lisp** uses "exception".

In ANSI CL, a condition is "raised", whereas a **TXR Lisp** exception is "thrown".

In ANSI CL, when a condition is raised, a condition object is created. Condition object are similar to class objects, but are not required to be in the Common Lisp Object System. They are related by inheritance and can have properties. **TXR Lisp** exceptions are unencapsulated: they consist of a symbol, plus zero or more arguments. The symbols are related by inheritance.

When a condition is raised in ANSI CL, the dynamic scope is searched for a handler, which is an ordinary function which receives the condition. No unwinding or nonlocal transfer takes place. The handler can return, in which case the search continues. Matching the condition to the handler is by inheritance. Handler functions are bound to exception type names. If a handler chooses to actually handle a condition (thereby terminating the search) it must itself perform some kind of dynamic control transfer, rather than return normally. ANSI CL provides a dynamic control mechanism known as restarts which is usually used for this purpose. A condition handler may invoke a particular restart handler. Restart handlers are similar to exception handlers: they are functions associated with symbols in the dynamic environment.

In **TXR Lisp**, the special behavior which occurs for exceptions derived from `error` and those from `warning` is built into the exception handling system, and tied to those types. When an error or warning exception is unhandled, the exception handling system itself reacts, so the special behaviors occur no matter how these exceptions are raised. In ANSI CL, the special behavior for unhandled `error` conditions (of invoking the debugger) is implemented only in the `error` function; `error` conditions signalled other than

via that function are not subject to any special behavior. There is a parallel situation with regard to warnings: the ANSI CL `warn` function implements a special behavior for unhandled warnings (of emitting a diagnostic) but warnings not signalled via that function are not treated that way. Thus in **TXR Lisp**, there is no way to raise an error or warning that is simply ignored due to being unhandled.

In **TXR Lisp** exceptions are a unification of conditions and restarts. From an ANSI CL perspective, **TXR Lisp** exceptions are a lot like CL restarts, except that the symbols are arranged in an inheritance hierarchy. **TXR Lisp** exceptions are used both as the equivalent of ANSI CL conditions and as restarts.

In **TXR Lisp** the terminology "catch" and "handle" is used in a specific way. To handle an exception means to receive it without unwinding, with the possibility of declining to handle it, so that the search continues for another handler. To catch an exception means to match an exception to a catch handler, terminate the search, unwind and pass control to the handler.

TXR Lisp provides an operator called `handler-bind` for specifying handlers. It has a different syntax from ANSI CL's `handler-bind`. **TXR Lisp** provides a macro called `handle` which simplifies the use of `handler-bind`. This macro superficially resembles ANSI CL's `handler-case`, but is semantically different. The most notable difference is that the bodies of handlers established by `handler-bind` execute without any unwinding taking place and may return normally, thereby declining to take the exception. In other words, `handle` has the same semantics as `handler-bind`, providing only convenient syntax.

TXR Lisp provides a macro called `catch` which has the same syntax as `handle` but specifies a catch point for exceptions. If, during an exception search, a `catch` clause matches an exception, a dynamic control transfer takes place from the throw site to the catch site. Then the clause body is executed. The `catch` macro resembles ANSI CL's `restart-case` or possibly `handler-case`, depending on point of view.

TXR Lisp provides unified introspection over handler and catch frames. A program can programmatically discover what handler and catches are available in a given dynamic scope. ANSI CL provides introspection over restarts only; the standard doesn't specify any mechanism for inquiring what condition handlers are bound at a given point in the execution.

Example:

The following two examples express a similar approach implemented using ANSI Common Lisp conditions and restarts, and then using **TXR Lisp** exceptions.

```
;; Common Lisp
(define-condition foo-error (error)
  ((arg :initarg :arg :reader foo-error-arg)))

(defun raise-foo-error (arg)
  (restart-case
    (let ((c (make-condition 'foo-error :arg arg)))
      (error c))
    (recover (recover-arg)
      (format t "recover, arg: ~s~%" recover-arg))))

(handler-bind ((foo-error
                 (lambda (cond)
                   (format t "handling foo-error, arg: ~s~%"
                           (foo-error-arg cond))
                   (invoke-restart 'recover 100))))
  (raise-foo-error 200))
```

The output of the above is:

```
handling foo-error, arg: 200
recover, arg: 100
```

The following is possible **TXR Lisp** equivalent for the above Common Lisp example. It produces identical output.

```
(defex foo-error error)

(defex recover restart) ;; recommended practice

(defun raise-foo-error (arg)
  (catch
    (throw 'foo-error arg)
    (recover (recover-arg)
      (format t "recover, arg: ~s\n" recover-arg))))

(handle
  (raise-foo-error 200)
  (foo-error (arg)
    (format t "handling foo-error, arg: ~s\n" arg)
    (throw 'recover 100)))
```

To summarize the differences: exceptions serve as both conditions and restarts in **TXR**. The same `throw` function is used to initiate exception handling for `foo-error` and then to transfer control out of the handler to the recovery code. The handler accepts one exception by raising another.

When an exception symbol is used for restarting, it is a recommended practice to insert it into the inheritance hierarchy rooted at the `restart` symbol, either by inheriting directly from `restart` or from an exception subtype of that symbol.

9.37.5 Treatment of `errno` In Built-in Exceptions

Some **TXR Lisp** library functions generate exceptions in response to conditions arising in the operating system, and those conditions are associated with a numeric code in the POSIX/ISO C variable `errno`. This code isn't represented as an exception argument. Rather, in many of these situations, the `errno` value is attached to the error message string which is passed as the first and only exception argument. The value can be retrieved by using the function `string-get-code` on the error message string. If this function returns `nil`, then no such code is available in connection with the given error.

Example:

```
(catch
  (open-file "AsDf")
  (error (msg)
    ;; the value 2 is retrieved from msg
    ;; 2 is the common value of ENOENT
    (list (string-get-code msg) msg)))

-> (2 "error opening \"AsDf\": 2/\"No such file or directory\"")
```

9.37.6 Functions `throw`, `throwf` and `error`

Syntax:

```
(throw symbol arg*)
```

```
(throwf symbol format-string format-arg*)
(error format-string format-arg*)
```

Description:

These functions generate an exception. The `throw` and `throwf` functions generate an exception identified by *`symbol`*, whereas `error` throws an exception of type `error`. The call `(error ...)` can be regarded as a shorthand for `(throwf 'error ...)`.

The `throw` function takes zero or more additional arguments. These arguments become the arguments of a `catch` handler which takes the exception. The handler will have to be capable of accepting that number of arguments.

The `throwf` and `error` functions generate an exception which has a single argument: a character string created by a formatted print to a string stream using the `format` string and additional arguments.

Because `error` throws an error exception, it does not return. If an error exception is not handled, **TXR** will issue diagnostic messages and terminate. Likewise, `throw` or `throwf` are used to generate an error exception, they do not return.

If the `throw` and `throwf` functions are used to generate an exception not derived from `error`, and no handler is found which accepts the exception, they return normally, with a value of `nil`.

9.37.7 Macros `catch`, `catch*` and `catch**`

Syntax:

```
(catch try-expression
   {(symbol (arg*) body-form*)}*)
(catch* try-expression
   {(symbol (type-arg arg*) body-form*)}*)
(catch** try-expression
   {(symbol desc (type-arg arg*) body-form*)}*)
```

Description:

The `catch` macro establishes an exception catching block around the *`try-expression`*. The *`try-expression`* is followed by zero or more catch clauses. Each catch clause consists of a symbol which denotes an exception type, an argument list, and zero or more body forms.

If *`try-expression`* terminates normally, then the catch clauses are ignored. The catch itself terminates, and its return value is that of the *`try-expression`*.

If *`try-expression`* throws an exception which is a subtype of one or more of the type symbols given in the exception clauses, then the first (leftmost) such clause becomes the exit point where the exception is handled. The exception is converted into arguments for the clause, and the clause body is executed. When the clause body terminates, the catch terminates, and the return value of the catch is that of the clause body.

If *`try-expression`* throws an exception which is not a subtype of any of the symbols given in the clauses, then the search for an exit point for the exception continues through the enclosing forms. The catch clauses are not involved in the handling of that exception.

When a clause catches an exception, the number of arguments in the catch must match the number of elements in the exception. A catch argument list resembles a function or lambda argument list, and may be dotted. For instance the clause `(foo (a . b))` catches an exception subtyped

from `foo`, with one or more elements. The first element binds to parameter `a`, and the rest, if any, bind to parameter `b`. If there is only one element, `b` takes on the value `nil`.

The `catch*` macro is a variant of `catch` with the following difference: when `catch*` invokes a clause, it passes the exception symbol as the leftmost argument *`type-arg`*. Then the exception arguments follow. In contrast, only the exception arguments are passed to the clauses of `catch`.

The `catch**` macro is a further variant, which differs from `catch*` by requiring each catch clause to provide a description *`desc`*, an expression which evaluates to a character string. The *`desc`* expressions are evaluated in left-to-right order prior to the evaluation of *`try-expression`*.

Also see: the `unwind-protect` operator, and the functions `throw`, `throwf` and `error`, as well as the `handler-bind` operator and `handle` macro.

9.37.8 Operator `unwind-protect`

Syntax:

```
(unwind-protect protected-form cleanup-form*)
```

Description:

The `unwind-protect` operator evaluates *`protected-form`* in such a way that no matter how the execution of *`protected-form`* terminates, the *`cleanup-form`*s will be executed.

The *`cleanup-form`*s, however, are not protected. If a *`cleanup-form`* terminates via some nonlocal jump, the subsequent *`cleanup-form`*s are not evaluated.

*`cleanup-form`*s themselves can "hijack" a nonlocal control transfer such as an exception. If a *`cleanup-form`* is evaluated during the processing of a dynamic control transfer such as an exception, and that *`cleanup-form`* initiates its own dynamic control transfer, the original control transfer is aborted and replaced with the new one.

The exit points for dynamic control transfers are removed as unwinding takes place. That is to say, at the start of a dynamic control transfer, a search takes place for the target exit point. That search might skip other exit points which aren't targets of the control transfer. Those skipped exit points are left undisturbed and are still visible during unwinding until their individual binding forms are abandoned. Thus at the time of execution of an `unwind-protect` *`cleanup-form`*, all of the exit points of dynamically surrounding forms are still visible, even ones which are nearer than the targeted exit point.

Example:

```
(block foo
  (unwind-protect
    (progn (return-from foo 42)
           (format t "not reached!\n"))
    (format t "cleanup!\n")))
```

In this example, the protected `progn` form terminates by returning from block `foo`. Therefore the form does not complete and so the output `"not reached!"` is not produced. However, the cleanup form executes, producing the output `"cleanup!"`.

9.37.9 Macro `ignerr`

Syntax:

```
(ignerr form*)
```

Description:

The `ignerr` macro operator evaluates each *`form`* similarly to the `progn` operator. If no forms are present, it returns `nil`. Otherwise it evaluates each *`form`* in turn, yielding the value of the last one.

If the evaluation of any *`form`* is abandoned due to an exception of type `error`, the code generated by the `ignerr` macro catches this exception. In this situation, the execution of the `ignerr` form terminates without evaluating the remaining forms, and yields `nil`.

9.37.10 Macro `ignwarn`

Syntax:

```
(ignwarn form*)
```

Description:

The `ignwarn` macro resembles `ignerr`. It arranges for the evaluation of each *`form`* in left-to-right order. If all the forms are evaluated, then the value of the last one is returned. If no forms are present, then `nil` is returned.

If any *`form`* throws an exception of type `warning` then this exception is intercepted by a handler established by `ignwarn`. This handler reacts by throwing an exception of type `continue`.

The effect is that the warning is ignored, since the handler doesn't issue any diagnostic, and passes control to the warning's continue point.

Note: all sites within **TXR** which throw a `warning` also provide a nearby catch for a `continue` exception, for resuming evaluation at the point where the warning was issued.

9.37.11 Operator `handler-bind`

Syntax:

```
(handler-bind function-form symbol-list body-form*)
```

Description:

The `handler-bind` operator establishes a handler for one or more exception types, and evaluates zero or more *`body-form`*s in a dynamic scope in which that handler is visible.

When the `handler-bind` form terminates normally, the handler is removed. The value of the last *`body-form`* is returned, or else `nil` if there are no forms.

The *`function-form`* argument is an expression which must evaluate to a function. The function must be capable of accepting the exception arguments. All exceptions functions require at least one argument, since the leftmost argument in an exception handler call is the exception type symbol.

The *`symbol-list`* argument is a list of symbols, not evaluated. If it is empty, then the handler isn't eligible for any exceptions. Otherwise it is eligible for any exception whose exception type is a subtype of any of the symbols.

If the evaluation of any *body-form* throws an exception which is not handled within that form, and the handler is eligible for that exception, then the function is invoked. It receives the exception's type symbol as the leftmost argument. If the exception has arguments, they appear as additional arguments in the function call. If the function returns normally, then the exception search continues. The handler remains established until the exception is handled in such a way that a dynamic control transfer abandons the `handler-bind` form.

Note: while a handler's function is executing, the handler is disabled. If the function throws an exception for which the handler is eligible, the handler will not receive that exception; it will be skipped by the exception search as if it didn't exist. When the handler function terminates, either via a normal return or a nonlocal control transfer, then the handler is reenabled.

9.37.12 Macros `handle` and `handle*`

Syntax:

```
(handle try-expression
   {(symbol (arg*) body-form*)}*)
(handle* try-expression
   {(symbol (type-arg arg*) body-form*)}*)
```

Description:

The `handle` macro is a syntactic sugar for the `handler-bind` operator. Its syntax is exactly like that of `catch`. The difference between `handle` and `catch` is that the clauses in `handle` are invoked without unwinding. That is to say, `handle` does not establish an exit point for an exception. When control passes to a clause, it is by means of an ordinary function call and not a dynamic control transfer. No evaluation frames are yet unwound when this takes place.

The `handle` macro establishes a handler, by `handler-bind` whose *symbol-list* consists of every *symbol* gathered from every clause.

The handler function established in the generated `handler-bind` is synthesized from all of the clauses, together with dispatch logic which which passes the exception and its arguments to the first eligible clause.

The *try-expression* is evaluated in the context of this handler.

The clause of the `handle` syntax can return normally, like a function, in which case the handler is understood to have declined the exception, and exception processing continues. To handle an exception, the clause of the `handle` macro must perform a dynamic control transfer, such returning from a block via `return` or throwing an exception.

The `handle*` macro is a variant of `handle` with the following difference: when `handle*` invokes a clause, it passes the exception symbol as the leftmost argument *type-arg*. Then the exception arguments follow. In contrast, only the exception arguments are passed to the clauses of `handle`.

9.37.13 Macro `with-resources`

Syntax:

```
(with-resources ({(sym [init-form [cleanup-form*]])}*)
   body-form*)
```

Description:

The `with-resources` macro provides a sequential binding construct similar to `let*`. Every *sym* is established as a variable which is visible to the *init-form*s of subsequent variables, to all subsequent *cleanup-form*s including that of the same variable, and to the *body-form*s.

If no *init-form* is supplied, then *sym* is bound to the value `nil`.

If an *init-form* is supplied, but no *cleanup-form*s, then *sym* is bound to the value of the *init-form*.

If one or more *cleanup-form*s are supplied in addition to *init-form*, they specify forms to be executed upon the termination of the `with-resources` construct.

When an instance of `with-resources` terminates, either normally or by a nonlocal control transfer, then for each *sym* whose *init-form* had executed, thus causing that *sym* to be bound to a value, the *cleanup-form*s corresponding to *sym* are evaluated in the usual left-to-right order.

The *sym*s are cleaned up in reverse (right-to-left) order. The *cleanup-form*s of the most recently bound *sym* are processed first; those of the least recently bound *sym* are processed last.

When the `with-resources` form terminates normally, the value of the last *body-form* is returned, or else `nil` if no *body-form*s are present.

Note:

From its inception, until **TXR** 265, `with-resources` featured an undocumented behavior. Details are given in the COMPATIBILITY section's Compatibility Version Values subsection, in the notes for compatibility value 265.

Example:

The following expression opens a text file and reads a line from it, returning that line, while ensuring that the stream is closed immediately:

```
(with-resources ((f (open-file "/etc/motd") (close-stream f)))
  (whilet ((l (get-line f)))
    (put-line l)))
```

Note that a better way to initialize exactly one stream resource is with the `with-stream` macro, which implicitly closes the stream when it terminates.

9.37.14 Special variable `*unhandled-hook*`

The `*unhandled-hook*` variable is initialized with `nil` by default.

It may instead be assigned a function which is capable of taking three arguments.

When an exception occurs which has no handler, this function is called, with the following arguments: the exception type symbol, the exception object, and a third value which is either `nil` or else the form which was being evaluated when the exception was thrown. The call occurs before any unwinding takes place.

If the variable is `nil`, or isn't a function, or the function returns after being called, then unwinding takes place, after which some informational messages are printed about the exception, and the process exits with

a failed termination status.

In the case when the variable contains a object other than `nil` which isn't a function, a diagnostic message is printed on the `*stderr*` stream prior to unwinding.

Prior to the function being called, the `*unhandled-hook*` variable is reset to `nil`.

Note: the functions `source-loc` or `source-loc-str` may be applied to the third argument of the `*unhandled-hook*` function to obtain more information about the form.

9.37.15 Macro `defex`

Syntax:

```
(defex {symbol}*)
```

Description:

The macro `defex` records hierarchical relationships among symbols, for the purposes of the use of those symbols as exceptions. It is closely related to the `@(defex)` directive in the **TXR** pattern language, performing the same function.

All symbols are considered to be exception subtypes, and every symbol is implicitly its own exception subtype. This macro does not introduce symbols as exception types; it only introduces subtype-supertype relationships.

If `defex` is invoked with no arguments, it has no effect.

If arguments are present, they must be symbols.

If `defex` is invoked with only one symbol as its argument, it has no effect.

At least two symbols must be specified for a useful effect to take place. If exactly two symbols are specified, then, subject to error checks, `defex` makes the left symbol an *exception subtype* of the right symbol.

This behavior generalizes to three or more arguments: if three or more symbols are specified, then each symbol other than the last is registered as a subtype of the symbol which follows.

If a `defex` has three or more arguments, they are processed from left to right. If errors are encountered during the processing, the correct registrations already made for prior arguments remain in place.

Every symbol is implicitly considered to be its own exception subtype, therefore it is erroneous to explicitly register a symbol as its own subtype.

The symbol `nil` is implicitly a subtype of every exception type. Therefore, it is erroneous to attempt to specify it as a supertype in a registration. Using `nil` as a subtype in a registration is silently permitted, but has no effect. No explicit registration is recorded between `nil` and its successor in the argument list.

The symbol `t` is implicitly the supertype of every exception type. Therefore, it is erroneous to attempt to register it as an exception subtype. Using `t` as a supertype in a registration is also erroneous.

A symbol `a` may not be registered as a subtype of a symbol `b` if the reverse relationship already

exists between those two symbols.

The foregoing rules allow redefinitions to take place, while forbidding cycles from being created in the exception subtype inheritance graph.

Keyword symbols may be used as exception types.

9.37.16 Function `register-exception-subtypes`

Syntax:

```
(register-exception-subtypes {symbol}*)
```

Description:

The `register-exception-subtypes` function constitutes the underlying implementation for the `defex` macro.

The following equivalence applies:

```
(defex a b ...)  <-->  (register-exception-subtypes 'a 'b ...)
```

That is, the `defex` macro works as if by generating a call to the function, with the arguments quoted.

The semantics of the function is precisely that of the macro.

9.37.17 Function `exception-subtype-p`

Syntax:

```
(exception-subtype-p left-symbol right-symbol)
```

Description:

The `exception-subtype-p` function tests whether two symbols are in a relationship as exception types, such that *`left-symbol`* is a direct or indirect exception subtype of *`right-symbol`*.

If that is the case, then `t` is returned, otherwise `nil`.

9.37.18 Function `exception-subtype-map`

Syntax:

```
(exception-subtype-map)
```

Description:

The `exception-subtype-map` function returns a tree structure which captures information about all registered exception types.

The map appears as an association list which contains an entry for every exception symbol, paired with that type's supertype path. The first element in the supertype path is the exception's immediate supertype. The next element is that type's supertype and so on. The last element in every path is the grand supertype `t`.

For instance, if only the types `a`, `b` and `c` existed in the system, and were linked according to this inheritance graph:

```
t ----+--- b --- a
      |
      +--- c
```

such that the supertype of `b` and `c` is `t`, and `a` has `b` as supertype, then the function might return:

```
((a b t) (b t) (c t) (t))
```

or any other equivalent permutation.

The returned list may share substructure, so that the `(t)` sublist is shared among all four entries, and `(b t)` between the first two.

If the program alters the tree structure returned by `exception-map-p`, the consequences are unspecified; this structure may be the actual object which represents the type hierarchy.

9.37.19 Structures `frame`, `catch-frame` and `handle-frame`

Syntax:

```
(defstruct frame nil)
(defstruct catch-frame frame types desc jump)
(defstruct handle-frame frame types fun)
```

Description:

The structure types `frame`, `catch-frame` and `handle-frame` are used by the `get-frames` and `find-frame` functions to represent information about the currently established exception catches (see the `catch` macro) and handlers (see `handler-bind` and `handle`).

The `frame` type serves as the common base for `catch-frame` and `handle-frame`.

Modifying any of the slots of these structures has no effect on the actual frame from which they are derived; the frame structures are only representation which provides information about frames. They are not the actual frames themselves.

Both `catch-frame` and `handle-frame` have a `types` slot. This holds the list of exception type symbols which are matched by the catch or handler.

The `desc` slot of a `catch-frame` holds a list of the descriptions produced by the `catch**` macro. If there are no descriptions, then this member is `nil`, otherwise it is a list whose elements are in correspondence with the list in the `types` slot.

The `jump` slot of a `catch-frame` is an opaque `cptr` ("C pointer") object which is related to the stack address of the catch frame. If it is altered, the catch frame object becomes invalid for the purposes of `invoke-catch`.

The `fun` slot of a `handle-frame` is the registered handler function. Note that all the clauses of a `handle` macro are compiled to a single function, which is established via `handler-bind`, so an instance of the `handle` macro corresponds to a single `handle-frame`.

9.37.20 Function `get-frames`

Syntax:

```
(get-frames)
```

Description:

The `get-frames` function inquires the current dynamic environment in order to retrieve information about established exception catch and handler frames. The function returns a list, ordered from the innermost nesting level to the outermost nesting, of structure objects derived from the `frame` structure type. The list contains two kinds of objects: structures of type `catch-frame` and of type `handle-frame`.

These objects are not the frames themselves, but only provide information about frames. Modifying the slots in these structures has no effect on the original frames. Also, these structures have their own lifetime and can endure after the original frames have disappeared. This has implications for the use of the `invoke-catch` function.

The `handle-frame` structures have a `fun` slot, which holds a function. It may be invoked directly.

A `catch-frame` structure may be passed as an argument to the `invoke-catch` function.

9.37.21 Functions `find-frame` and `find-frames`

Syntax:

```
(find-frame [exception-symbol [frame-type]])
(find-frames [exception-symbol [frame-type]])
```

Description:

The `find-frame` function locates the first (innermost) instance of a specific kind of exception frame (a catch frame or a handler frame) which is eligible for processing an exception of a specific type. If such a frame is found, it is returned. The returned frame object is of the same kind as the objects which comprise the list returned by the function `get-frames`. If such a frame is not found, `nil` is returned.

The *`exception-symbol`* argument specifies a match by exception type: the candidate frame must specify in its list of matches at least one type which is an exception supertype of *`exception-symbol`*. If this argument is omitted, it defaults to `nil` which finds any handler that matches at least one type. There is no way to search for handlers which match an empty set of types; the `find-frame` function skips such frames.

The *`frame-type`* argument specifies which frame type to find. Useful values for this argument are the structure type names `catch-frame` and `handle-frame` or the actual structure type objects which these type names denote. If any other value is specified, the function returns `nil`. If the argument is omitted, it defaults to the type of the `catch-frame` structure. That is to say, by default, the function looks for catch frames.

Thus, if `find-frame` is called with no arguments at all it finds the innermost catch frame, if any exists, or else returns `nil`.

The `find-frames` function is similar to `find-frame` except that it returns all matching frames, ordered from the innermost nesting level to the outermost nesting. If called with no arguments, it returns a list of the catch frames.

9.37.22 Function `invoke-catch`

Syntax:

```
(invoke-catch catch-frame symbol argument*)
```

Description:

The `invoke-catch` function abandons the current evaluation context to perform a nonlocal control transfer directly to the catch described by the *catch-frame* argument, which must be a structure of type `catch-frame` obtained using any of the functions `get-frames`, `find-frames` or `find-frame`.

The control transfer is possible only if the catch frame represented by *catch-frame* structure is still established, and if the structure hasn't been tampered with.

If a given `catch-frame` structure is usable with `invoke-catch`, then a copy of that structure made with `copy-struct` is also usable, denoting the same catch frame.

The *symbol* argument should be an exception symbol. It is passed to the exception frame, as if it had appeared as the first argument of the `throw` function. Similarly, the *argument*s are passed to the catch frame as if they were the trailing arguments of a `throw`. The difference between `invoke-catch` and `throw` is that `invoke-catch` targets a specific catch frame as its exit point, rather than searching for a matching catch or handler frame. That specific frame receives the control. The frame receives control even if it it is not otherwise eligible for catching the exception type denoted by *symbol*.

9.37.23 Macro `assert`

Syntax:

```
(assert expr [format-string format-arg*])
```

Description:

The `assert` macro evaluates *expr*. If *expr* yields any true value, then `assert` terminates normally, and that value is returned.

If instead *expr* yields `nil`, then `assert` throws an exception of type `assert`. The exception carries an informative character string that contains a diagnostic detailing the expression which yielded `nil`, and the source location of that expression, if available.

If the *format-string* and possibly additional format arguments are given to `assert` then those arguments are used to format additional text which is appended to the diagnostic message after a separating character such as a colon.

9.38 Static Error Diagnosis

This section describes a number of features related to the diagnosis of errors during the static processing of program code prior to evaluation. The material is of interest to developers of macros intended for broad reuse.

9.38.1 Error Exceptions

TXR Lisp uses exceptions of type `eval-error` to identify erroneous situations during both transformation of code and its evaluation. These exceptions have one argument, which is a character string. If not handled by program code, `eval-error` exceptions are specially recognized and treated by the built-in handling logic. The message is incorporated into diagnostic output which includes more information which is deduced.

9.38.2 Warning Exceptions

TXR Lisp uses exceptions of type `warning` to identify certain situations of interest. Ordinary non-deferrable warnings have a structure identical to errors, except for the exception symbol. **TXR**'s provides built-in "auto continue" handling for warnings. If a warning exception is not intercepted by a catch or an accepting handler, then a diagnostic is issued on the `*stderr*` stream, after which a `continue` exception is thrown with no arguments. If that `continue` exception is not handled, then control returns normally to the point that exception to resume the computation which generated the warning.

Callers which invoke code that may generate warning exceptions are therefore not required to handle them. However, callers which do handle warning exceptions expect to be able to throw a `continue` exception in order to resume the computation that triggered the warning, without allowing other handlers to see the exception.

The generation of a warning should thus conform to the following pattern:

```
(catch
  (throw 'warning "message")
  (continue ()))
```

9.38.3 Deferrable Warnings

TXR supports a form of diagnostic known as a *deferrable warning*. A deferrable warning is distinguished in two ways. Firstly, it is either of the type `defr-warning` or subtyped from that type. The `defr-warning` type itself is a direct subtype of `warning`.

Secondly, a deferrable warning carries an additional tag argument after the exception message. A deferrable exception is thrown according to this pattern:

```
(catch
  (throw 'defr-warning "message" . tag)
  (continue ()))
```

TXR's built-in exception handling logic reacts specially to the presence of the tag material in the exception. First, the global *tentative definition list* is searched for the presence of the tag, using `equal` equality. If the tag is found, then the warning is discarded. If the tag is not found, then the exception argument list is added to the global *deferred warning list*. In either case, the `continue` exception is thrown to resume the computation which threw the warning, as in the case of an ordinary non-deferrable warning.

The purpose of this mechanism is to suppress warnings which become superfluous when more of the program code is examined. For instance, a warning about a call to an undefined function is superfluous if a definition of that function is supplied later, yet before that function call is executed.

Deferred warnings accumulate in the deferred warning list from which they can be removed. The list is purged at various times such as when a top-level load completes, and the deferred warnings are released, as if by a call to the `release-deferred-warnings` function.

9.38.4 Functions `compile-error` and `compile-warning`

Syntax:

```
(compile-error context-obj fmt-string fmt-arg*)
(compile-warning context-obj fmt-string fmt-arg*)
```

Description:

The functions `compile-error` and `compile-warning` provide a convenient and uniform way for code transforming functions such as macro-expanders to generate diagnostics. The `compile-error` function throws an exception of type `eval-error`. The `compile-warning` function throws an exception of type `warning` and internally provides a `catch` for the `continue` exception which allow a warning handler to resume execution after the warning. If a handler throws a `continue` exception which is caught by `compile-warning`, then `compile-warning` returns `nil`.

Because `compile-warning` throws a non-error exception, it returns `nil` in the event that no catch is found for the exception, and no handler which accepts it.

The argument conventions are the same for both functions. The *`context-obj`* is typically a compound form to which the diagnostic applies.

The functions produce a diagnostic message which incorporates the location information and symbol obtained from *`context-obj`* and the `format`-style arguments *`fmt-string`* and its *`fmt-arg`*s.

9.38.5 Function `compile-defr-warning`

Syntax:

```
(compile-defr-warning context-obj tag
   fmt-string fmt-arg*)
```

Description:

The `compile-defr-warning` function throws an exception of type `defr-warning` and internally provides a `catch` for the `continue` exception needed to resume after the warning.

The function produces a diagnostic message which incorporates the location information and symbol obtained from *`context-obj`* and the `format`-style arguments *`fmt-string`* and its *`fmt-arg`*s. This diagnostic message constitutes the first argument of the exception. The *`tag`* argument is taken as the second argument.

If the exception isn't intercepted by a catch or by an accepting handler, `compile-defr-warning` returns `nil`. In also returns `nil` if it catches a `continue` exception.

9.38.6 Function `purge-deferred-warning`

Syntax:

```
(purge-deferred-warning tag)
```

Description:

The `purge-deferred-warning` removes all warnings marked with *`tag`* from the deferred list. It also removes all tags matching *`tag`* from the tentative definition list. Tags are compared using the `equal` function.

9.38.7 Function `register-tentative-def`

Syntax:

```
(register-tentative-def tag)
```

Description:

The `register-tentative-def` function adds *tag* to the list of tentative definitions which are used to suppress deferrable warnings.

The idea is that a definition of some construct has been seen, but not yet executed. Thus the construct is not defined, but it can reasonably be expected that it will be defined; hence, warnings about its nonexistence can be suppressed.

For example, in the following code, when the expression `(foo)` is being expanded and transformed, the `foo` function does not exist:

```
(progn (defun foo ()) (foo))
```

The function won't be defined until the `progn` is evaluated. Thus a warning is generated that `(foo)` refers to an undefined function. However, this warning is discarded, because the expander for `defun` registers a tentative definition tag for `foo`.

When the definition of `foo` takes place, the `defun` operator will call `purge-deferred-warning` which will remove not only all accumulated warnings related to the undefinedness of `foo` but also remove the tentative definition.

Note: this mechanism isn't perfect because it will still suppresses the warning in situations like

```
(progn (if nil (defun foo ())) (foo))
```

9.38.8 Function `tentative-def-exists`

Syntax:

```
(tentative-def-exists tag)
```

Description:

The `tentative-def-exists` function checks whether *tag* has been registered via `register-tentative-def` and not yet purged by `purge-deferred-warning`.

9.38.9 Function `defer-warning`

Syntax:

```
(defer-warning args)
```

Description:

The `defer-warning` function attempts to register a deferred warning. The *args* argument corresponds to the arguments which are passed to the `throw` function in order to generate a warning exception, not including the exception symbol.

Args is expected to have at least two elements, the second of which is a deferred warning tag.

The `defer-warning` function returns `nil`.

Note: this function is intended for use in exception handlers. The following example shows a handler which intercepts warnings. It defers deferrable warnings, and prints ordinary warnings:

```
(handle
```

```
(some-form ..) ;; some code which might generate warnings
(defr-warning (msg tag) ;; catch deferrable and defer
  (defer-warning (cons msg tag))
  (throw 'continue)) ;; warning processed: resume execution
(warning (msg)
  (put-line `warning: @msg`) ;; print non-deferrable
  (throw 'continue))) ;; warning processed: resume execution
```

9.38.10 Function `release-deferred-warnings`

Syntax:

```
(release-deferred-warnings)
```

Description:

The `release-deferred-warnings` removes all warnings from the deferred list. Then, it issues each deferred warning as an ordinary warning.

Note: there is normally no need for user programs to use this function since deferred warnings are issued automatically.

9.38.11 Function `dump-deferred-warnings`

Syntax:

```
(dump-deferred-warnings stream)
```

Description:

The `dump-deferred-warnings` empties the list of deferred warnings, and converts each one into a diagnostic message sent to sent to *`stream`*. After the diagnostics are printed, the list of pending warnings is cleared.

Note: there is normally no need for user programs to use this function since deferred warnings are issued automatically.

9.39 Delimited Continuations

TXR Lisp supports delimited continuations, which are integrated with the `block` feature. Any named or anonymous block, including the implicit blocks created around function bodies, can be used as the delimiting *prompt* for the capture of a continuation.

A delimited continuation is section of a possible future of the computation, up to a delimiting prompt, *reified* as a first class function.

Example:

```
(defun receive (cont)
  (format t "cont returned ~a\n" (call cont 3)))

(defun function ()
  (sys:capture-cont 'abcd (fun receive)))

(block abcd
  (format t "function returned ~a\n" (function))
  4)
```

```
Output:

function returned 3
cont returned 4
function returned t
```

Evaluation begins with the `block` form. This form calls `function` which uses `sys:capture-cont` to capture a continuation up to the `abcd` prompt. The continuation is passed to the `receive` function as an argument.

This captured object represents the continuation of computation up to that prompt. It appears as a one-argument function which, when called, resumes the captured computation. Its argument emerges out of the `sys:capture-cont` call as a return value. When the computation eventually returns all the way to the delimiting prompt, the return value of that prompt will then appear as the return value of the continuation function.

In this example, the function `receive` immediately invokes the continuation function which it receives, passing it the argument value `3`. And so, evaluation now continues in the resumed future represented by the continuation. Inside the continuation, `sys:capture-cont` appears to return, yielding the value `3`. This bubbles up through `function` up to the `block abcd` where a message is printed: `"function returned 3"`.

The `block` terminates, yielding the value 4. Thereby, the continuation ends, since it is delimited up to that block. Control now returns to the `receive` function which invoked the continuation, where the function call form `(call cont)` terminates, yielding the value `4` that was returned by the continuation's delimiting `block` form. The message `"cont returned 4"` is printed. The `receive` function returns normally, returning the value `t` which emerged from the `format` call. Control is now back in `function` where the `sys:capture-cont` form terminates and returns the `t`. This bubbles up to `block` which prints `"function returned t"`.

In summary, a continuation represents, as a function, the subsequent computation that is to take place starting at some point, up to some recently established, dynamically enclosing delimiting prompt. When the continuation is captured, that future doesn't have to take place; an alternative future can carry out in which that continuation is available as a function. That alternative future can invoke the continuation at will. Invocations (resumptions) of the continuation appear as additional returns from the capture operator. A resumption of a continuation terminates when the delimiting prompt terminates, and the continuation yields the value which emerges from the prompt.

Delimited continuations are implemented by capturing a segment of the evaluation stack between the prompt and the capture point. When a continuation is resumed, this saved copy of a stack segment is inserted on top of the current stack and the procedure context is resumed such that evaluation appears to emerge from the capture operator. As the continuation runs to completion, it simply pops these inserted stack frames naturally. Eventually it pops out of the delimiting prompt, at which point control ends up at the point which invoked the continuation function.

The low-level operator for capturing a continuation is `sys:capture-cont`. More expressive and convenient programming with continuations is provided by the macros `obtain`, `obtain-block`, `yield-from` and `yield`, which create an abstraction which models the continuation as a suspended procedure supporting two-way communication of data. A `suspend` operator is provided, which is more general. It is identical to the `shift` operator described in various computer science literature about delimited continuations, except that it refers to a specific delimiting prompt by name.

Continuations raise the issue of what to do about unwinding. The language Scheme provides the much criticized `dynamic-wind` operator which can execute initialization and clean-up code as a continuation is

entered and abandoned. **TXR** takes a simpler, albeit risky approach. It provides a non-unwinding escape operator `sys:abscond-from` for use with continuations. Code which has captured a continuation can use this operator to escape from the delimiting block without triggering any unwinding among the frames between the capture point and the delimiter. When the continuation is restarted, it will then do so with all of the resources associated with it frames intact. When the continuation executes normal returns within its context, the unwinding takes place then. Thus tidy, "thread-like" use of continuations is possible with a small measure of coding discipline. Unfortunately, the absconding operator is dangerous: its use breaks the language guarantee that clean-up associated with a form is done no matter how a form terminates.

9.39.1 Comparison with Lexical Closures

Delimited continuations resemble lexical closures in some ways. Both constructs provide a way to return to some context whose evaluation has already been abandoned, and to access some aspects of that context. However, lexical closures are statically scoped. Closures capture the lexically apparent scope at a given point, and produce a function whose body has access to that scope, as well as to some arbitrary arguments. Thus, a lexical scope is reified as a first-class function. By contrast, a delimited continuation is dynamic. It captures an an entire segment of a program activation chain, up to the delimiting prompt. This segment includes scopes which are not lexically visible at the capture point: the scopes of parent functions. Moreover, the segment includes not only scopes, but also other aspects of the evaluation context, such as the possibility of returning to callers, and the (captured portion of) the original dynamic environment, such as exception handlers. That is to say, a lexical closure's body cannot return to the surrounding code or see any of its original dynamic environment; it can only inspect the environment, and then return to its own caller. Whereas a restarted delimited continuation can continue evaluation of the surrounding code, return to surrounding forms and parent functions, and access the dynamic environment. The continuation function returns to its caller when that entire restarted context terminates, whereas a closure returns to its caller as soon as the closure body terminates.

9.39.2 Differences in Compiled vs. Interpreted Behavior

Delimited continuations in **TXR** expose a behavioral difference between compiled and interpreted code which mutates the values of lexical variables.

When a continuation is captured in compiled code, it captures not only the bindings of lexical variables, but also potentially their current values at the time of capture. What this means is that whenever the continuation is resumed, those variables will appear to have the captured values, regardless of any mutations that have taken place since. In other words, the captured future includes those specific values. This is because in compiled code, variables are allocated on the stack, which is copied as part of creating a continuation. Those variables are effectively newly instantiated in each resumption of the continuation, when the captured stack segment is reinstated into the stack, and take on those original values.

In contrast, interpretation of code only maintains an environment pointer on the stack; the lexical environment is a dynamically allocated object whose contents aren't included in the continuation's stack segment capture. If the captured variables are modified after the capture, the continuation will see the updated values: all resumptions of the continuation share the same instance of the captured environment among themselves, and with the original context where the capture took place.

An additional complication is that when compiled code captures lexical closures, captured variables are moved into dynamic storage and then they become shared: the semantics of the mutation of those variables is then similar to the situation in interpreted code. Therefore, the above described non-sharing capture behavior of compiled code is not required to hold.

In continuation-based code which relies on mutation of lexical variables created with `let` or `let*`, the macros `hlet` and `hlet*` can be used instead. These macros create variable bindings whose storage is always outside of the stack, and therefore the variables will exhibit consistent interpreted and compiled semantics under continuations. All contexts which capture the same lexical binding of a given

`hlet`/`hlet*` variable share a single instance. The most recent assignment to the variable taking place in any context establishes its value, as seen by any other context. The resumption of a continuation will not restore such a variable to a previous value.

If the affected variables are other kinds of bindings such as function parameters or variables created with specialized binding constructs such as `with-stream`, additional coding changes may be required to get interpreted code working under compilation.

9.39.3 Function `sys:capture-cont`

Syntax:

```
(sys:capture-cont name receive-fun [context-form])
```

Description:

The `sys:capture-cont` function captures a continuation, and also serves as the resume point for the resulting continuation. Which of these two situations is the case (capture or resumption) is distinguished by the use of the *`receive-fun`* argument, which must be a function capable of being called with one argument.

A block named *`name`* must be visible; the continuation is delimited by the closest enclosing block of this name.

The optional *`context-form`* argument should be a compound form. If `sys:capture-cont` reports an error, it reports it against this form, and uses the form's operator symbol as the name of the function which encountered the error. If the argument is omitted, `sys:capture-cont` uses its own name.

The `sys:capture-cont` function captures a continuation, represented as a function. It immediately calls *`receive-fun`*, passing it it the continuation function as an argument. If *`receive-fun`* returns normally, then `sys:capture-cont` returns whatever value *`receive-fun`* returns.

Resuming a continuation is done by invoking the continuation function. When this happens, the entire continuation context is restored by recreating its captured evaluation frames on top of the current stack. Inside the continuation, the `sys:capture-cont` function call which captured the continuation now appears to return, and yields a value. That value is precisely the value which was just passed to the continuation function moments ago.

The resumed continuation can terminate in one of three ways. Firstly, it can simply keep executing until it discards all of its evaluation frames below the delimiting block, and then allows that block to terminate naturally by evaluating the last form contained in the block. Secondly, can use `return-from` against its delimiting block to explicitly abandon all evaluations in between and terminate that block. Or it may perform a nonlocal control transfer past the delimited block somewhere into the evaluation frames of the caller. In the first two cases, the termination of the block turns into an ordinary return from the continuation function, and the result value of the terminated block becomes the return value of that function call. In the last case, the call of the continuation function is abandoned and unwinding continues through the caller.

If the symbol `sys:cont-poison` is passed to the continuation function, the continuation will be resumed in a different manner: its context will be restored as in the ordinary resume case, whereupon it will be immediately abandoned by a nonlocal exit, causing unwinding to take place across all of the continuation's evaluation frames. The function then returns `nil`.

If the symbol `sys:cont-free` is passed to the continuation function, the continuation isn't be

resumed at all; rather, the buffer which holds the saved context of the continuation is released. Thereafter, an attempt to resume the continuation results in an error exception being thrown. After releasing the buffer, the function returns `nil`.

Notes:

The continuation function may be used any time after it is produced, and may be called more than once, regardless of whether the originally captured dynamic context is still executing. The continuation object may be communicated into the resumed continuation, which can then use it to call itself, resulting in multiple nested resumptions of the same continuation. A delimited continuation is effectively a first class function.

The underlying continuation object produced by `sys:capture-cont` stores a copy of the captured dynamic context. Whenever the continuation function is invoked, a copy of the captured is reinstated as if it were a new context. Thus each apparent return from the `sys:capture-cont` inside a resumed continuation is not actually made in the original context, but in a copy of that context. That context can be resumed multiple times sequentially or recursively.

Just like lexical closures, continuations do not copy lexical environments; they capture lexical environments by reference. If a continuation modifies the values of captured lexical variables, those modifications are visible to other resumptions of the same continuation, to other continuations which capture the same environment, to lexical closures which capture the same environment and to the original context which created that environment, if it is still active.

Unlike lexical closures, continuations do capture the local bindings of special variables. That is to say, if `*var*` is a special variable, then a lexical closure created inside a `(let ((*var* 42)) ...)` form will not capture the local rebinding of `*var*` which holds 42. When the closure is invoked and accesses `*var*`, it accesses whatever value of `*var*` is dynamically current, as dictated by the environment which calls the closure, rather than the capturing environment.

With continuations, the behavior is different. If a continuation is captured inside a `(let ((*var* 42)) ...)` form then it does capture the local binding. This is regardless whether the delimited prompt of the capture is enclosed in this form, or outside of the form. The special variable has a binding in a dynamic environment. There is always a reference to a current dynamic environment associated with every evaluation context, and a continuation captures that reference. Because it is a reference, it means that the binding is shared. That is to say, all invocations of all continuations which capture the same dynamic environment in which that `(let ((*var* 42)) ...)` binding was made share the same binding; if `*var*` is modified by assignment, the modification is visible to all those views.

Inside a resumed continuation, a form which binds a special variable such as `(let ((*var* 42)) ...)` may terminate. As expected, this causes the binding to be removed, revealing either another local binding of `*var*` or the global binding. However, this unbinding only affects only that that executing continuation; it has no effect inside other instances of the same continuation or other continuations which capture the same variable. Unbinding isn't a mutation of the dynamic environment, but may be understood as merely the restoration of an earlier dynamic environment reference.

Example:

The following example shows an implementation of the `suspend` operator.

```
(defmacro suspend (:form form name var . body)
  ^(sys:capture-cont ',name (lambda (,var)
                              (sys:abscond-from ,name ,*body))
```

```
',form))
```

9.39.4 Operator `sys:abscond-from`

Syntax:

```
(sys:abscond-from name [value])
```

Description:

The `sys:abscond-from` operator closely resembles `return-from` and performs the same function: it causes an enclosing block *name* to terminate with *value* which defaults to `nil`.

However, unlike `return-from`, `sys:abscond-from` does not perform any unwinding.

This operator should never be used for any purpose other than implementing primitives for the use of delimited continuations. It is used by the `yield-from` and `yield` operators to escape out of a block in which a continuation has been captured. Neglecting to unwind is valid due to the expectation that control will return into a restarted copy of that context.

9.39.5 Function `sys:abscond*`

Syntax:

```
(sys:abscond* name [value])
```

Description:

The `sys:abscond*` function is similar to the `sys:abscond-from` operator, except that `name` is an ordinary function parameter, and so when `return*` is used, an argument expression must be specified which evaluates to a symbol. Thus `sys:abscond*` allows the target block of a return to be dynamically computed.

The following equivalence holds between the operator and function:

```
(sys:abscond-from a b)  <-->  (sys:abscond* 'a b)
```

Expressions used as *name* arguments to `abscond*` which do not simply quote a symbol have no equivalent in `abscond-from`.

9.39.6 Macros `obtain` and `yield-from`

Syntax:

```
(obtain forms*)
(yield-from name [form])
```

Description:

The `obtain` and `yield-from` macros closely interoperate.

The `obtain` macro treats zero or more *form*s as a suspendable execution context called the *obtain block*. It is expected that *form*s establish a block named *name* and return its result value to `obtain`.

Without evaluating any of the forms in the obtain block, `obtain` returns a function, which takes one optional argument. This argument, called the *resume value*, defaults to `nil` if it is omitted.

The function represents the suspended execution context.

The context is resumed whenever the function is called, and executes until the next `yield-from` statement which references the block named *name*. The function's reply argument is noted.

If the `yield-from` specifies a *form* argument, then the execution context suspends, and the resume function terminates and returns the value of that form. When the function is called again to resume the context, the `yield-from` returns the previously noted resume value (and the new resume value just passed is noted in its place).

If the `yield-from` specifies no *form* argument, then it briefly suspends the execution context only to retrieve the resume value, without producing an item. Since no item is produced, the resume function does not return. The execution context implicitly resumes.

When execution reaches the last form in the obtain block, the resume value is discarded. The execution context terminates, and the most recent call to the resume function returns the value of that last form.

Notes:

The `obtain` macro registers a finalizer against the returned resume function. The finalizer invokes the function, passing it the symbol `sys:cont-poison`, thereby triggering unwinding in the most recently captured continuation. Thus, abandoned `obtain` blocks are subject to unwinding when they become garbage.

The `yield-from` macro works by capturing a continuation and performing a nonlocal exit to the nearest block called *name*. It passes a special yield object to that block. The `obtain` macro generates code which knows what to do with this special yield object.

Examples:

The following example shows a function which recursively traverses a `cons` cell structure, yielding all the non-`nil` atoms it encounters. Finally, it returns the object `nil`. The function is invoked on a list, and the invocation is wrapped in an `obtain` block to convert it to a generating function.

The generating function is then called six times to retrieve the five atoms from the list, and the final `nil` value. These are collected into a list.

This example demonstrates the power of delimited continuations to suspend and resume a recursive procedure.

```
(defun yflatten (obj)
  (labels ((flatten-rec (obj)
             (cond
               ((null obj))
               ((atom obj) (yield-from yflatten obj))
               (t (flatten-rec (car obj))
                  (flatten-rec (cdr obj))))))
    (flatten-rec obj)
    nil))

(let ((f (obtain (yflatten '(a (b (c . d)) e)))))
  (list [f] [f] [f] [f] [f] [f]))
--> (a b c d e nil)
```

The following interactive session log exemplifies two-way communication between the main code

and a suspending function.

Here, `mappend` is invoked on a list of symbols representing fruit and vegetable names. The objective is to return a list containing only fruits. The `lambda` function suspends execution and yields a question out of the `map` block. It then classifies the item as a fruit or not according to the reply it receives. The reply emerges as the result value of the `yield-from` call.

The `obtain` macro converts the block to a generating function. The first call to the function is made with no argument, because the argument would be ignored anyway. The function returns a question, asking whether the first item in the list, the potato, is a fruit. To answer positively or negatively, the user calls the function again, passing in `t` or `nil`, respectively. The function returns the next question, which is answered in the same manner.

When the question for the last item is answered, the function call yields the final item: the ordinary result of the block, which is the list of fruit names.

```
1> (obtain
     (block map
       (mappend (lambda (item)
                  (if (yield-from map `is @item a fruit?`)
                    (list item)))
                '(potato apple banana lettuce orange carrot))))
#<interpreted fun: lambda (: reply)>
2> (call *1)
"is potato a fruit?"
3> (call *1 nil)
"is apple a fruit?"
4> (call *1 t)
"is banana a fruit?"
5> (call *1 t)
"is lettuce a fruit?"
6> (call *1 nil)
"is orange a fruit?"
7> (call *1 t)
"is carrot a fruit?"
8> (call *1 nil)
(apple banana orange)
```

The following example demonstrates an accumulator. Values passed to the resume function are added to a counter which is initially zero. Each call to the function returns the updated value of the accumulator. Note the use of `(yield-from acc)` with no arguments to receive the value passed to the first call to the resume function, without yielding an item. The first return value `1` is produced by the `(yield-from acc sum)` form, not by `(yield-from acc)`. The latter only obtains the initial value `1` and uses it to establish the seed value of the accumulator. Without causing the resume function to terminate and return, control passes into the loop, which yields the first item, causing the resume function call `(call *1 1)` to return `1`:

```
1> (obtain
     (block acc
       (let ((sum (yield-from acc)))
          (while t (inc sum (yield-from acc sum))))))
#<interpreted fun: lambda (: resume-val)>
2> (call *1 1)
1
```

```
3> (call *1 2)
3
4> (call *1 3)
6
5> (call *1 4)
10
```

9.39.7 Macro `obtain-block`

Syntax:

```
(obtain-block name forms*)
```

Description:

The `obtain-block` macro combines `block` and `obtain` into a single expression. The *form*s are evaluated in a block named `name`.

That is to say, the following equivalence holds:

```
(obtain-block n f ...)  <-->  (obtain (block n f ...))
```

9.39.8 Macro `yield`

Syntax:

```
(yield [form])
```

Description:

The `yield` macro is to `yield-from` as `return` is to `return-from`: it yields from an anonymous block.

It is equivalent to calling `yield-from` using `nil` as the block name.

In other words, the following equivalence holds:

```
(yield x)  <-->  (yield-from nil x)
```

Example:

```
;; Yield the integers 0 to 4 from a for loop, taking
;; advantage of its implicit anonymous block:

(defvarl f (obtain (for ((i 0)) ((< i 5)) ((inc i))
                      (yield i))))

[f] -> 0
[f] -> 1
[f] -> 2
[f] -> 3
[f] -> 4
[f] -> nil
[f] -> nil
```

9.39.9 Macros `obtain*` and `obtain*-block`

Syntax:

```
(obtain* forms*)
(obtain*-block name forms*)
```

Description:

The `obtain*` and `obtain*-block` macros implement a useful variation of `obtain` and `obtain-block`.

The `obtain*` macro differs from `obtain` in exactly one regard: prior to returning the function, it invokes it one time, with the argument value `nil`.

Thus, the following equivalence holds

```
(obtain* forms ...)   <-->  (let ((f (obtain forms ...)))
                              (call f)
                              f)
```

In other words, the suspended block is immediately resumed, so that it executes either to completion (in which case its value is discarded), or to its first `yield` or `yield-from` call (in which case the yielded value is discarded).

Note: the `obtain*` macro is useful in creating suspensions which accept data rather than produce data.

The `obtain*-block` macro combines `obtain*` and `block` in the same manner that `obtain-block` combines `obtain` and `block`.

Example:

```
;; Pass three values into suspended block,
;; which get accumulated into list.
(let ((f (obtain*-block nil
           (list (yield nil) (yield nil) (yield nil)))))
  (call f 1)
  (call f 2)
  (call f 3))  ->  (1 2 3)

;; Under obtain, extra call is required:
(let ((f (obtain-block nil
           (list (yield nil) (yield nil) (yield nil)))))
  (call f nil)  ;; execute block to first yield
  (call f 1)    ;; resume first yield with 1
  (call f 2)
  (call f 3))  ->  (1 2 3)
```

9.39.10 Macro `suspend`

Syntax:

```
(suspend block-name var-name body-form*)
```

Description:

The `suspend` operator captures a continuation up to the prompt given by the symbol *`block-name`* and binds it to the variable name given by *`var-name`*, which must be a symbol suitable for binding variables with `let`.

Each *body-form* is then evaluated in the scope of the variable *var-name*.

When the last *body-form* is evaluated, a nonlocal exit takes place to the block named by *block-name* (using the `sys:abscond-from` operator, so that unwinding isn't performed).

When the continuation bound to *var-name* is invoked, a copy of the entire block *block-name* is restarted, and in that copy, the `suspend` call appears to return normally, yielding the value which had been passed to the continuation.

Example

Define John McCarthy's `amb` function using `block` and `suspend`:

```
(defmacro amb-scope (. forms)
  ^(block amb-scope ,*forms))

(defun amb (. args)
  (suspend amb-scope cont
    (each ((a args))
      (if a
        (iflet ((r (call cont a)))
          (return-from amb-scope r))))))
```

Use `amb` to bind the `x` and `y` which satisfy the predicate `(eql (* x y) 8)` nondeterministically:

```
(amb-scope
  (let ((x (amb 1 2 3))
        (y (amb 4 5 6)))
    (amb (eql (* x y) 8))
    (list x y)))
-> (2 4)
```

9.39.11 Macros `hlet` and `hlet*`

Syntax:

```
(hlet ({sym | (sym init-form)}*) body-form*)
(hlet* ({sym | (sym init-form)}*) body-form*)
```

Description:

The `hlet` and `hlet*` macros behave exactly like `let` and `let*`, respectively except that they guarantee that the variable bindings are allocated in storage which isn't captured by delimited continuations.

The `h` in the names stands for "heap", serving as a mnemonic based on the implementation concept of these bindings being "heap-allocated".

9.40 Regular-Expression Library

TXR provides a "pure" regular-expression implementation based on automata theory, which equates regular expressions, finite automata and sets of strings. A regular expression determines whether or not a string of input characters belongs to a set. **TXR** regular expressions do not support features such as "anchoring" a match to the start or end of a string, or capturing parenthesized subexpression matches into registers. Parenthesis syntax denotes only grouping, with no additional meaning.

The semantics of whether a regular expression is used for a substring search, prefix match, suffix match, string splitting and so forth comes from the functions which use regular expressions to perform these operations.

9.40.1 Regular Expressions as Functions

Syntax:

```
[regex [start [from-end]] string]
```

Description:

A regular expression is callable as a function in **TXR Lisp**. When used this way, it requires a string argument. It searches the string for the leftmost match for itself, and returns the matching substring, which could be empty. If no match is found, it returns `nil`.

A regex takes one, two, or three arguments. The required *`string`* is always the rightmost argument. This allows for convenient partial application over optional arguments using macros in the `op` family, and macros in which the `op` syntax is implicit.

The optional arguments *`start`* and *`from-end`* are treated exactly as their like-named counterparts in the `search-regst` function.

Example:

Keep those elements from a list of strings which match the regular expression `#/a.*b/`:

```
(keep-if #/a.*b/ '#"abracadabra zebra hat adlib adobe deer")
--> ("abracadabra" "adlib" "adobe")
```

9.40.2 Functions `search-regex`, `range-regex` and `search-regst`

Syntax:

```
(search-regex string regex [start [from-end]])
(range-regex string regex [start [from-end]])
(search-regst string regex [start [from-end]])
```

Description:

The `search-regex` function searches through *`string`* starting at position *`start`* for a match for *`regex`*.

If *`start`* is omitted, the search starts at position 0. If *`from-end`* is specified and has a non-`nil` value, the search proceeds in reverse, from the position just beyond the last character of *`string`*, toward *`start`*.

If *`start`* exceeds the length of the string, then `search-regex` returns `nil`.

If *`start`* is negative then it indicates positions from the end of the string, such that -1 is the last character, -2 the second last and so forth. If the value is so negative that it refers beyond the start of the string, then the starting position is deemed to be zero.

If *`start`* is equal to the length of *`string`*, and thus refers to the position one character past its length, then a match occurs at that position if *`regex`* admits such a match.

The `search-regex` function returns `nil` if no match is found, otherwise it returns a cons, whose `car` indicates the position of the match, and whose `cdr` indicates the length of the match.

If *regex* is capable of matching empty strings, and no other kind of match is found within *string*, then search regex reports a zero length match. If *from-end* is false, then this match is reported at *start*, otherwise it is reported at the position one character beyond the end of the string.

The `range-regex` function is similar to `search-regex`, except that when a match is found, it returns a position range, rather than a position and length. A range object is returned whose `from` field indicates the position of the match, and whose `to` indicates the position one element past the last character of the match. If the match is empty, the two integers are equal.

Also see the `rr` function, which provides an alternative argument syntax for the semantics of `range-regex`.

The `search-regst` differs from `search-regex` in the representation of the return value in the matching case. Rather than returning the position and length of the match, it returns the matching substring of *string*.

9.40.3 Functions `match-regex` and `match-regst`

Syntax:

```
(match-regex string regex [position])
(match-regst string regex [position])
```

Description:

The `match-regex` function tests whether *regex* matches at *position* in *string*.

If *position* is not specified, it is taken to be zero. Negative values of *position* index from the right end of the string such that -1 refers to the last character. Excessively negative values which index before the first character cause `nil` to be returned.

If the regex matches, then the length of the match is returned. If it does not match, then `nil` is returned.

The `match-regst` differs from `match-regex` in the representation of the return value in the matching case. Rather than returning the length of the match, it returns matching substring of *string*.

9.40.4 Functions `match-regex-right` and `match-regst-right`

Syntax:

```
(match-regex-right string regex [end-position])
(match-regst-right string regex [end-position])
```

Description:

The `match-regex-right` function tests whether some substring of *string* which terminates at the character position just before *end-position* matches *regex*.

If *end-position* is not specified, it defaults to the length of the string, and the function performs a right-anchored regex match.

The *end-position* argument can be a negative integer, in which case it denotes positions from the end of the string, such that -1 refers to the last character. If the value is excessively negative such that the position immediately before it is before the start of the string, then `nil` is returned.

If *end-position* is a positive value beyond the length of *string*, then, likewise, `nil` is returned.

If a match is found, then the length of the match is returned.

A more precise way of articulating the role of *end-position* is that for the purposes of matching, `string` is considered to terminate just before *end-position*: in other words, that *end-position* is the length of the string. The match is then anchored to the end of this effective string.

The `match-regst-right` differs from `match-regst-right` in the representation of the return value in the matching case. Rather than returning the length of the match, it returns the matching substring of *string*.

Examples:

```
;; Return matching portion rather than length thereof.

(defun match-regex-right-substring (str reg : end-pos)
  (set end-pos (or end-pos (length str)))
  (let ((len (match-regex-right str reg end-pos)))
    (if len
      [str (- end-pos len)..end-pos]
      nil)))

(match-regex-right-substring "abc" #/c/) -> ""

(match-regex-right-substring "acc" #/c*/) -> "cc"

;; Regex matches starting at multiple positions, but all
;; the matches extend past the limit.
(match-regex-right-substring "acc" #/c*/ 2) -> nil

;; If the above behavior is not wanted, then
;; we can extract the string up to the limiting
;; position and do the match on that.
(match-regex-right-substring ["acc" 0..2] #/c*/) -> "c"

;; Equivalent of above call
(match-regex-right-substring "ac" #/c*/) -> "c"
```

9.40.5 Function `regex-prefix-match`

Syntax:

```
(regex-prefix-match regex string [position])
```

Description:

The `regex-prefix-match` determines whether the input string might be the prefix of a string which matches regular expression *regex*.

The result is true if the input string matches *regex* exactly. However, it is also true in situations in which the input string doesn't match *regex*, yet can be extended with one or more additional characters beyond the end such that the extended string **does** match.

The *string* argument must be a character string. The function takes the input string to be the suffix of *string* which starts at the character position indicated by the *position* argument. If that argument is omitted, then *string* is taken as the input in its entirety. Negative values index backwards from the end of *string* according to the usual conventions elsewhere in the library.

Note: this function is not to be confused for the semantics of a regex matching a prefix of a string: that capability is provided by the functions `match-regex`, `m^`, `r^`, `f^` and `fr^`.

Examples:

```
;; The empty string is not a viable prefix match for
;; a regex that matches no strings at all:
(regex-prefix-match #/~.*/ "") -> nil
(regex-prefix-match #/[]/ "") -> nil

;; The empty string is a viable prefix of any regex
;; which matches at least one string:
(regex-prefix-match #// "") -> t
(regex-prefix-match #/abc/ "") -> t

;; This string doesn't match the regex because
;; it doesn't end in b, but is a viable prefix:
(regex-prefix-match #/a*b/ "aa") -> t

(regex-prefix-match #/a*b/ "ab") -> t

(regex-prefix-match #/a*b/ "ac") -> nil

(regex-prefix-match #/a*b/ "abc") -> nil
```

9.40.6 Function `regsub`

Syntax:

```
(regsub regex replacement string)
(regsub substring replacement string)
(regsub function replacement string)
```

Description:

The `regsub` function operates in two modes, depending on whether the first argument is a regular expression, or function.

If the first argument is a regular expression or string, then `regsub` searches *string* for multiple occurrences of non-overlapping matches for that *regex* or *substring*. A new string is constructed similar to *string* but in which each matching region is replaced with using *replacement* as follows.

The *replacement* object may be a character or a string, in which case it is simply taken to be the replacement for each match of the regular expression.

The *replacement* object may be a function of one argument, in which case for every match which is found, this function is invoked, with the matching piece of text as an argument. The function's return value is then taken to be the replacement text.

If the first argument is a function, then it is called, with *string* as its argument. The return value

must be either a range object (see the `rcons` function) which indicates the extent of *`string`* to be replaced, or else `nil` which indicates that no replacement is to take place.

Examples:

```
;; match every lowercase e or o, and replace by filtering
;; through the upcase-str function:

[regsub #/[eo]/ upcase-str "Hello world!"] -> "HEllO wOrld!"

;; Replace Hello with Goodbye:
(regsub #/Hello/ "Goodbye" "Hello world!") -> "Goodbye world!"

;; Same, as a simple substring match, rather than regex:
(regsub "Hello" "Goodbye" "Hello world!") -> "Goodbye world!"

;; Left-anchored replacement with r^ function:
(regsub (fr^ #/H/) "J" "Hello, hello!") -> "Jello, hello!"
```

9.40.7 Function `regexp`

Syntax:

```
(regexp obj)
```

Description:

The `regexp` function returns `t` if *`obj`* is a compiled regular-expression object. For any other object type, it returns `nil`.

9.40.8 Functions `trim-left` and `trim-right`

Syntax:

```
(trim-left {regex | prefix} string)
(trim-right {regex | suffix} string)
```

Description:

The `trim-left` and `trim-right` functions return a new string, equivalent to *`string`* with a leading or trailing portion removed.

If the first argument is a regular expression *`regex`*, then, respectively, `trim-left` and `trim-right` find a prefix or suffix of *`string`* which matches the regular expression. If there is no match, or if the match is empty, then *`string`* is returned. Otherwise, a copy of *`string`* is returned in which the matching characters are removed. If *`regex`* matches all of *`string`* then the empty string is returned.

If the first argument is a character string, then it is treated as an exact match for that sequence of characters. Thus, `trim-left` interprets that string as a *`prefix`* to be removed, and `trim-right` as a *`suffix`*. If *`string`* starts with *`prefix`*, then `trim-left` returns a copy of *`string`* with *`prefix`* removed. Otherwise, *`string`* is returned. Likewise, if *`string`* ends with *`suffix`*, then `trim-right` returns a copy of *`string`* with *`suffix`* removed. Otherwise, *`string`* is returned.

9.40.9 Function `regex-compile`

Syntax:

```
(regex-compile form-or-string [error-stream])
```

Description:

The `regex-compile` function takes the source code of a regular expression, expressed as a Lisp data structure representing an abstract syntax tree, or else a regular expression specified as a character string, and compiles it to a regular-expression object.

If *form-or-string* is a character string, it is parsed to an abstract syntax tree first, if by the `regex-parse` function. If the parse is successful (the result is not `nil`) then the resulting tree structure is compiled by a recursive call to `regex-compile`.

The optional *error-stream* argument is passed down to `regex-parse` as well as in the recursive call to `regex-compile`, if that call takes place.

If *error-stream* is specified, it must be a stream. Any error diagnostics are sent to that stream.

Examples:

```
;; the equivalent of #/[a-zA-Z0-9_]/
(regex-compile '(set (#\a . #\z) (#\A . #\Z)
                     (#\0 . #\9) #\_))

;; the equivalent of #/.*/ and #/.+/
(regex-compile '(0+ wild))
(regex-compile '(1+ wild))

;; #/a|b|c/
(regex-compile '(or (or #\a #\b) #\c))

;; string
(regex-compile "a|b|c")
```

9.40.10 Function `regex-source`

Syntax:

```
(regex-source regex)
```

Description:

The `regex-source` function returns the source code of compiled regular expression *regex*.

The source code isn't the textual notation, but the Lisp data structure representing the abstract syntax tree: the same representation as what is returned by `regex-parse`.

9.40.11 Function `regex-parse`

Syntax:

```
(regex-parse string [error-stream])
```

Description:

The `regex-parse` function parses a character string which contains a regular expression and turns it into a Lisp data structure (the abstract syntax tree representation of the regular expression).

The regular-expression syntax `#/RE/` produces the same structure, but as a literal which is

processed at the time **TXR** source code is read; the `regex-parse` function performs this parsing at run time.

If there are parse errors, the function returns `nil`.

The optional *`error-stream`* argument specifies a stream to which error messages are sent from the parser. By default, diagnostic output goes to the `*stdnull*` stream, which discards it. If *`error-stream`* is specified as `t`, then the diagnostic output goes to the `*stdout*` stream.

If `regex-parse` returns a non-`nil` value, that structure is then something which is suitable as input to `regex-compile`.

There is a small difference in the syntax accepted by `regex-parse` and the syntax of regular-expression literals. Any `/` (slash) characters occurring in any position within *`string`* are treated as ordinary characters, not as regular-expression delimiters. The call `(regex-parse "/a/")` matches three characters: a slash, followed by the letter "a", followed by another slash. Note that the slashes are not escaped.

Note: if a `regex-parse` call is written using a string literal as the *`string`* argument, then note that any backslashes which are to be processed by the regular expression must be doubled up, otherwise they belong to the string literal:

```
(regex-parse "\*")  ;; error, invalid string literal escape
(regex-parse "\\*") ;; correct: the \* literal match for *
```

The double backslash in the string literal produces a single backslash in the resulting string object that is processed by `regex-parse`.

9.40.12 Function `regex-optimize`

Syntax:

```
(regex-optimize regex-tree-syntax)
```

Description:

The `regex-compile` function accepts the source code of a regular expression, expressed as a Lisp data structure representing an abstract syntax tree, and calculates an equivalent structure in which certain simplifications have been performed, or in some cases substitutions which eliminate the dependence on derivative-based processing.

The *`regex-tree-syntax`* argument is assumed to be correct, as if it were produced by the `regex-parse` or `regex-from-trie` functions. Incorrect syntax produces unspecified results: an exception may be thrown, or some object may appear to be successfully returned.

Note: it is unnecessary to call this function to prepare the input for `regex-compile` because that function optimizes internally. However, the source code attached to a compiled regular-expression object is the original unoptimized syntax tree, and that is used for rendering the `#/.../` notation when the object is printed. If the syntax is passed through `regex-optimize` before `regex-compile`, the resulting object will have the optimized code attached to it, and subsequently render that way in printed form.

Examples:

```
;; a|b|c -> [abc]
(regex-optimize '(or #\a (or #\b #\c))) -> (set #\a #\b #\c)
```

```
;; (a|) -> a?
(regex-optimize '(or #\a nil)) -> (? #\a)
```

9.40.13 Function `read-until-match`

Syntax:

```
(read-until-match regex [stream [include-match]])
```

Description:

The `read-until-match` function reads characters from *`stream`*, accumulating them into a string, which is returned.

If an argument is not specified for *`stream`*, then the `*stdin*` stream is used.

The *`include-match`* argument is Boolean, indicating whether the delimiting text matched by *`regex`* is included in the returned string. It defaults to `nil`.

The accumulation of characters is terminated by a match on *`regex`*, the end of the stream, or an error.

This means that characters are read from the stream and accumulated while the stream has more characters available, and while its prefix does not match *`regex`*.

If *`regex`* matches the stream before any characters are accumulated, then an empty string is returned.

If the stream ends or an non-exception-throwing error occurs before any characters are accumulated, the function returns `nil`.

When the accumulation of characters terminates by a match on *`regex`*, the longest possible matching sequence of characters is removed from the stream. If *`include-match`* is true, that matching text is included in the returned string. Otherwise, it is discarded. The next available character in the stream is the first nonmatching character following the matched text. However, the next available character, as well as some number of subsequent characters, may originate from the stream's push-back buffer, rather than from the underlying operating system object, due to this function's internal use of the `unget-char` function. Therefore, the stream position, as would be reported by `seek-stream`, is unspecified.

9.40.14 Functions `scan-until-match` and `count-until-match`

Syntax:

```
(scan-until-match regex [stream])
(count-until-match regex [stream])
```

Description:

The functions `scan-until-match` and `count-until-match` read characters from *`stream`* until a match occurs in the stream for regular expression *`regex`*, the stream runs out of characters, or an error occurs.

If the stream runs out of characters, or a non-exception-throwing error occurs, before a match for *`regex`* is identified, these functions return `nil`.

If a match for *`regex`* occurs in *`stream`*, then `count-until-match` returns the number of characters that were read and discarded prior to encountering the first matching character. In the

same situation, the `scan-until-match` function returns a `cons` cell whose `car` holds the count of discarded characters, that being the same value as what would be returned by `count-until-match`, and whose `cdr` holds a character string that comprises the text matched by *regex*. The text matched by *regex* is as long as possible, and is removed from the stream. The next available character in the stream is the first nonmatching character following the matched text. However, the next available character, as well as some number of subsequent characters, may originate from the stream's push-back buffer, rather than from the underlying operating system object, due to these functions' internal use of the `unget-char` function. Therefore, the stream position, as would be reported by `seek-stream`, is unspecified.

9.40.15 Functions `m^$`, `m^` and `m$`

Syntax:

```
(m^$ regex [position] string)
(m^ regex [position] string)
(m$ regex [end-position] string)
```

Description:

These functions provide functionality similar to the `match-regst` and `match-regst-right` functions, but under alternative interfaces which are more convenient.

The `^` and `$` notation used in their names are an allusion to the regular-expression search-anchoring operators found in familiar POSIX utilities such as `grep`.

The *position* argument, if omitted, defaults to zero, so that the entire *string* is operated upon.

The *end-position* argument defaults to the length of *string*, so that the end position coincides with the end of the string.

If the *position* or *end-position* arguments are negative, they index backwards from the length of *string* so that -1 denotes the last character.

A value in either parameter which is excessively negative or positive, such that it indexes before the start of the string or exceeds its length results in a failed match and consequently `nil` being returned.

The `m^$` function tests whether the entire portion of *string* starting at *position* through to the end of the string is in the set of strings matched by *regex*. If this is true, then that portion of the string is returned. Otherwise `nil` is returned.

The `m^` function tests whether the portion of the *string* starting at *position* has a prefix which matches *regex*. If so, then this matching prefix is returned. Otherwise `nil` is returned.

The `m$` function tests whether the portion of *string* ending just before *end-position* has a suffix which matches *regex*. If so, then this matching suffix is returned. Otherwise `nil` is returned.

9.40.16 Functions `r^$`, `r^`, `r$` and `rr`

Syntax:

```
(r^$ regex [position] string)
(r^ regex [position] string)
(r$ regex [end-position] string)
```

```
(rr regex [position [from-end]] string)
```

Description:

The first three of these functions perform the same operations as, respectively, `m^$`, `m^` and `m$`, with the same argument conventions. They differ in return value. When a match is found, they return a range value indicating the extent of the matching substring within *string* rather than the matching substring itself.

The `rr` function performs the same operation as `range-regex` with different conventions with regard to argument order, harmonizing with those of the other three functions above.

The *position* argument, if omitted, defaults to zero, so that the entire *string* is operated upon.

The *end-position* argument defaults to the length of *string*, so that the end position coincides with the end of the string.

With one exception, a value in either parameter which is excessively negative or positive, such that it indexes before the start of the string or exceeds its length results in a failed match and consequently `nil` being returned. The exception is that the `rr` function permits a negative *position* value which refers before the start of the string; this is effectively treated as zero.

The *from-end* argument defaults to `nil`.

The `r^$` function tests whether the entire portion of *string* starting at *position* through to the end of the string is in the set of strings matched by *regex*. If this is true, then the matching range is returned, as a range object.

The `r^` function tests whether the portion of the *string* starting at *position* has a prefix which matches *regex*. If so, then the matching range is returned, as a range object. Otherwise `nil` is returned.

The `r$` function tests whether the portion of *string* ending just before *end-position* has a suffix which matches *regex*. If so, then the matching range is returned. Otherwise `nil` is returned.

The `rr` function searches *string* starting at *position* for a match for `regex`. If *from-end* is specified and true, the rightmost match is reported. If a match is found, it is reported as a range.

A regular expression which matches empty strings matches at the start position, and every other position, including the position just after the last character, coinciding with the length of *string*.

Except for the different argument order such that *string* is always the rightmost argument, the `rr` function is equivalent to the `range-regex` function, such that correspondingly named arguments have the same semantics.

9.40.17 Function `rra`

Syntax:

```
(rra regex [start [end]] string)
```

Description:

The `rra` function searches *string* between the *start* and *end* position for matches for the regular expression *regex*.

The matches are returned as a list of range objects.

The *start* argument defaults to zero, and *end* defaults to the length of the string (the position one past the last character).

Negative values of *start* and *end* indicate positions from the end of the string, such that -1 denotes the last character, -2 the second-to-last and so forth.

If *start* is so negative that it refers before the start of *string*, it is treated as zero. If this situation is true of the *end* argument, then the function returns `nil`.

If *start* refers to a character position beyond the length of *string* (two characters or more beyond the end of the string), then the function returns `nil`. If this situation is true of *end*, then *end* is curtailed to the string length.

The `rra` function returns all non-overlapping matches, including zero length matches. Zero length matches may occur before the first character of the string, or after the last character. If so, these are included.

9.40.18 Functions `f^$`, `f^` and `f$`

Syntax:

```
(f^$ regex [position])
(f^ regex [position])
(f$ regex [end-position])
```

Description:

These regular-expression functions do not directly perform regex operations. Rather, they each return a function of one argument which performs a regex operation.

The returned functions perform the same operations as, respectively, `m^$`, `m^` and `m$`.

The following equivalences nearly hold, except that the functions on the right side produced by `op` can accept two arguments when only `r` is curried, whereas the functions on the left take only one argument:

```
[f^$ r]     <-->  (op m^$ r)
[f^$ r p]   <-->  (op m^$ r p)
[f^ r]      <-->  (op m^ r)
[f^ r p]    <-->  (op m^ r p)
[f$ r]      <-->  (op m$ r)
[f$ r p]    <-->  (op m$ r p)
```

That is to say, `f^$` returns a function which binds *regex* and possibly the optional *position*. When this function is invoked, it must be given an argument which is a string. It performs the same operation as `m^$` being called on *regex* and possibly *position*. The same holds between `f^` and `m^`, and between `f$` and `m$`.

Examples:

```
;; produce list which contains only strings
;; beginning with "cat":
(keep-if (f^ #/cat/) '#"dog catalog cat fox catapult")
--> ("catalog" "cat" "catapult")
```

```
;; map all strings in a list to just their trailing
;; digits.
(mapcar (f$ #/\d*/) '#"a123 4 z bc465")
--> ("123" "4" "" "465")

;; check that all strings consist of digits after
;; the third position.
(all '#"ABC123 DFE45 12379" (f^$ #/\d*/ 3))
--> "79"  ; i.e. true
(all '#"ABC123 DFE45 12379A" (f^$ #/\d*/ 3))
--> nil
```

9.40.19 Functions `fr^$`, `fr^`, `fr$` and `frr`

Syntax:

```
(fr^$ regex [position])
(fr^ regex [position])
(fr$ regex [end-position])
(frr regex [[start-position] from-end])
```

Description:

These regular-expression functions do not directly perform regex operations. Rather, they each return a function of one argument which performs a regex operation.

The returned functions perform the same operations as, respectively, `r^$`, `r^`, `r$` and `rr`.

The following equivalences nearly hold, except that some of the functions on the right side produced by op `op` can accept additional arguments after the input string, whereas the functions on the left produced by `f^$` et al. accept only one parameter: the input string.

```
[fr^$ r]     <-->  (op m^$ r)
[fr^$ r p]   <-->  (op m^$ r p)
[fr^ r]      <-->  (op m^ r)
[fr^ r p]    <-->  (op m^ r p)
[fr$ r]      <-->  (op m$ r)
[fr$ r p]    <-->  (op m$ r p)
[frr r]      <-->  (op m$ r)
[frr r s]    <-->  (op m$ r s)
[frr r s fe] <-->  (op m$ r s fe)
```

That is to say, `fr^$` returns a function which binds *`regex`* and possibly the optional *`position`*. When this function is invoked, it must be given an argument which is a string. It performs the same operation as `r^$` being called on *`regex`* and possibly *`position`*, and the string. The same holds between `fr^` and `r^`, between `fr$` and `r$`, and between `frr` and `rr`.

Examples:

```
;; Remove leading digits from "123A456",
;; other than first digit:
(regsub (fr^ #/\d+/ 1) "" "123A456")
--> "1A456"
```

9.41 Hashing Library

A hash table is an object which retains an association between pairs of objects. Each pair consists of a key and a value. Given an object which is similar to a key in the hash table, it is possible to retrieve the corresponding value. Entries in a hash table are not ordered in any way, and lookup is facilitated by hashing: quickly mapping a key object to a numeric value which is then used to index into one of many buckets where the matching key will be found (if such a key is present in the hash table).

In addition to keys and values, a hash table contains a storage location which allows it to be associated with user data.

Important to the operation of a hash table is the criterion by which keys are considered same. By default, this similarity follows the `eql` function. A hash table will search for a stored key which is `eql` to the given search key. A hash table constructed with the `equal`-based property compares keys using the `equal` function instead.

In addition to storing key-value pairs, a hash table can have a piece of information associated with it, called the user data.

TXR hash tables contain a seed value which permutes the hashing operation, at least for keys of certain types. This feature, if the seed is randomized, helps to prevent software from being susceptible to hash collision denial-of-service attacks. However, by default, the seed is not randomized. Newly created hash tables for which a seed value is not specified take their seed value from the `*hash-seed*` special variable, which is initialized to zero. That includes hash tables created by parsing hash literal syntax. Security-sensitive programs requiring protection against collision attacks may use `gen-hash-seed` to create a randomized hash seed, and, depending on their specific need, either store that value in `*hash-seed*`, or pass the value to hash-table constructors like `make-hash`, or both. Note: randomization of hash seeding isn't a default behavior because it affects program reproducibility. The seed value affects the order in which keys are traversed, which can change the output of programs whose inputs have not changed, and whose logic is is otherwise deterministic.

A hash table can be traversed to visit all of the keys and data. The order of traversal bears no relation to the order of insertion, or to any properties of the key type.

During an open traversal, new keys can be inserted into a hash table or deleted from it while a a traversal is in progress. Insertion of a new key during traversal will not cause any existing key to be visited twice or to be skipped; however, it is not specified whether the new key will be traversed. Similarly, if a key is deleted during traversal, and that key has not yet been visited, it is not specified whether it will be visited during the remainder of the traversal. These remarks apply not only to deletion via `remhash` or the `del` operator, but also to wholesale deletion of all keys via `clearhash`.

The garbage collection of hash tables supports weak keys and weak values. If a hash table has weak keys, this means that from the point of view of garbage collection, that table holds only weak references to the keys stored in it. Similarly, if a hash table has weak values, it means that it holds a weak reference to each value stored. A weak reference is one which does not prevent the reclamation of an object by the garbage collector. That is to say, when the garbage collector discovers that the only references to some object are weak references, then that object is considered garbage, just as if it had no references to it. The object is reclaimed, and the weak references "lapse" in some way, which depends on what kind they are. Hash-table weak references lapse by entry removal. When an object used as a key in one or more weak-key hash tables becomes unreachable, those hash entries disappear. This happens even if the values are themselves reachable. Vice versa, when an object appearing as a value in one or more weak-value hash tables becomes unreachable, those entries disappear, even if the keys are reachable. When a hash table has both weak keys and weak values, then an the behavior is one of two possible semantics. Under the `or`-semantics, the hash table entry is removed if either the key or the value is unreachable. Under the `and`-semantics, the entry is removed only if both the key and value are unreachable.

If the keys of a weak-key hash table are reachable from the values, or if the values of a weak-key hash table are reachable from the keys, then the weak semantics is defeated for the affected entries: the hash table retains those entries as if it were an ordinary table. A hash table with both weak keys and values does not have this issue, regardless of its semantics.

An open traversal of a hash table is performed by the `maphash` function and the `dohash` operator. The traversal is open because code supplied by the program is evaluated for each entry.

The functions `hash-keys`, `hash-values`, `hash-pairs`, and `hash-alist` also perform an open traversal, because they return lazy lists. The traversal isn't complete until the returned lazy list is fully instantiated. In the meanwhile, the **TXR** program can mutate the hash table from which the lazy list is being generated.

Certain hash operations expose access to the internal key-value association entries of a hash table, which are represented as ordinary `cons` cells. Modifying the `car` field of such a cell potentially violates the integrity of the hash table; the behavior of subsequent lookup and insertion operations becomes unspecified.

Similarly, if an object is used as a key in an `equal`-based hash table, and that object is mutated in such a way that its equality to other objects under the `equal` function is affected or its hash value under `hash-equal` is altered, the behavior of subsequent lookup and insertion operations on the becomes unspecified.

9.41.1 Functions `make-hash` and `hash`

Syntax:

```
(make-hash weak-keys weak-vals
           equal-based [hash-seed])
(hash {:weak-keys | :weak-vals | :weak-or | :weak-and
       :eql-based | :equal-based |
       :eq-based | :userdata obj}*)
```

Description:

These functions construct a new hash table.

`make-hash` takes three mandatory Boolean arguments. The Boolean *`weak-keys`* argument specifies whether the hash table shall have weak keys. The *`weak-vals`* argument specifies whether it shall have weak values, and *`equal-based`* specifies whether it is `equal`-based.

If the *`weak-keys`* argument is one of the keywords `:weak-and` or `:weak-or` then the hash table shall have both weak keys and weak values, with the semantics implied by the keyword: `:weak-and` specifies `and`-semantics and `:weak-or` specifies `or`-semantics. The *`weak-vals`* argument is then ignored.

If both *`weak-keys`* and *`weak-vals`* are true, and *`weak-keys`* is not one of the keywords `:weak-and` or `:weak-or`, then the hash table has `or`-semantics.

The `hash` function defaults all three of these properties to false, and allows them to be overridden to true by the presence of keyword arguments.

The optional *`hash-seed`* parameter must be an integer, if specified. Its value perturbs the hashing function of the hash table, which affects `:equal-based` hash tables, when character strings and buffers are used as keys. If *`hash-seed`* is omitted, then the value of the `*hash-seed*` variable is used as the seed.

It is an error to attempt to construct an `equal`-based hash table which has weak keys.

The `hash` function provides an alternative interface. It accepts optional keyword arguments. The supported keyword symbols are: `:weak-keys`, `:weak-vals`, `:weak-and`, `:weak-or`, `:equal-based`, `:eql-based` `:eq-based` and `:userdata` which can be specified in any order to turn on the corresponding properties in the newly constructed hash table.

Only one of `:equal-based`, `:eql-based` and `:eq-based` may be specified. If specified, then the hash table uses `equal`, `eql` or `eq` equality, respectively, for considering two keys to be the same key. If none of these is specified, the `hash` function produces an `:equal-based` hash table by default.

If `:weak-keys`, `:weak-and` or `:weak-or` is specified, then `:equal-based` may not be specified.

At most one of `:weak-and` or `:weak-or` may be specified. If either of these is specified, then the `:weak-keys` and `:weak-vals` keywords are redundant and unnecessary.

If `:weak-keys` and `:weak-vals` are both specified, and `:weak-and` isn't specified, the situation is equivalent to `:weak-or`.

If `:userdata` is present, it must be followed by an argument value; that value specifies the user data for the hash table, which can be retrieved using the `hash-userdata` function.

Note: there doesn't exist a keyword for specifying the seed. This omission is deliberate. These hash construction keywords may appear in the hash literal `#H` syntax. A seed keyword would allow literals to specify their own seed, which would allow malicious hash literals to be crafted that perpetrate a hash collision attack against the parser.

9.41.2 Functions `hash-construct`, `hash-from-pairs` and `hash-from-alist`

Syntax:

```
(hash-construct hash-args key-val-pairs)
(hash-from-pairs key-val-pairs hash-arg*)
(hash-from-alist alist hash-arg*)
```

Description:

The `hash-construct` function constructs a populated hash in one step. The *`hash-args`* argument specifies a list suitable as an argument list in a call to the `hash` function. The *`key-val-pairs`* is a sequence of pairs, which are two-element lists representing key-value pairs.

A hash is constructed as if by a call to `(apply hash` *`hash-args`*`)`, then populated with the specified pairs, and returned.

The `hash-from-pairs` function is an alternative interface to the same semantics. The *`key-val-pairs`* argument is first, and the *`hash-args`* are passed as trailing variadic arguments, rather than a single list argument.

The `hash-from-alist` function is similar to `hash-from-pairs`, except that the *`alist`* argument specifies they keys and values as an association list. The elements of the list are `cons` cells, each of whose `car` is a key, and whose `cdr` is the value.

9.41.3 Function `hash-list`

Syntax:

```
(hash-list key-list hash-arg*)
```

Description:

The `hash-list` function constructs a hash as if by a call to `[apply hash hash-args]`, where *`hash-args`* is a list of the individual *`hash-arg`* variadic arguments.

The hash is then populated with keys taken from *`key-list`* and returned.

The value associated with each key is that key itself.

9.41.4 Function `hash-zip`

Syntax:

```
(hash-zip key-seq value-seq hash-arg*)
```

Description:

The `hash-zip` function constructs a hash as if by a call to `(apply hash hash-args)`, where *`hash-args`* is a list of the individual *`hash-arg`* variadic arguments.

The hash is then populated with keys taken from *`key-seq`* which are paired with values taken from from *`value-seq`*, and returned.

If *`key-seq`* is longer than *`value-seq`*, then the excess keys are ignored, and vice versa.

9.41.5 Function `hash-update`

Syntax:

```
(hash-update hash function)
```

Description:

The `hash-update` function replaces each value in *`hash`*, with the value of *`function`* applied to that value.

The return value is *`hash`*.

9.41.6 Function `hash-update-1`

Syntax:

```
(hash-update-1 hash key function [init])
```

Description:

The `hash-update-1` function operates on a single entry in the hash table.

If *`key`* exists in the hash table, then its corresponding value is passed into *`function`*, and the return value of *`function`* is then installed in place of the key's value. The value is then returned.

If *`key`* does not exist in the hash table, and no *`init`* argument is given, then `hash-update-1` does nothing and returns `nil`.

If *`key`* does not exist in the hash table, and an *`init`* argument is given, then *`function`* is applied to *`init`*, and then *`key`* is inserted into *`hash`* with the value returned by *`function`* as the datum. This value is also returned.

9.41.7 Functions `group-by` and `group-map`

Syntax:

```
(group-by by-fun sequence option*)
(group-map by-fun filter-fun sequence option*)
```

Description:

The `group-by` function produces a hash table from *sequence*, which is a list or vector. Entries of the hash table are not elements of *sequence*, but lists of elements of *sequence*. The function *by-fun* is applied to each element of *sequence* to compute a key. That key is used to determine which list the item is added to in the hash table.

The trailing arguments *option** if any, consist of the same keywords that are understood by the `hash` function, and determine the properties of the hash.

The `group-map` fun extends the semantics of `group-by` with a filtering step. It groups the elements of *sequence* in exactly the same manner, using *by-fun*. These lists of elements are then passed to *filter-fun* whose return values become the values associated with the hash table keys.

The effect of `group-map` may be obtained by a combination of `group-by` and `hash-update` according to the following equivalence:

```
(group-map bf ff seq)  <--> (let ((h (group-by bf seq)))
                                 (hash-update h ff))
```

Examples:

Group the integers from 0 to 10 into three buckets keyed on 0, 1 and 2 according to the modulo 3 congruence:

```
(group-by (op mod @1 3) (range 0 10))
-> #H(() (0 (0 3 6 9)) (1 (1 4 7 10)) (2 (2 5 8)))
```

Same as above, but associate the keys with the sums of the buckets:

```
[group-map (op mod @1 3) sum (range 0 10)]
-> #H(() (0 18) (1 22) (2 15))
```

9.41.8 Function `group-reduce`

Syntax:

```
(group-reduce hash classify-fun binary-fun seq
   [init-value [filter-fun]])
```

Description:

The `group-reduce` updates hash table *hash* by grouping and reducing sequence *seq*.

The function regards the hash table as being populated with keys denoting accumulator values. Missing accumulators which need to be created in the hash table are initialized with *init-value* which defaults to `nil`.

The function iterates over *seq* and treats each element according to the following steps:

1. Each element is mapped to a hash key through *classify-fun*.
2. The value associated with the hash key (the accumulator for that key) is retrieved. If it doesn't exist, *init-value* is used.
3. The function *binary-fun* is invoked with two arguments: the accumulator from step 2, and the original element from *seq*.
4. The resulting value from step 3 is stored back into the hash table under the key from step 2.

After the above processing, one more step is performed if the *filter-fun* argument is present. In this case, the hash table is destructively mapped through *filter-fun* before being returned. That is to say, every value in the hash table is projected through *filter-fun* and stored back in the table under the same key, as if by an invocation the `(hash-update hash filter-fun)` expression.

If `group-reduce` is invoked on an empty hash table, its net result closely resembles a `group-by` operation followed by separately performing a `reduce-left` on each value in the hash.

Examples:

Frequency histogram:

```
[group-reduce (hash) identity (do inc @1)
  "fourscoreandsevenyearsago" 0]
--> #H(() (#\a 3) (#\c 1) (#\d 1) (#\e 4) (#\f 1)
      (#\g 1) (#\n 2) (#\o 3) (#\r 3) (#\s 3)
      (#\u 1) (#\v 1) (#\y 1))
```

Separate the integers 1–10 into even and odd, and sum these groups:

```
[group-reduce (hash) evenp + (range 1 10) 0]
-> #H(() (t 30) (nil 25))
```

9.41.9 Functions `make-similar-hash` and `copy-hash`

Syntax:

```
(make-similar-hash hash)
(copy-hash hash)
```

Description:

The `make-similar-hash` and `copy-hash` functions create a new hash object based on the existing *hash* object.

`make-similar-hash` produces an empty hash table which inherits all of the attributes of *hash*. It uses the same kind of key equality, the same configuration of weak keys and values, and has the same user data (see the `set-hash-userdata` function).

The `copy-hash` also produces a hash table similar to *hash*, in the same way as `make-similar-hash`. However, rather than producing producing an empty hash table, it returns a duplicate table which has all the same elements as *hash*: it contains the same key and value objects.

9.41.10 Function `inhash`

Syntax:

```
(inhash hash key [init])
```

Description:

The `inhash` function searches hash table *`hash`* for *`key`*. If *`key`* is found, then it return the hash table's cons cell which represents the association between *`hash`* and *`key`*. Otherwise, it returns `nil`.

If argument *`init`* is specified, then the function will create an entry for *`key`* in *`hash`* whose value is that of *`init`*. The cons cell representing that association is returned.

Note: for as long as the *`key`* continues to exist inside *`hash`*. modifying the `car` field of the returned cons has ramifications for the logical integrity of the hash; doing so results in unspecified behavior for subsequent insertion and lookup operations.

Modifying the `cdr` field has the effect of updating the association with a new value.

9.41.11 Accessor `gethash`

Syntax:

```
(gethash hash key [alt])
(set (gethash hash key [alt]) new-value)
```

Description:

The `gethash` function searches hash table *`hash`* for key *`key`*. If the key is found then the associated value is returned. Otherwise, if the *`alt`* argument was specified, it is returned. If the *`alt`* argument was not specified, `nil` is returned.

A valid `gethash` form serves as a place. It denotes either an existing value in a hash table or a value that would be created by the evaluation of the form. The *`alt`* argument is meaningful when `gethash` is used as a place, and, if present, is always evaluated whenever the place is evaluated. In place update operations, it provides the initial value, which defaults to `nil` if the argument is not specified. For example `(inc (gethash h k d))` will increment the value stored under key `k` in hash table `h` by one. If the key does not exist in the hash table, then the value `(+ 1 d)` is inserted into the table under that key. The expression `d` is always evaluated, whether or not its value is needed.

If a `gethash` place is subject to a deletion, but doesn't exist, it is not an error. The operation does nothing, and `nil` is considered the prior value of the place yielded by the deletion.

9.41.12 Function `sethash`

Syntax:

```
(sethash hash key value)
```

Description:

The `sethash` function places a value into *`hash`* table under the given *`key`*. If a similar key already exists in the hash table, then that key's value is replaced by *`value`*. Otherwise, the *`key`* and *`value`* pair is newly inserted into *`hash`*.

The `sethash` function returns the *`value`* argument.

9.41.13 Function `pushhash`

Syntax:

```
(pushhash hash key element)
```

Description:

The `pushhash` function is useful when the values stored in a hash table are lists. If the given *`key`* does not already exist in *`hash`*, then a list of length one is made which contains *`element`*, and stored in *`hash`* table under *`key`*. If the *`key`* already exists in the hash table, then the corresponding value must be a list. The *`element`* value is added to the front of that list, and the extended list then becomes the new value under *`key`*.

The return value is Boolean. If true, indicates that the hash-table entry was newly created. If false, it indicates that the push took place on an existing entry.

9.41.14 Function `remhash`

Syntax:

```
(remhash hash key)
```

Description:

The `remhash` function searches *`hash`* for a key similar to the *`key`*. If that key is found, then that key and its corresponding value are removed from the hash table.

If the key is found and removal takes place, then the associated value is returned. Otherwise `nil` is returned.

9.41.15 Function `clearhash`

Syntax:

```
(clearhash hash)
```

Description:

The `clearhash` function removes all key-value pairs from *`hash`*, causing it to be empty.

If *`hash`* is already empty prior to the operation, then `nil`, is returned.

Otherwise an integer is returned indicating the number of entries that were purged from *`hash`*.

9.41.16 Function `hash-count`

Syntax:

```
(hash-count hash)
```

Description:

The `hash-count` function returns an integer representing the number of key-value pairs stored in *`hash`*.

9.41.17 Accessor `hash-userdata`

Syntax:

```
(hash-userdata hash)
(set (hash-userdata hash) new-value)
```

Description:

The `hash-userdata` function retrieves the user data object associated with *`hash`*.

A hash table can be created with user data using the `:userdata` keyword in a hash-table literal or in a call to the `hash` function, directly, or via other hash-constructing functions which take the hash construction keywords, such as `group-by`. If a hash table is created without user data, its user data is initialized to `nil`.

Because `hash-userdata` is an accessor, a `hash-userdata` form can be used as a place. Assigning a value to this place causes the user data of *`hash`* to be replaced with that value.

9.41.18 Function `get-hash-userdata`

Syntax:

```
(get-hash-userdata hash)
```

Description:

The `get-hash-userdata` function is a deprecated synonym for `hash-userdata`.

9.41.19 Function `set-hash-userdata`

Syntax:

```
(set-hash-userdata hash object)
```

Description:

The `set-hash-userdata` replaces, with the *`object`*, the user data object associated with *`hash`*.

9.41.20 Function `hashp`

Syntax:

```
(hashp object)
```

Description:

The `hashp` function returns `t` if the *`object`* is a hash table, otherwise it returns `nil`.

9.41.21 Function `maphash`

Syntax:

```
(maphash binary-function hash)
```

Description:

The `maphash` function successively invokes *`binary-function`* for each entry stored in *`hash`*. Each entry's key and value are passed as arguments to *`binary-function`*.

The function returns `nil`.

9.41.22 Functions `hash-revget` and `hash-keys-of`

Syntax:

```
(hash-revget hash value [testfun [keyfun]])
(hash-keys-of hash value [testfun [keyfun]])
```

Description:

The `hash-revget` function performs a reverse lookup on *hash*.

It searches through the entries stored in *hash* for an entry whose value matches *value*.

If such an entry is found, that entry's key is returned. Otherwise `nil` is returned.

If multiple matching entries exist, it is not specified which entry's key is returned.

The `hash-keys-of` function has exactly the same argument conventions, and likewise searches the *hash*. However, it returns a list of all keys whose values match *value*.

The *keyfun* function is applied to each value in *hash* and the resulting value is compared with *value*. The default *keyfun* is the `identity` function.

The comparison is performed using *testfun*.

The default *testfun* is the `equal` function.

9.41.23 Function `hash-invert`

Syntax:

```
(hash-invert hash [joinfun [unitfun hash-arg*]])
```

Description:

The `hash-invert` function calculates and returns an inversion of hash table *hash*. The values in *hash* become keys in the returned hash table. Conversely, the values in the returned hash table are derived from the keys.

The optional *joinfun* and *unitfun* arguments must be functions, if they are given. These functions determine the behavior of `hash-invert` with regard to duplicate values in *hash* which turn into duplicate keys. The *joinfun* function must be callable with two arguments, and *joinfun* must accept one argument. If *joinfun* is omitted, it defaults to the `identity*` function; *unitfun* defaults to `identity`.

The `hash-invert` function constructs a hash table as if by a call to the `hash` function, passing the *hash-arg* arguments which determine the properties of the newly created hash.

The new hash table is then populated by iterating over the key-value pairs of *hash* and inserting them as follows: The key from *hash* is turned into a value *v1* by invoking the *unitfun* function on it, and taking the return value. The value from *hash* is used as a key to perform a lookup in the new hash table. If no entry exists, then a new entry is created, whose value is *v1*. Otherwise if the entry already exists, then the value *v0* of that entry is combined with *v1* by calling the *joinfun* on the arguments *v0* and *v1*. The entry is updated with the resulting value.

The new hash table is then returned.

Examples:

```
;; Invert simple 1 to 1 table:

(hash-invert #H(() (a 1) (b 2) (c 3)))
--> #H(() (1 a) (2 b) (3 c))
```

```
;; Invert table such that the keys of duplicate values
;; are accumulated into lists:

[hash-invert #H(() (1 a) (2 a) (3 c) (5 c) (7 d)) append list]
--> #H(() (d (7)) (c (3 5)) (a (1 2)))

;; Invert table such that keys of duplicate values are summed:

[hash-invert #H(() (1 a) (2 a) (3 c) (5 c) (7 d)) +]
--> #H(() (d 7) (c 8) (a 3))
```

9.41.24 Functions `hash-eql` and `hash-equal`

Syntax:

```
(hash-eql object)
(hash-equal object [hash-seed])
```

Description:

These functions each compute an integer hash value from the internal representation of *object*, which satisfies the following properties. If two objects `A` and `B` are the same under the `eql` function, then `(hash-eql A)` and `(hash-eql B)` produce the same integer hash value. Similarly, if two objects `A` and `B` are the same under the `equal` function, then `(hash-equal A)` and `(hash-equal B)` each produce the same integer hash value. In all other circumstances, the hash values of two distinct objects are unrelated, and may or may not be the same.

Object of struct type may support custom hashing by way of defining an equality substitution via an `equal` method. See the Equality Substitution section under Structures.

The optional *hash-seed* value perturbs the hashing function used by `hash-equal` for strings and buffer objects. This seed value must be a nonnegative integer no wider than 64 bits: that is, in the range 0 to 18446744073709551615. If the value isn't specified, it defaults to zero. On systems with 32-bit addresses, only the low 32 bits of this value may be significant.

Effectively, each possible value of the significant part of the seed specifies a different hashing function. If two objects `A` and `B` are the same under the `equal` function, then `(hash-equal A S)` and `(hash-equal B S)` each produce the same integer hash value for any valid seed value `S`.

The value returned is a `fixnum` value, and may be negative. It may be any value in the range `fixnum-min` to `fixnum-max`.

9.41.25 Functions `hash-keys`, `hash-values`, `hash-pairs` and `hash-alist`

Syntax:

```
(hash-keys hash)
(hash-values hash)
(hash-pairs hash)
(hash-alist hash)
```

Description:

These functions retrieve the bulk key-value data of hash table *hash* in various ways. `hash-keys` retrieves a list of the keys. `hash-values` retrieves a list of the values. `hash-pairs` retrieves a list of pairs, which are two-element lists consisting of the key, followed by the value. Finally, `hash-alist` retrieves the key-value pairs as a Lisp association list: a list of cons cells

whose car fields are keys, and whose cdr fields are the values. Note that hash-alist returns the actual entries from the hash table, which are conses. Modifying the cdr fields of these conses constitutes modifying the hash values in the original hash table. Modifying the car fields interferes with the integrity of the hash table, resulting in unspecified behavior for subsequent hash insertion and lookup operations.

These functions all retrieve the keys and values in the same order. For example, if the keys are retrieved with hash-keys, and the values with hash-values, then the corresponding entries from each list pairwise correspond to the pairs in *hash*.

The list returned by each of these functions is lazy, and hence constitutes an open traversal of the hash table.

9.41.26 Operator dohash

Syntax:

```
(dohash (key-var value-var hash-form [result-form])
   body-form*)
```

Description:

The dohash operator iterates over a hash table. The *hash-form* expression must evaluate to an object of hash-table type. The *key-var* and *value-var* arguments must be symbols suitable for use as variable names. Bindings are established for these variables over the scope of the *body-form*s and the optional *result-form*.

For each element in the hash table, the *key-var* and *value-var* variables are set to the key and value of that entry, respectively, and each *body-form*, if there are any, is evaluated.

When all of the entries of the table are thus processed, the *result-form* is evaluated, and its return value becomes the return value of the dohash form. If there is no *result-form*, the return value is nil.

The *result-form* and *body-form*s are in the scope of an implicit anonymous block, which means that it is possible to terminate the execution of dohash early using (return *value*) or (return).

9.41.27 Functions hash-uni, hash-diff, hash-symdiff and hash-isec

Syntax:

```
(hash-uni hash1 hash2 [joinfun [map1fun [map2fun]]])
(hash-diff hash1 hash2)
(hash-symdiff hash1 hash2)
(hash-isec hash1 hash2 [joinfun])
```

Description:

These functions perform basic set operations on hash tables in a nondestructive way, returning a new hash table without altering the inputs. The arguments *hash1* and *hash2* must be compatible hash tables. This means that their keys must use the same kind of equality.

The resulting hash table inherits attributes from *hash1*, as if created by the make-similar-hash function. If *hash1* has userdata, the resulting hash table has the same userdata. If *hash1* has weak keys, the resulting table has weak keys, and so forth.

The hash-uni function performs a set union. The resulting hash contains all of the keys from

hash1 and all of the keys from *hash2*, and their corresponding values. If a key occurs both in *hash1* and *hash2*, then it occurs only once in the resulting hash. In this case, if the *joinfun* argument is not given, the value associated with this key is the one from *hash1*. If *joinfun* is specified then it is called with two arguments: the respective data items from *hash1* and *hash2*. The return value of this function is used as the value in the union hash. If *map1fun* is specified it must be a function that can be called with one argument. All values from *hash1* are projected through this function: the function is applied to each value, and the function's return value is used in place of the original value. Similarly, if *map2fun* is present, specifies a function through which values from *hash2* are projected.

The `hash-diff` function performs a set difference. First, a copy of *hash1* is made as if by the `copy-hash` function. Then from this copy, all keys which occur in `hash2` are deleted.

The `hash-symdiff` function performs a symmetric difference. A new hash is returned which contains all of the keys from *hash1* that are not in *hash2* and vice versa: all of the keys from *hash2* that are not in *hash1*. The keys carry their corresponding values from *hash1* and *hash2*, respectively.

The `hash-isec` function performs a set intersection. The resulting hash contains only those keys which occur both in *hash1* and *hash2*. If *joinfun* is not specified, the values selected for these common keys are those from *hash1*. If *joinfun* is specified, then for each key which occurs in both *hash1* and *hash2*, it is called with two arguments: the respective data items. The return value is then used as the data item in the intersection hash.

9.41.28 Functions `hash-subset` and `hash-proper-subset`

Syntax:

```
(hash-subset hash1 hash2)
(hash-proper-subset hash1 hash2)
```

Description:

The `hash-subset` function returns `t` if the keys in *hash1* are a subset of the keys in *hash2*.

The `hash-proper-subset` function returns `t` if the keys in *hash1* are a proper subset of the keys in *hash2*. This means that *hash2* has all the keys which are in *hash1* and at least one which isn't.

Note: the return value may not be mathematically meaningful if *hash1* and *hash2* use different equality. In any case, the actual behavior may be understood as follows. The implementation of `hash-subset` tests whether each of the keys in *hash1* occurs in *hash2* using their respective equalities. The implementation of `hash-proper-subset` applies `hash-subset` first, as above. If that is true, and the two hashes have the same number of elements, the result is falsified.

9.41.29 Functions `hash-begin`, `hash-reset`, `hash-next` and `hash-peek`

Syntax:

```
(hash-begin hash)
(hash-reset hash-iter hash)
(hash-next hash-iter)
(hash-peek hash-iter)
```

Description:

The `hash-begin` function returns a an iterator object capable of retrieving the entries in stored in *hash* one by one.

The `hash-reset` function changes the state of an existing iterator, such that it becomes prepared to retrieve the entries stored in the newly given *hash*, which may be the same one as the previously associated hash. In addition, `hash-reset` may be given a *hash* argument of `nil`, which dissociates it from its hash table.

The `hash-next` function's *hash-iter* argument is a hash iterator returned by `hash-begin`. If unvisited entries remain in *hash*, then `hash-next` returns the next one as a cons cell whose `car` holds the key and whose `cdr` holds the value. That entry is then considered visited by the iterator. If no more entries remain to be visited, `hash-next` returns `nil`. The `hash-next` function also returns `nil` if the iterator has been dissociated from a hash table by `hash-reset`.

The `hash-peek` function returns the same value that a subsequent call to `hash-next` will return for the same *hash-iter*, without changing the state of *hash-iter*. That is to say, if a cell representing a hash entry is returned, that entry remains unvisited by the iterator.

9.41.30 Macro `with-hash-iter`

Syntax:

```
(with-hash-iter (isym hash-form [ksym [vsym]])
   body-form*)
```

Description:

The `with-hash-iter` macro evaluates *body-form*s in an environment in which a lexically scoped function is visible.

The function is named by *isym* which must be a symbol suitable for naming functions with `flet`.

The *hash-form* argument must be a form which evaluates to a hash-table object.

Invocations of the function retrieve successive entries of the hash table as cons-cell pairs of keys and values. The function returns `nil` to indicate no more entries remain.

If either of the *ksym* or *vsym* arguments are present, they must be symbols suitable as variable names. They are bound as variables visible to *body-form*s, initialized to the value `nil`.

If *ksym* is specified, then whenever the function *isym* macro is invoked and retrieves a hash-table entry, the *ksym* variable is set to the key. If the function returns `nil` then the value of *ksym* is set to `nil`.

Similarly, if *vsym* is specified, then the function stores the retrieved hash value in that variable, or else sets the variable to `nil` if there is no next value.

9.41.31 Special variable `*hash-seed*`

Description:

The `*hash-seed*` special variable is initialized with a value of zero. Whenever a new hash table is explicitly or implicitly created, it takes its seed from the value of the `*hash-seed*` variable in the current dynamic environment.

The only situation in which `*hash-seed*` is not used when creating a new hash table is when `make-hash` is called with an argument given for the optional *hash-seed* argument.

Only `equal`-based hash tables make use of their seed, and only for keys which are strings and

buffers. The purpose of the seed is to scramble the hashing function, to make a hash table resistant to a type of denial-of-service attack, whereby a malicious input causes a hash table to be populated with a large number of keys which all map to the same hash-table chain, causing the performance to severely degrade.

The value of `*hash-seed*` must be a nonnegative integer, no wider than 64 bits. On systems with 32-bit addresses, only the least significant 32 bits of this value may be significant.

9.41.32 Function `gen-hash-seed`

Syntax:

```
(gen-hash-seed)
```

Description:

The `gen-hash-seed` function returns an integer value suitable for the `*hash-seed*` variable, or as the `hash-seed` argument of the `make-hash` and `hash-equal` functions.

The value is derived from the host environment, from information such as the process ID and time of day.

9.42 Search Tree Library

TXR Lisp provides binary search trees, which are objects of type `tree`. Trees have a printed notation denoted by the `#T` prefix. A tree may be constructed by invoking the `tree` function.

Binary search trees differ from hashes in that they maintain items in order. They also differ from hashes in that they store only elements, not key-value pairs. Every tree is associated with three *key abstraction functions*: It has a *key function* which is applied to the elements to map each one to a key. It also has a *less function* and *equal function* for comparing keys.

If these three functions are not specified, they respectively default to `identity`, `less` and `equal`, which means that the tree uses its elements as keys directly, and that they are compared using `less` and `equal`. Note: these default functions work for simple elements such as character strings or numbers, and also structures implementing *equality substitution*.

The elements are stored inside a tree using tree nodes, which are objects of type `tnode`, whose printed notation is introduced by the `#N` prefix.

Several tree-related functions take `tnode` objects as arguments or return `tnode` objects.

Trees may store duplicate elements. The `#T` literal syntax may freely specify duplicate elements. The `tree` constructor function specifies an initial sequence of elements to be populated into the newly constructed tree. If this initial sequence contains duplicate elements, they are preserved if the optional *`allow-dupes`* argument is true, otherwise only the rightmost member of any duplicate group appears in the tree.

The insertion functions `tree-insert` and `tree-insert-node` also overwrite duplicates by default, but optionally allow them. Duplicates are ordered by insertion: most recently inserted duplicate is rightmost. However, tree lookup chooses an unspecified duplicate.

9.42.1 Function `tnode`

Syntax:

```
(tnode key left right)
```

Description:

The `tnode` function allocates, initializes and returns a single tree node. A tree node has three fields *`key`*, *`left`* and *`right`*, which are accessed using the functions `key`, `left` and `right`.

9.42.2 Function `tnodep`

Syntax:

```
(tnodep value)
```

Description:

The `tnodep` function returns `t` if *`value`* is a tree node. Otherwise, it returns `nil`.

9.42.3 Accessors `key`, `left` and `right`

Syntax:

```
(key node)
(left node)
(right node)
(set (key node) new-key)
(set (left node) new-left)
(set (right node) new-right)
```

Description:

The `key`, `left` and `right` functions retrieve the corresponding fields of the *`node`* object, which must be of type `tnode`.

Forms based on the `key`, `left` and `right` symbol are defined as syntactic places. Assigning a value `v` to `(key n)` using the `set` operator, as in `(set (key n) v)`, is equivalent to `(set-key n v)` except that the value of the expression is `v` rather than `n`. Similar statements hold true for `left` and `right` in relation to `set-left` and `set-right`.

9.42.4 Functions `set-key`, `set-left` and `set-right`

Syntax:

```
(set-key node new-key)
(set-left node new-left)
(set-right node new-right)
```

Description:

The `set-key`, `set-left` and `set-right` functions replace the corresponding fields of *`node`* with new values.

The *`node`* argument must be of type `tnode`.

These functions all return *`node`*.

9.42.5 Function `copy-tnode`

Syntax:

```
(copy-tnode node)
```

Description:

The `copy-tnode` function creates a new `tnode` object, whose `key`, `left` and `right` fields are copied from `node`.

9.42.6 Function `tree`

Syntax:

```
(tree [elems
       [keyfun [lessfun [equalfun [allow-dupes]]]])
```

Description:

The `tree` function constructs and returns a new tree object. All arguments are optional.

The *`elems`* argument specifies a sequence of the elements to be stored in the tree. If the argument is absent or the sequence is empty, then an empty tree is created.

The *`keyfun`* argument specifies the function which is applied to every element to produce a key. If omitted, the tree object shall behave as if the `identity` function were used, taking the elements themselves to be keys.

The *`lessfun`* argument specifies the function by which two keys are compared for inequality. If omitted, the `less` function is used. A function used as *`lessfun`* should take two arguments, produce a Boolean result, and have ordering properties similar to the `less` function.

The *`equalfun`* argument specifies the function by which two keys are compared for equality. The default value is the `equal` function. A function used as *`equalfun`* should take two arguments, produce a Boolean result, and have the properties of an equivalence relation.

These three functions are collectively referred to as the tree's *key abstraction functions*.

The *`allow-dupes`* argument, which defaults to `nil`, is relevant if an *`elems`* sequence is specified containing some elements which which appear to be duplicates, according to the tree object's *`equalfun`* function. If *`allow-dupes`* is true then duplicates are preserved: the tree will have as many nodes as there are elements in the *`elems`* sequence. Moreover, the duplicates appear in the same relative order in the tree as they appear in the original *`elems`* sequence. If *`allow-dupes`* is false, then duplicates are suppressed: if any element appears more than once in *`elements`*, then only the last occurrence of that element appears in the tree.

Note: the `tree-insert` and `tree-insert-node` functions also has an optional argument indicating whether an duplicate insertion replaces an existing element.

Note: although the order of duplicate elements is preserved, when the `tree-lookup` function is used look up an key which is duplicated, the element which is retrieved is unspecified, and can change when the tree is reorganized due to insertions and deletions.

9.42.7 Function `treep`

Syntax:

```
(treep value)
```

Description:

The `treep` function returns `t` if *`value`* is a tree. Otherwise, it returns `nil`.

9.42.8 Function `tree-count`

Syntax:

```
(tree-count tree)
```

Description:

The `tree-count` function returns an integer indicating the number of nodes currently inserted into *tree*, which must be a search tree object.

9.42.9 Function `tree-insert-node`

Syntax:

```
(tree-insert-node tree node [allow-dupe])
```

Description:

The `tree-insert-node` function inserts an existing *node* object into a search tree.

The *tree* object must be of type `tree`, and *node* must be of type `tnode`.

The `key` field of the *node* object holds the element that is being inserted. The actual search key which is associated with this element is determined by applying *tree*'s *keyfun* to the *node*'s `key` value.

The *node* object must not currently be inserted into any existing tree. The values stored in the `left` and `right` fields of *node* are overwritten as required by the semantics of the insertion operation. Their original values are ignored.

The *allow-dupe* argument, defaulting to `nil`, is concerned with what happens if the tree already contains one or more nodes having a key equal to the *node*'s key. If *allow-dupe* is false, then *node* replaces an unspecified one of those existing nodes: that replaced node is deleted from the tree. Key equivalence is determined using tree's equality function (see the *equalfun* argument of the `tree` function). If *allow-dupe* is true, then the new node is inserted without replacing any node, and appears together with the existing duplicate or duplicates. Among the duplicates, the newly inserted node is the rightmost node in the tree order.

The `tree-insert-node` function returns the *node* argument.

9.42.10 Function `tree-insert`

Syntax:

```
(tree-insert tree elem [allow-dupe])
```

Description:

The `tree-insert` function inserts *elem* into *tree*.

The *tree* argument must be an object of type `tree`.

The *elem* value may be of any type which is semantically compatible with *tree*'s key abstraction functions.

The `tree-insert` function allocates a new `tnode` as if by invoking `(tnode elem nil nil)` function, and inserts that `tnode` as if by using the `tree-insert-node` function.

If one or more elements equal to *elem* already exist in the tree, then the behavior is determined by

the *allow-dupe* argument, which defaults to `nil`. The semantics of *allow-dupe* is as given in the description of `tree-insert-node`.

The `tree-insert` function returns the newly inserted `tnode` object.

9.42.11 Function `tree-lookup-node`

Syntax:

```
(tree-lookup-node tree key)
```

Description:

The `tree-lookup-node` searches *tree* for an element which matches *key*.

The *tree* argument must be an object of type `tree`.

The *key* argument may be a value of any type.

An element inside *tree* matches *key* if the tree's *keyfun* applied to that element produces a key value which is equal to *key* under the tree's *equalfun* function.

If such an element is found, then `tree-lookup-node` returns the tree node which contains that element as its *key* field.

If no such element is found, then `tree-lookup-node` returns `nil`.

If multiple nodes exist in the tree which have a matching key, it is unspecified which one of those nodes is retrieved.

9.42.12 Function `tree-lookup`

Syntax:

```
(tree-lookup tree key)
```

Description:

The `tree-lookup` function finds an element inside *tree* which matches the given *key*.

If the element is found, it is returned. Otherwise, `nil` is returned.

Note: the semantics of the `tree-lookup` function can be understood in terms of `tree-lookup-node`. A possible implementation is this:

```
(defun tree-lookup (tree key)
  (iflet ((node (tree-lookup-node tree key)))
    (key node)))
```

If the tree contains multiple elements which match *key*, it is unspecified which element is retrieved.

9.42.13 Function `tree-delete-node`

Syntax:

```
(tree-delete-node tree key)
```

Description:

The `tree-delete-node` function searches *`tree`* for an element which matches *`key`*.

The *`tree`* argument must be an object of type `tree`.

The *`key`* argument may be a value of any type which is semantically compatible with *`tree`*'s key abstraction functions.

If the matching element is found, then its node is removed from the tree, and returned.

Otherwise, if a matching element is not found, then `nil` is returned.

If more than one element exists inside *`tree`* which matches *`key`*, it is unspecified which node is deleted and returned.

9.42.14 Function `tree-delete`

Syntax:

```
(tree-delete tree key)
```

Description:

The `tree-delete` function tries to removes from *`tree`* the element which matches *`key`*.

If successful, it returns that element, otherwise it returns `nil`.

If more than one element exists inside *`tree`* which matches *`key`*, it is unspecified which one is deleted.

Note: the semantics of the `tree-delete` function can be understood in terms of `tree-delete-node`. A possible implementation is this:

```
(defun tree-delete (tree key)
  (iflet ((node (tree-delete-node tree key)))
    (key node)))
```

9.42.15 Function `tree-delete-specific-node`

Syntax:

```
(tree-delete-specific-node tree node)
```

Description:

The `tree-delete-specific-node` function searches *`tree`* to find the specific node given by the *`node`* argument. If *`node`* is inserted into the tree, then it is deleted, and returned.

If *`node`* is not found in the tree, then the tree is unchanged, and `nil` is returned.

Note: the search for *`node`* is informed by *`node`*'s key, for efficiency. However, if the tree contains duplicates of that key, then a linear search takes place among the duplicates.

9.42.16 Functions `tree-min-node` and `tree-min`

Syntax:

```
(tree-min-node tree)
```

```
(tree-min tree)
```

Description:

The `tree-min-node` function returns the node in *`tree`* which holds the lowest element. If the tree is empty, it returns `nil`.

The `tree-min` function returns the lowest element, or else `nil` if the tree is empty.

9.42.17 Functions `tree-del-min-node` and `tree-del-min`

Syntax:

```
(tree-del-min-node tree)
(tree-del-min tree)
```

Description:

The `tree-del-min-node` function returns the node in *`tree`* which has the lowest key, and removes that node from the tree. If the tree is empty, it returns `nil`.

The `tree-del-min` function returns the lowest element and removes it from the tree, or else `nil` if the tree is empty.

The following equivalence holds:

```
(tree-del-min tr)  <-->  (iflet ((node (tree-del-min-node tr)))
                           (key node))
```

Note: `tree-insert` together with `tree-del-min` provide the basis for using a tree as a priority queue. Elements are inserted into the queue using `tree-insert` and then removed in priority order using `tree-del-min`.

9.42.18 Function `tree-root`

Syntax:

```
(tree-root tree)
```

Description:

The `tree-root` function returns the root node of *`tree`*, which must be a `tree` object.

If *`tree`* is empty, then `nil` is returned.

9.42.19 Function `tree-clear`

Syntax:

```
(tree-clear tree)
```

Description:

The `tree-clear` function deletes all elements from *`tree`*, which must be a `tree` object.

If *`tree`* is already empty, then the function returns `nil`, otherwise it returns an integer which gives the count of the number of deleted nodes.

9.42.20 Function `copy-search-tree`

Syntax:

```
(copy-search-tree tree)
```

Description:

The `copy-search-tree` returns a new tree object which is a copy of *`tree`*.

The *`tree`* argument must be an object of type `tree`.

The returned object has the same key abstraction functions as *`tree`* and contains the same elements.

The nodes held inside the new tree are freshly allocated, but their key objects are shared with the original tree.

9.42.21 Function `make-similar-tree`

Syntax:

```
(make-similar-tree tree)
```

Description:

The `copy-search-tree` returns a new, empty search tree object.

The *`tree`* argument must be an object of type `tree`.

The returned object has the same key abstraction functions as *`tree`*.

9.42.22 Function `tree-begin`

Syntax:

```
(tree-begin tree [low-key [high-key]])
```

Description:

The `tree-begin` function returns a new object of type `tree-iter` which provides in-order traversal of nodes stored in *`tree`*.

The *`tree`* argument must be an object of type `tree`.

If the *`low-key`* argument is specified, then nodes with keys lesser than *`low-key`* are omitted from the traversal.

If the *`high-key`* argument is specified, then nodes with keys equal to or greater than *`high-key`* are omitted from the traversal.

The nodes are traversed by applying the `tree-next` function to the returned `tree-iter` object.

A `tree-iter` object is iterable.

Example:

```
(collect-each ((el (tree-begin #T(() 1 2 3 4 5)
                               2 5)))
```

```
  (* 10 el))
--> (20 30 40)
```

9.42.23 Function `tree-reset`

Syntax:

```
(tree-reset iter tree [low-key [high-key]])
```

Description:

The `tree-reset` functions is closely analogous to `tree-begin`.

The *`iter`* argument must be an existing `tree-iter` object, previously returned by a call to `tree-begin`.

Regardless of its current state, the *`iter`* object is re-initialized to traverse the specified *`tree`* with the specified parameters, and is then returned.

The `tree-reset` function prepares *`iter`* to traverse in the same manner as would new iterator returned by `tree-begin` for the specified *`tree`*, *`low-key`* and *`high-key`* arguments.

9.42.24 Functions `tree-next` and `tree-peek`

Syntax:

```
(tree-next iter)
(tree-peek iter)
```

Description:

The `tree-next` and `tree-peek` function returns the next node in sequence from the tree iterator *`iter`*. The iterator must be an object of type `tree-iter`, returned by the `tree-begin` function.

If there are no more nodes to be visited, these functions `nil`.

If, during the traversal of a tree, nodes are inserted or deleted, the behavior of `tree-next` and `tree-peek` on `tree-iter` objects that were obtained prior to the insertion or deletion is not specified. An attempt to complete the iteration may not successfully visit all keys that should be visited.

The `tree-next` function changes the state of the iterator. If `tree-next` is invoked repeatedly on the same iterator, it returns successive nodes of the tree.

If `tree-peek` is invoked more than once on the same iterator without any intervening calls to `tree-next`, it returns the same node; it does not appear to change the state of the iterator and therefore does not advance through successive nodes.

9.42.25 Function `sub-tree`

Syntax:

```
(sub-tree tree [from-key [to-key]])
```

Description:

The `sub-tree` function selects elements from *`tree`*, which must be a search tree.

If *`from-key`* is specified, then elements lesser than *`from-key`* are omitted from the selection.

If *to-key* is specified, the elements greater than or equal to *to-key* are omitted from the selection.

A list of the selected elements is returned, in which the elements appear in the same order as they do in *tree*.

9.42.26 Function `copy-tree-iter`

Syntax:

```
(copy-tree-iter iter)
```

Description:

The `copy-tree-iter` function creates and returns a duplicate of the *iter* object, which must be a tree iterator returned by `tree-begin`.

The returned object has the same state as the original; it references the same traversal position in the same tree. However, it is independent of the original. Calls to `tree-next` on the original have no effect on the duplicate and vice versa.

9.42.27 Function `replace-tree-iter`

Syntax:

```
(replace-tree-iter dest-iter src-iter)
```

Description:

The `replace-tree-iter` function causes the tree iterator *dest-iter* to be in the same state as *src-iter*.

Both *dest-iter* and *src-iter* must be tree iterator objects returned by `tree-begin`.

The contents of *dest-iter* are updated such that it now references the same tree as *src-iter*, at the same position.

The *dest-iter* argument is returned.

9.42.28 Special variable `*tree-fun-whitelist*`

Description:

The `*tree-fun-whitelist*` variable holds a list of function names that may be used in the `#T` tree literal syntax as the *keyfun*, *lessfun* or *equalfun* operations of a tree. The initial value of this variable is a list which holds at least the following three symbols: `identity`, `less` and `equal`.

The application may change the value of this variable, or dynamically bind it, in order to allow `#T` literals to be processed which specify functions other than these three.

9.43 Partial Evaluation and Combinators

9.43.1 Macros `op` and `do`

Syntax:

```
(op form+)
(do oper form*)
```

Description:

Like the `lambda` operator, the `op` macro denotes an anonymous function. Unlike `lambda`, the arguments of the function are implicit, or optionally specified within the expression, rather than as a formal parameter list which precedes a body.

The *form* arguments of `op` are implicitly turned into a DWIM expression, which means that argument evaluation follows Lisp-1 rules. (See the `dwim` operator).

The argument forms of `op` are arbitrary expressions, within which special conventions are permitted regarding the use of certain implicit variables:

`@`*num* A number preceded by a `@` is, syntactically, a metanumber. If it appears inside `op` as an expression, it behaves as a positional argument, whose existence it implies. For instance `@2` means that the function shall have at least two arguments, the second argument of which is be substituted in place of the `@2`. `op` generates a function which has a number of required arguments equal to the highest value of *num* appearing in a `@`*num* construct in the body. For instance `(op car @3)` generates a three-argument function (which passes its third argument to `car`, returning the result, and ignores its first two arguments). There is no way to use `op` to generate functions which have optional arguments. The positional arguments are mutable; they may be assigned.

`@rest` If the meta-symbol `@rest` appears in the `op` syntax as an expression, it explicitly denotes and evaluates to the list of trailing arguments. Like the metanumber positional arguments, it may be assigned.

`@rec` If the meta-symbol `@rec` appears in the `op` syntax as an expression, it denotes a mutable variable which is bound to the function itself which is generated by that `op` expression.

`@(rec ...)`
If this syntax appears inside `op`, it specifies a recursive call to the function.

Functions generated by `op` are always variadic; they always take additional arguments after any required ones, whether or not the `@rest` syntax is used.

If the body does not contain any `@`*num* or `@rest` syntax, then `@rest` is implicitly inserted. What this means is that, for example, since the form `(op foo)` does not contain any implicit positional arguments like `@1`, and does not contain `@rest`, it is actually a shorthand for `(op foo . @rest)`: a function which applies `foo` to all of its arguments. If the body does contain at least one `@`*num* or `@rest`, then `@rest` isn't implicitly inserted. The notation `(op foo @1)` denotes a function which takes any number of arguments, and ignores all but the first one, which is passed to `foo`.

The `do` operator is similar to `op`, with the following three differences:

1. The first argument of `do`, namely *oper*, is an operator. This argument is not processed for the presence of implicit variables. Thus for instance `(do @1 ...)` is invalid. By contrast, `(op @1 ...)` is possible, and makes sense under the right circumstances. The *oper* argument may be the name of a macro or special operator, whereas `op` doesn't support the invocation of macros or special operators. For instance `(do let ((x @1)) (+ x 1))` is possible.
2. The *form* arguments of `do` are not implicitly treated as DWIM expressions, but as ordinary expressions.
3. When `do` syntax doesn't contain any references to implicit variables (metanumbers or `@rest`) then a variadic function is generated which requires one argument. That argument is added to the form. Thus for instance `(do set x)` effectively serves as a shorthand for `(do set x @1)`. The corresponding defaulting behavior in `op` is that a

variadic function is generated which requires no arguments. All of the available arguments are applied. Thus `(op f x)` is effectively a shorthand for `(op f x . @rest)`.

Because it accepts operators, `do` can be used with imperative constructs which are not functions, like `set`. For example, `(do set x)` produces an anonymous function which, if called with one argument, stores that argument into *x*.

The actions of `op` and `do` can be understood by the following examples, which convey how the syntax is rewritten to lambda. However, note that the real translator uses generated symbols for the arguments, which are not equal to any symbols in the program.

```
(op) -> invalid

(op +) -> (lambda rest [+ . rest])

(op + foo) -> (lambda rest [+ foo . rest])

(op @1 . @rest) -> (lambda (arg1 . rest) [arg1 . @rest])

(op @1 @rest) -> (lambda (arg1 . rest) [arg1 @rest])

(op @1 @2) -> (lambda (arg1 arg2 . rest) [arg1 arg2])

(op foo @1 (@2) (bar @3)) -> (lambda (arg1 arg2 arg3 . rest)
                                [foo arg1 (arg2) (bar arg3)])

(op foo @rest @1) -> (lambda (arg1 . rest) [foo rest arg1])

(do + foo) -> (lambda (arg1 . rest) (+ foo arg1))

(do @1 @2) -> (lambda (arg1 arg2 . rest) (@1 arg2)) ;; invalid!

(do foo @rest @1) -> (lambda (arg1 . rest) (foo rest arg1))
```

Note that if argument `@n` appears in the syntax, it is not necessary for arguments `@1` through `@n-1` to appear. The function will have `n` arguments:

```
(op @3) -> (lambda (arg1 arg2 arg3 . rest) [arg3])
```

The `op` and `do` operators can be nested, in any combination. This raises the question: if an expression like `@1`, `@rest` or `@rec` occurs in an `op` that is nested within an `op`, what is the meaning?

An expression with a single `@` always belongs with the innermost `op` or `do` operator. So for instance `(op (op @1))` means that an `(op @1)` expression is nested within an outer `op` expression that contains no references to its implicit variables. The `@1` belongs to the inner `op`.

There is a way for an inner `op` to refer to the implicit variables of an outer one. This is expressed by adding an extra `@` prefix for every level of escape. For example in `(op (op @@1))` the `@@1` belongs to the outer `op`: it is the same as `@1` appearing in the outer `op`. That is to say, in the expression `(op @1 (op @@1))`, the `@1` and `@@1` are the same thing: both are parameter 1 of the lambda function generated by the outer `op`. By contrast, in the expression `(op @1 (op @1))` there are two different parameters: the first `@1` is argument of the outer function, and the second `@1` is the first argument of the inner function. If there are three levels of nesting, then three `@` meta-prefixes are needed to insert a parameter from the outermost `op` into the innermost `op`.

Note that the implicit variables belonging to an `op` can be used in the dot position of a function call, such as:

```
[(op list 1 . @1) 2] -> (1 . 2)
```

This is a consequence of the special transformations described in the paragraph **Dot Position in Function Calls** in the subsection **Additional Syntax** of the **TXR Lisp** section.

The `op` syntax works in conjunction with quasiliterals which are nested within it. The metanumber notation as well as `@rest` are recognized without requiring an additional `@` escape, which is effectively optional:

```
(apply (op list `@1-@rest`) '(1 2 3)) -> "1-2 3"

(apply (op list `@@1-@@rest`) '(1 2 3)) -> "1-2 3"
```

Though they produce the same result, the above two examples differ in that `@rest` embeds a metasymbol into the quasiliteral structure, whereas `@@rest` embeds the Lisp expression `@rest` into the quasiliteral. Either way, in the scope of `op`, `@rest` undergoes the macro-expansion which renames it to the machine-generated function argument symbol of the implicit function denoted by the `op` macro form.

This convenient omission of the `@` character isn't supported for reaching the arguments of an outer `op` from a quasiliteral within a nested `op`:

```
;; To reach @@1, @@@1 must be written.
;; @@1 Lisp expression introduced by @.
(op ... (op ... `@@@1`))
```

Because the `do` macro may be applied to operators, it is possible to apply it to itself, as well as to `op`, as in the following example:

```
[[[[(do do do op list) 1] 2] 3] 4] -> (1 2 3 4)
```

The chained application associates right-to-left: the rightmost `do` is applied to `op`; the second rightmost `do` is applied to the rightmost one and so on. The effect is that partial application has been achieved. The value `1` is passed to the resulting function, which returns another function which takes the next argument. Finally, all these chained argument values are passed to `list`.

Each `do`/`op` level is processed independently. The following examples show how the list may be permuted into several different orders by referring to an implicit argument at various levels of nesting, making it the first argument of `list`. The unmentioned arguments implicitly follow, in order. This works because mentioning the argument explicitly means that its corresponding `do` operator no longer inserts its argument implicitly into body of the function which it generates:

```
[[[[(do do do op list @1) 1] 2] 3] 4] -> (4 1 2 3)
[[[[(do do do op list @@1) 1] 2] 3] 4] ->(3 1 2 4)
[[[[(do do do op list @@@1) 1] 2] 3] 4] -> (2 1 3 4)
[[[[(do do do op list @@@@1) 1] 2] 3] 4] -> (1 2 3 4))
```

The following example mentions all arguments at every `do`/`op` nesting level, thereby explicitly establishing the order in which they are passed to `list`:

```
[[[[(do do do op list @1 @@1 @@@1 @@@@1) 1] 2] 3] 4] -> (4 3 2 1)
```

Examples:

```
(let ((c 0))
  (mapcar (op cons (inc c)) '(a b c)))
--> ((1 . a) (2 . b) (3 . c))

(reduce-left (op + (* 10 @1) @2) '(1 2 3)) --> 123
```

9.43.2 Macro `lop`

Syntax:

```
(lop form+)
```

Description:

The `lop` macro is variant of `op` with special semantics.

The *`form`* arguments support the same notation as those of the `op` operator.

If only one *`form`* is given then `lop` is equivalent to `op`.

If two or more *`form`* arguments are present, then `lop` generates a variadic function which inserts all of its trailing arguments between the first and second *`form`*s.

That is to say, trailing arguments coming into the anonymous function become the left arguments of the function or function-like object denoted by the first *`form`* and the remaining *`form`*s give additional arguments. Hence the name `lop`, which stands for "left-inserting `op`".

This left insertion of the trailing arguments takes place regardless of whether `@rest` occurs in any *`form`*.

The *`form`* syntax determines the number of required arguments of the generated function, according to the highest-valued meta-number. The trailing arguments which are inserted into the left position are any arguments in excess of the required arguments.

The `lop` macro's expansion can be understood via the following equivalences, except that in the real implementation, the symbols `rest` and `arg1` through `arg3` are replaced with hygienic, unique symbols.

```
(lop f)  <-->  (op f)   <-->   (lambda (. rest) [f . rest])

(lop f x y)  <-->  (lambda (. rest)
                     [apply f (append rest [list x y])])

(lop f x @3 y)  <-->  (lambda (arg1 arg2 arg3 . rest)
                        [apply f
                                (append rest
                                        [list x arg3 y])])
```

Examples:

```
(mapcar (lop list 3) '(a b c)) --> ((a 3) (b 3) (c 3))

(mapcar (lop list @1) '(a b c)) --> ((a) (b) (c))

(mapcar (lop list @1) '(a b c) '(d e f))
```

```
--> ((d a) (e b) (f c))
```

9.43.3 Macro `ldo`

Syntax:

```
(ldo oper form*)
```

Description:

The `ldo` macro provides a shorthand notation for uses of the `do` macro which inserts the first argument of the anonymous function as the leftmost argument of the specified operator.

The `ldo` syntax can be understood in terms of these equivalences:

```
(ldo f)  <-->  (do f @1)
(ldo f x)  <-->  (do f @1 x)
(ldo f x y)  <-->  (do f @1 x y)
(ldo f x @2 y)  <-->  (do f @1 x @2 y)
```

The implicit argument `@1` is always inserted as the leftmost argument of the operator specified by the first form.

Example:

```
;; push elements of l1 onto l2.
(let ((l1 '(a b c)) l2)
  (mapdo (ldo push l2) l1)
  l2)
--> (c b a)
```

9.43.4 Macros `ap`, `ip`, `ado` and `ido`

Syntax:

```
(ap form+)
(ip form+)
(ado form+)
(ido form+)
```

Description:

The `ap` macro is based on the `op` macro and has identical argument conventions.

The `ap` macro analyzes its arguments and produces a function *f*, in exactly the same same way as the `op` macro. However, instead of returning *f*, directly, it returns a different function *g*, which is a one-argument function that accepts a list. The list specifies arguments to which *g* applies *f*, and then returns the resulting value.

In other words, the following equivalence holds:

```
(ap form ...) <--> (apf (op form ...))
```

The `ap` macro nests properly with `op` and `do`, in any combination, in regard to the `...@@n` notation.

The `ip` macro is similar to the `ap` macro, except that it is based on the semantics of the function

`iapply` rather than `apply`, according to the following equivalence:

```
(ip form ...) <--> (ipf (op form ...))
```

The `ado` and `ido` macros are related to do macro in the same way that `ap` and `ip` are related to `op`. They produce a one-argument function which works as if by applying the function generated by do to its its own arguments, according to the following equivalence:

```
(ado form ...) <--> (apf (do form ...))

(ido form ...) <--> (ipf (do form ...))
```

See also: the `apf` and `ipf` functions.

Example:

```
;; Take a list of pairs and produce a list in which those pairs
;; are reversed.

(mapcar (ap list @2 @1) '((1 2) (a b)))   ->   ((2 1) (b a))
```

9.43.5 Macros `opip` and `oand`

Syntax:

```
(opip clause*)
(oand clause*)
```

Description:

The `opip` and `oand` operators make it possible to chain together functions which are expressed using the `op` syntax. (See the `op` operator for more information).

Both macros perform the same transformation except that `opip` translates its arguments to a call to the `chain` function, whereas `oand` translates its arguments in the same way to a call to the `chand` function.

More precisely, these macros perform the following rewrites:

```
(opip arg1 arg2 ... argn) -> [chain {arg1} {arg2} ... {argn}]
(oand arg1 arg2 ... argn) -> [chand {arg1} {arg2} ... {argn}]
```

where the above `{arg}` notation denotes the following transformation applied to each argument:

```
;; these specific form patterns are left untransformed:

(dwim ...)      -> (dwim ...)
[...]           -> [...]
(qref ...)      -> (qref ...)
(uref ...)      -> (uref ...)
(op ...)        -> (op ...)
(do ...)        -> (do ...)
(lop ...)       -> (lop ...)
(ldo ...)       -> (ldo ...)
(ap ...)        -> (ap ...)
(ip ...)        -> (ip ...)
```

```
(ado ...)       -> (ado ...)
(ido ...)       -> (ido ...)
(ret ...)       -> (ret ...)
(aret ...)      -> (aret ...)
.slot           -> .slot
.(method ...)   -> .(method ...)
atom            -> atom

;; other compound forms are transformed like this:

(function ...) -> (op function ...)
(operator ...) -> (do operator ...)
(macro ...)    -> (do macro ...)
```

In other words, compound forms whose leftmost symbol is a macro or operator are translated to the `do` notation. Compound forms denoting function calls are translated to the `op` notation. Compound forms which are `dwim` invocations, either explicit or via the DWIM brackets notation, are used without transformation. Used without transformation also are forms denoting struct slot access, either explicitly using `uref` or `qref` or the respective dot notations, forms which invoke any of the `do` family of operators, as well as any atom forms.

Note: the `opip` and `oand` macros use their macro environment in determining whether a form is a macro call, thereby respecting lexical scoping.

Note: an `opip` form with no arguments specifies a function which returns `nil`, which follows from a documented property of the `chain` function.

Example:

Take each element from the list `(1 2 3 4)` and multiply it by three, then add 1. If the result is odd, collect that into the resulting list:

```
(mappend (opip (* 3)
               (+ 1)
               [iff oddp list])
         (range 1 4))
```

The above is equivalent to:

```
(mappend (chain (op * 3)
                (op + 1)
                [iff oddp list])
         (range 1 4))
```

The `(* 3)` and `(+ 1)` terms are rewritten to `(op * 3)` and `(op + 1)`, respectively, whereas `[iff oddp list]` is passed through untransformed.

9.43.6 Macro `flow`

Syntax:

(`flow` *form opip-arg**)

Description:

The `flow` macro passes the value of *form* through the processing stages described by the *opip-*

arg arguments, yielding the resulting value.

The *opip-arg* arguments follow the semantics of the `opip` macro.

The following equivalence holds:

```
(flow x ...)  <-->  [(opip ...) x]
```

That is to say, `flow` is equivalent to the application of an `opip`-generated function to the value of *form*.

Note: if there are no *opip-arg* arguments, then `flow` evaluates the `x` argument and returns `nil`; which follows from the behavior of `opip` when that operator is invoked with no arguments.

Examples:

```
(flow 1 (+ 2) (* 3) (cons 0)) -> (0 . 9)

(flow "abc" (upcase-str) (regsub #/B/ "ZTE")) -> "AZTEC"
```

9.43.7 Macro `ret`

Syntax:

```
(ret form)
```

Description:

The `ret` macro's *form* argument is treated similarly to the second and subsequent arguments of the `op` operator.

The `ret` macro produces a function which takes any number of arguments, and returns the value specified by *form*.

form can contain `op` meta syntax like `@n` and `@rest`.

The following equivalence holds:

```
(ret x) <--> (op identity* x))
```

Thus the expression `(ret @2)` returns a function similar to `(lambda (x y . z) y)`, and the expression `(ret 42)` returns a function similar to `(lambda (. rest) 42)`.

9.43.8 Macro `aret`

Syntax:

```
(aret form)
```

Description:

The `aret` macro's *form* argument is treated similarly to the second and subsequent arguments of the `op` operator.

The `aret` macro produces a function which takes any number of arguments, and returns the value specified by *form*.

form can contain `ap` meta syntax like `@n` and `@rest`.

The following equivalence holds:

```
(aret x) <--> (ap identity* x))
```

Thus the expression `(aret @2)` returns a function similar to `(lambda (. rest) (second rest))`, and the expression `(aret 42)` returns a function similar to `(lambda (. rest) 42)`.

9.43.9 Function `dup`

Syntax:

```
(dup func)
```

Description:

The `dup` function returns a one-argument function which calls the two-argument function *`func`* by duplicating its argument.

Example:

```
;; square the elements of a list
(mapcar [dup *] '(1 2 3)) -> (1 4 9)
```

9.43.10 Function `flipargs`

Syntax:

```
(flipargs func)
```

Description:

The `flipargs` function returns a two-argument function which calls the two-argument function *`func`* with reversed arguments.

9.43.11 Functions `chain` and `chand`

Syntax:

```
(chain func*)
(chand func*)
```

Description:

The `chain` function accepts zero or more functions as arguments, and returns a single function, called the chained function, which represents the chained application of those functions, in left-to-right order.

If `chain` is given no arguments, then it returns a variadic function which ignores all of its arguments and returns `nil`.

Otherwise, the first function may accept any number of arguments. The second and subsequent functions, if any, must accept one argument.

The chained function can be called with an argument list which is acceptable to the first function. Those arguments are in fact passed to the first function. The return value of that call is then passed to the second function, and the return value of that call is passed to the third function and so on. The final return value is returned to the caller.

The `chand` function is similar, except that it combines the functionality of `andf` into chaining.

The difference between `chain` and `chand` is that `chand` immediately terminates and returns `nil` whenever any of the functions returns `nil`, without calling the remaining functions.

Example:

```
(call [chain + (op * 2)] 3 4) -> 14
```

In this example, a two-element chain is formed from the + function and the function produced by `(op * 2)` which is a one-argument function that returns the value of its argument multiplied by two. (See the definition of the `op` operator).

The chained function is invoked using the `call` function, with the arguments `3` and `4`. The chained evaluation begins by passing `3` and `4` to +, which yields `7`. This `7` is then passed to the `(op * 2)` doubling function, resulting in `14`.

A way to write the above example without the use of the DWIM brackets and the op operator is this:

```
(call (chain (fun +) (lambda (x) (* 2 x))) 3 4)
```

9.43.12 Function `juxt`

Syntax:

```
(juxt func*)
```

Description:

The `juxt` function accepts a variable number of arguments which are functions. It combines these into a single function which, when invoked, passes its arguments to each of these functions, and collects the results into a list.

Note: the `juxt` function can be understood in terms of the following reference implementation:

```
(defun juxt (funcs)
  (lambda (. args)
    (mapcar (lambda (fun)
              (apply fun args))
            funcs)))
```

The `callf` function generalizes `juxt` by allowing the combining function to be specified.

Example:

```
;; separate list (1 2 3 4 5 6) into lists of evens and odds,
;; which end up juxtaposed in the output list:

[(op [juxt keep-if remove-if] evenp)
 '(1 2 3 4 5 6)] -> ((2 4 6) (1 3 5))

;; call several functions on 1, collecting their results:
[[juxt (op + 1) (op - 1) evenp sin cos] 1]'
-> (2 0 nil 0.841470984807897 0.54030230586814)
```

9.43.13 Functions `andf` and `orf`

Syntax:

```
(andf func*)
(orf func*)
```

Description:

The `andf` and `orf` functions are the functional equivalent of the `and` and `or` operators. These functions accept multiple functions and return a new function which represents the logical combination of those functions.

The input functions should have the same arity. Failing that, there should exist some common argument arity with which each of these can be invoked. The resulting combined function is then callable with that many arguments.

The `andf` function returns a function which combines the input functions with a short-circuiting logical conjunction. The resulting function passes its arguments to the input functions successively, in left-to-right order. As soon as any of the functions returns `nil`, then `nil` is returned and the remaining functions are not called. If none of the functions return `nil`, then the value returned by the last function is returned. If the list of functions is empty, then `t` is returned. That is, `(andf)` returns a function which accepts any arguments and returns `t`.

The `orf` function returns a function which combines the input functions with a short-circuiting logical disjunction. The resulting function passes its arguments to the input functions successively, in left-to-right order. As soon as any of the functions returns a non-`nil` value, that value is returned and the remaining functions are not called. If all of the functions return `nil`, then `nil` is returned. If the list of functions is empty, then `nil` is returned. That is, `(orf)` returns a function which accepts any arguments and returns `nil`.

9.43.14 Function `notf`

Syntax:

```
(notf function)
```

Description:

The `notf` function returns a function which is the Boolean negation of *`function`*.

The returned function takes a variable number of arguments. When invoked, it passes all of these arguments to *`function`* and then inverts the result as if by application of `not`.

9.43.15 Functions `nandf` and `norf`

Syntax:

```
(nandf func*)
(norf func*)
```

Description:

The `nandf` and `norf` functions are the logical negation of the `andf` and `orf` functions. They are related according to the following equivalences:

```
[nandf f0 f1 f2 ...] <--> (notf [andf f0 f1 f2 ...])
[norf f0 f1 f2 ...] <--> (notf [orf f0 f1 f2 ...])
```

9.43.16 Functions `iff` and `iffi`

Syntax:

```
(iff condfun [thenfun [elsefun]])
(iffi condfun thenfun [elsefun])
```

Description:

The `iff` function is the functional equivalent of the `if` operator. It accepts functional arguments and returns a function.

The resulting function takes its arguments, if any, and applies them to *condfun*. If *condfun* yields true, then the arguments are passed to *thenfun* and the resulting value is returned. Otherwise the arguments are passed to *elsefun* and the resulting value is returned.

If *thenfun* is omitted then `identity` is used as default. This omission is not permitted by `iffi`, only `iff`.

If *elsefun* needs to be called, but is omitted, then `nil` is returned.

The `iffi` function differs from `iff` only in the defaulting behavior with respect to the *elsefun* argument. If *elsefun* is omitted in a call to `iffi` then the default function is `identity`. This is useful in situations when one value is to be replaced with another one when the condition is true, otherwise preserved.

The following equivalences hold between `iffi` and `iff`:

```
(iffi a b c)            <--> (iff a b c)

(iffi a b)              <--> (iff a b identity)

[iffi a b nilf]         <--> [iff a b]

[iffi a identity nilf]  <--> [iff a]
```

The following equivalences illustrate `iff` with both optional arguments omitted:

```
[iff a]  <--->  [iff a identity nilf]  <--->  a
```

9.43.17 Functions `tf`, `nilf` and `ignore`

Syntax:

```
(tf arg*)
(nilf arg*)
(ignore arg*)
```

Description:

The `tf` and `nilf` functions take zero or more arguments, and ignore them. The `tf` function returns `t`, and the `nilf` function returns `nil`.

The `ignore` function is a synonym of `nilf`.

Note: the following equivalences hold between these functions and the `ret` operator, and `retf` function.

```
(fun tf) <--> (ret t) <--> (retf t)
(fun nilf) <--> (ret nil) <--> (ret) <--> (retf nil)
```

In Lisp-1-style code, `tf` and `nilf` behave like constants which can replace uses of `(ret t)` and `(ret nil)`:

```
[mapcar (ret nil) list] <--> [mapcar nilf list]
```

Note: the `ignore` function can be used for suppressing unused variable warnings.

Example:

```
;; tf and nilf are useful when functions are chained together.
;; test whether (trunc n 2) is odd.

(defun trunc-n-2-odd (n)
  [[chain (op trunc @1 2) [iff oddp tf nilf]] n])
```

In this example, two functions are chained together, and `n` is passed through the chain such that it is first divided by two via the function denoted by `(op trunc @1 2)` and then the result is passed into the function denoted by `[iff oddp tf nilf]`. The `iff` function passes its argument into `oddp`, and if `oddp` yields true, it passes the same argument to `tf`. Here `tf` proves its utility by ignoring that value and returning `t`. If the argument (the divided value) passed into `iff` is even, then iff passes it into the `nilf` function, which ignores the value and returns `nil`.

The following example shows how `ignore` may be used to suppress compiler warnings about unused parameters or other variables:

```
(defun (x y)
  (ignore x y))
```

9.43.18 Function `retf`

Syntax:

```
(retf value)
```

Description:

The `retf` function returns a function. That function can take zero or more arguments. When called, it ignores its arguments and returns *value*.

See also: the `ret` macro.

Example:

```
;; the function returned by (retf 42)
;; ignores 1 2 3 and returns 42.
(call (retf 42) 1 2 3) -> 42
```

9.43.19 Functions `apf` and `ipf`

Syntax:

```
(apf function arg*)
(ipf function arg*)
```

Description:

The `apf` function returns a one-argument function whose argument conventions are similar to those of the `apply` function: it accepts one or more arguments, the last of which should be a list. When that function is called, it applies *`function`* to these arguments to as if by `apply`. It then returns whatever *`function`* returns.

If one or more additional *`arg`*s are passed to `apf`, then these are stored in the function which is returned. When the function is invoked, it prepends all of these stored arguments to those that it is being given, and the resulting combined arguments are applied. Thus the *`arg`*s become the left-most arguments of *`function`*.

The `ipf` function is similar to `apf`, except that the argument conventions and application semantics of the function returned by `ipf` are based on `iapply` rather than `apply`.

See also: the `ap` macro.

Example:

```
;; Function returned by [apf +] accepts the
;; (1 2 3) list and applies it to +, as
;; if (+ 1 2 3) were called.

(call [apf +] '(1 2 3)) -> 6
```

9.43.20 Function `callf`

Syntax:

```
(callf main-function arg-function*)
```

Description:

The `callf` function returns a function which applies each *`arg-function`* to its arguments, juxtaposing the return values of these calls to form arguments to which *`main-function`* is then applied. The return value of *`main-function`* is returned.

The following equivalence holds, except for the order of evaluation of arguments:

```
(callf fm f0 f1 f2 ...) <--> (chain (juxt f0 f1 f2 ...)
                                    (apf fm))
```

Example:

```
;; Keep those pairs which are two of a kind

(keep-if [callf eql first second] '((1 1) (2 3) (4 4) (5 6)))
-> ((1 1) (4 4))
```

The following equivalence holds between `juxt` and `callf`:

```
[juxt f0 f1 f2 ...]  <-->  [callf list f0 f1 f2 ...]:w
```

Thus, `juxt` may be regarded as a specialization of `callf` in which the main function is implicitly `list`.

9.43.21 Function `mapf`

Syntax:

```
(mapf main-function arg-function*)
```

Description:

The `mapf` function returns a function which distributes its arguments into the *arg-function*s. That is to say, each successive argument of the returned function is associated with a successive *arg-function*.

Each *arg-function* is called, passed the corresponding argument. The return values of these functions are then passed as arguments to *main-function* and the resulting value is returned.

If the returned function is called with fewer arguments than there are *arg-function*s, then only that many functions are used. Conversely, if the function is called with more arguments than there are *arg-function*s, then those arguments are ignored.

The following equivalence holds:

```
(mapf fm f0 f1 ...) <--> (lambda (. rest)
                            [apply fm [mapcar call
                                              (list f0 f1 ...)
                                              rest]])
```

Example:

```
;; Add the squares of 2 and 3
[[mapf + [dup *] [dup *]] 2 3] -> 13
```

9.44 Input and Output (Streams)

TXR Lisp supports input and output streams of various kinds, with generic operations that work across the stream types.

In general, I/O errors are usually turned into exceptions. When the description of error reporting is omitted from the description of a function, it can be assumed that it throws an error.

9.44.1 Special variables `*stdout*`, `*stddebug*`, `*stdin*`, `*stderr*` and `*stdnull*`

Description:

These variables hold predefined stream objects. The `*stdin*`, `*stdout*` and `*stderr*` streams closely correspond to the underlying operating system streams. Various I/O functions require stream objects as arguments.

The `*stddebug*` stream goes to the same destination as `*stdout*`, but is a separate object which can be redirected independently, allowing debugging output to be separated from normal output.

The `*stdnull*` stream is a special kind of stream called a null stream. To read operations, the stream appears empty, like a stream open on an empty file. To write operations, it appears as a data sink of infinite capacity which consumes data and discards it. This stream is similar to the `/dev/null` device on Unix, and in fact has a relationship to it. If an attempt is made to obtain the underlying file descriptor of `*stdnull*` using the `fileno` function, then the `/dev/null`

device is open, if the host platform supports it. The resulting file descriptor number is returned, and also retained in the `*stdnull*` device. When `close-stream` is invoked on `*stdnull*`, that descriptor is closed. This feature of `*stdnull*` allows it to be useful for establishing redirections around the execution of external utilities.

Example:

```
;; redirect output of ls *.txt command to /dev/null
(let ((*stderr *stdnull*))
  (sh "ls *.txt"))
```

9.44.2 Special variables `*print-flo-format*` and `*pprint-flo-format*`

Description:

The `*print-flo-format*` variable determines the conversion format which is applied when a floating-point value is converted to decimal text by the functions `print`, `prinl`, and `tostring`.

The default value is `~s`.

The related variable `*pprint-flo-format*` similarly determines the conversion format applied to floating-point values by the functions `pprint`, `pprinl`, and `tostringp`.

The default value is `~a`.

The format string in either variable must specify the consumption of exactly one `format` argument.

The conversion string may use embedded width and precision values: for instance, `~3,4f` is a valid value for `*print-flo-format*` or `*pprint-flo-format*`.

9.44.3 Special variable `*print-flo-precision*`

Description:

The `*print-flo-precision*` special variable specifies the default floating-point printing precision which is used when the `~a` or `~s` conversion specifier of the `format` function is used for printing a floating-point value, and no precision is specified.

Note that since the default value of the variable `*print-flo-format*` is the string `~s`, the `*printf-flo-precision*` variable, by default, also determines the precision which applies when floating-point values are converted to decimal text by the functions `print`, `pprint`, `prinl`, `pprinl`, `tostring` and `tostringp`.

The default value of `*print-flo-precision*` is that of the `flo-dig` variable.

Note: to print floating-point values in such a way that their values can be precisely recovered from the printed representation, it is recommended to override `*print-flo-precision*` to the value of the `flo-max-dig` variable.

9.44.4 Special variable `*print-flo-digits*`

Description:

The `*print-flo-precision*` special variable specifies the default floating-point printing

precision which is used when the `~f` or `~e` conversion specifier of the `format` function is used for printing a floating-point value, and no precision is specified.

Its default value is `3`.

9.44.5 Special variable `*print-base*`

Description:

The `*print-base*` variable controls the base (radix) used for printing integer values. It applies when the functions `print`, `pprint`, `prinl`, `pprinl`, `tostring` and `tostringp` process an integer value. It also applies when the `~a` and `~s` conversion specifiers of the `format` function are used for printing an integer value.

The default value of the variable is `10`.

Meaningful values are: `2`, `8`, `10` and `16`.

When base 16 is selected, hexadecimal digits are printed as uppercase characters.

9.44.6 Special variable `*print-circle*`

Description:

The `*print-circle*` variable is a Boolean which controls whether the circle notation is in effect for printing aggregate objects: conses, ranges, vectors, hash tables and structs. The initial value of this variable is `nil`: circle notation printing is disabled.

The circle notation works for structs also, including structs which have user-defined `print` methods. When a `print` method calls functions which print objects, such as `print`, `pprinl` or `format` on the same stream, the detection of circularity and substructure sharing continues in these recursive invocations.

However, there are limitations in the degree of support for circle notation printing across `print` methods. Namely, a `print` method of a struct *S* must not procure and submit for printing objects which are not part of the ordinary structure that is reachable from the (static or instance) slots of *S*, if those objects have already been printed prior to invoking the `print` method, and have been printed without a `#=` circle notation label. The "ordinary structure that is reachable from the slots" denotes structure that is directly reachable by traversing conses, ranges, vectors, hashes and struct slots: all printable aggregate objects.

9.44.7 Special variable `*read-unknown-structs*`

Description:

The `*read-unknown-structs*` variable controls the behavior of the parser upon encountering structure literal `#S` syntax which specifies an unknown structure type.

If this variable's value is `nil` then such a literal is erroneous; an exception is thrown. Otherwise, such a structure is converted not into a structure object, which is impossible, but into a list object whose first element is the symbol `sys:struct-lit`. The remaining elements are taken from the `#S` syntax.

9.44.8 Function `format`

Syntax:

```
(format stream-designator format-string format-arg*)
```

Description:

The `format` function performs output to a stream given by *`stream-designator`*, by interpreting the actions implicit in a *`format-string`*, incorporating material pulled from additional arguments given by *`format-arg*`*. Though the function is simple to invoke, there is complexity in format string language, which is documented below.

The *`stream-designator`* argument can be a stream object, or one of the values `t` or `nil`. The value `t` serves as a shorthand for `*stdout*`. The value `nil` means that the function will send output into a newly instantiated string output stream, and then return the resulting string.

Format string syntax:

Within *`format-string`*, most characters represent themselves. Those characters are simply output. The character ~ (tilde) introduces formatting directives, which are denoted by a single character, usually a letter.

The special sequence ~~ (tilde-tilde) encodes a single tilde. Nothing is permitted between the two tildes.

The syntax of a directive is generally as follows:

```
~[width] [,precision] letter
```

In other words, the ~ (tilde) character, followed by a *`width`* specifier, a *`precision`* specifier introduced by a comma, and a *`letter`*, such that *`width`* and *`precision`* are independently optional: either or both may be omitted. No whitespace is allowed between these elements.

The *`letter`* is a single alphabetic character which determines the general action of the directive. The optional width and precision are specified as follows:

`width` The width specifier consists of an optional < (left angle bracket) character or ^ (caret) character followed by an optional width specification.

If the leading < character is present, then the printing will be left-adjusted within this field. If the ^ character is present, the printing will be centered within the field. Otherwise it will be right-adjusted by default.

The width can be specified as a decimal integer with an optional leading minus sign, or as the character `*`. The `*` notation means that instead of digits, the value of the next argument is consumed, and expected to be an integer which specifies the width. If the width, specified either way, is negative, then the field will be left-adjusted. If the value is positive, but either the < or ^ prefix character is present in the width specifier, then the field is adjusted according to that character.

The padding calculations for alignment and centering take into account character display width, as defined by the `display-width` function. For instance, a character string containing four Chinese characters (kanji) has a display width of 8, not 4.

The width specification does not restrict the printed portion of a datum. Rather, for some kinds of conversions, it is the precision specification that performs such truncation. A datum's display width (or that of its printed portion, after such truncation is applied) can

equal or exceed the specified field width. In this situation it overflows the field: the printed portion is rendered in its entirety without any padding applied on either side for alignment or centering.

precision

The precision specifier is introduced by a leading comma. If this comma appears immediately after the directive's ~ character, then it means that *width* is being omitted; there is only a precision field.

The precision specifier may begin with these optional characters, whose effect

0	the "leading zero option": pad with leading zeros;
+	print a sign for positive values;
−	print a single leading zero in place of a positive sign; and
space	print a space in place of a positive sign.

The precision value influences the printing of values of all types. The precision options apply only when the value being printed is a number; otherwise they are ignored.

If the +, − or space are multiply specified, the rightmost one takes precedence.

The precision specifier itself follows: it must be either a decimal integer or the * character indicating that the precision value comes from an integer argument.

The leading zero option is only active if accompanied by a precision value, either coming from additional digits in the formatting directive, or from an argument indicated by *. If no precision specifier is present, then the leading zero option is interpreted as a specifier indicating a precision value of zero, rather than requesting leading zeros. To request zero padding together with zero precision, either two or more zero digits are required, or else the leading zero indicator must be given together with the * specifier.

For non-numeric values, the precision specifies the maximum number of print positions to occupy, taking into account the display width of each character of the printed representation of the object, as according to the `display-width` function. The object's printed representation is truncated, if necessary, to the maximum number of characters which will not exceed the specified number of print positions.

A numeric argument is formatted into the field in two distinct steps, both of which involve the precision value in a different role. The details of the first of these steps, and the role payed by precision, depend on which conversion directive is used, as well as whether the argument is integer or floating-point. That first step prepares the printed representation of a number which is then fitted into the field by the second step, and also calculates the *effective precision* value, which is based on the original width and precision. The second step works with the effective precision rather than the original precision. Its description follows.

First, the length of the printed representation of the number, not including its sign, is calculated. If this part of the number is shorter than the effective precision, then it is padded on the left with spaces or leading zeros so that the resulting string is equal to the precision.

Next, if the number is negative, or else if adding a positive sign has been requested, then the sign is added. It is added to the left of the padding zeros, or else to the right of padding spaces, whichever the case may be.

At this stage, if the number is not yet adorned with a sign, and either the - or space precision option had been given, then the appropriate character, the digit `0` or a space, is added in the place where the sign would go. This is done only if the result will not overflow the field width, but without regard for whether the character will overflow the effective precision.

Finally, the resulting number is rendered into the field, using the requested left, right or center adjustment, as if it were a character string. If it overflows the field, it is reproduced in its entirety without any adjustment being performed.

Format directives:

Format directives are case sensitive, so that for example `~x` and `~X` have a different effect, and `~A` doesn't exist whereas `~a` does. They are:

`a` Prints any object in an aesthetic way, as if by the `pprint` function. The aesthetic notation violates read-print consistency: this notation is not necessarily readable if it is implanted in **TXR** source code. The field width specifier is honored, including the left-right adjustment semantics.

When the `a` specifier is used for numbers, the formatting is performed in two distinct steps: the printed representation of the number is calculated first, and then that representation is set into the field. At the same time, an effective precision is calculated, based on the precision and width, and that effective precision is used in the second step.

In the first step, the rendering of a floating-point number to its printed representation, the precision specifies the maximum number of total significant figures, which do not include any digits in the exponent, if one is printed. Numbers are printed in E notation if their magnitude is small, or else if their exponent exceeds their precision. If the precision is not specified, then it is obtained from the `*print-flo-precision*` special variable, whose default value is the same as that of the `flo-dig` variable. The effective precision for the second step is then taken from the original precision, or one less than the width, whichever of the these two values is smaller, but no lower than zero. If the width is unspecified, it is taken as zero.

Floating point values which are integers are printed without a trailing `.0` (point zero). The + flag in the precision is honored for rendering an explicit + sign on nonnegative values. If a leading zero is specified in the precision, and a nonzero width is specified, then the printed value's integer part will be padded with leading zeros up to one less than the field width. These zeros are placed before the sign. A precision value of zero imposed on floating-point values is equivalent to a value of one; it is not possible to request zero significant figures.

Integers are not affected by the precision value in the conversion to text; all of the digits of the integer are taken into the second step. In the case of integers, The effective precision for the second step is then taken from the original precision, or one less than the width, whichever of the these two values is smaller. However, if the width is not specified, or given as zero, then the unmodified precision value is taken as the effective precision. Thus, in the zero width or missing width case, integers are always padded with spaces or leading zeros due to the precision value, even if such padding overflows the field width.

Rationale: the purpose of the elaborate rules for calculating the effective precision is to both obtain consistency in the printing of integers and floating-point values that are integers, as well as to break that consistency when the width is omitted or zero. This break in

consistency has two benefits. The common situation of adding leading spaces or zeros to integers can be specified without specifying the width. For instance `"~,8a"` will format an integer right-justified in an eight-character extent, without width having to be used in order to specify a field to accommodate that padding. The effective padding amount going into the second step is 8, exceeding the zero width, and thus allowing the padding to overflow the field. In the case of floating-point, precision alone can express the common requirement for limiting the number of digits can be expressed by the precision, without causing unwanted padding when there are fewer digits. If the above `"~,8a"` is used to format a floating-point value, it will be limited to 8 digits of precision, regardless of its magnitude and the position of its decimal point, or whether or not exponential notation is used. The effective precision for field placement shall then be zero in the second step, so that no padding is generated. However, if a nonzero width is used, then formatting becomes consistent between floating-point and integer so that, for instance, the format directive `"~8,8a"` produces the same output for the argument values 42 and 42.0, namely an eight-character-wide field in which the digits `"42"` appear right-aligned.

`s` Prints any object in a standard way, as if by the `print` function. Objects for which read-print consistency is possible are printed in a way such that if their notation is implanted in **TXR** source, they are readable. The field width specifier is honored, including the left-right adjustment semantics. The precision field is treated similarly to the `~a` format directive, except that non-exponentiated floating point numbers that would be mistaken for integers include a trailing `.0` for the sake of read-print consistency. Objects truncated by precision may not have read-print consistency. For instance, if a string object is truncated, it loses its trailing closing quote, so that the resulting representation is no longer a properly formed string object. For integer objects, the `*print-base*` variable is honored. Effectively, an integer is printed by the `s` directive as if by the `b`, `o`, `d`, or `x` directive, depending on the value of the variable.

`d` Requires an argument of integer or character type type. The integer value or character code is printed in decimal. Width and precision semantics are as described for the `a` format directive, for integers.

`x` Requires an argument of character, integer or buffer type. The integer value, character code, or buffer contents are printed in hexadecimal, using lowercase letters for the digits `a` through `f`. Width and precision semantics are as described for the `a` format directive, for integers.

`X` Like the `x` directive, but the hexadecimal digits `a` through `f` are rendered in uppercase.

`o` Like the `x` directive, but octal is used instead of hexadecimal.

`b` Like the `x` directive, but binary is used instead of hexadecimal.

`f` The `f` directive prints numbers in a fixed point decimal notation, with a fixed number of digits after the decimal point. It requires a numeric argument. (Unlike `x`, `X` and `o`, it does not allow an argument of character type).

The formatting performed by `f` is performed in two distinct steps: the printed representation of the number is calculated first, and then that representation is set into the field. The precision parameter coming from the directive is only involved in the first step.

In the first step, the precision specifier gives the number of digits past the decimal point.

The number is rounded off to the specified precision, if necessary. Furthermore, that many digits are always printed, regardless of the actual precision of the number or its type. If it is omitted, then the value is obtained from the special variable `*print-flo-digits*`, whose default value is three: three digits past the decimal point. A precision of zero means no digits past the decimal point, and in this case the decimal point is suppressed (regardless of whether the numeric argument is floating-point or integer).

No limit is placed on the number of significant figures in the number by either the precision or width value.

When the resulting textual number passes to the second formatting step, the precision value, for the purposes of that step, is calculated by taking one less than the field width, or else zero if the field width is zero. This value is not related to the precision that had been used to determine the number of places past the decimal point.

`e` The `e` directive prints numbers in E notation. It requires a numeric argument. (Unlike `x`, `X` and `o`, it does not allow an argument of character type).

The formatting performed by `e` is performed in two distinct steps: the printed representation of the number is calculated first, and then that representation is set into the field. The precision parameter coming from the directive is only involved in the first step.

In the first step, the precision specifier gives the number of digits past the decimal point printed in the E notation, not counting the digits in the exponent. Exactly that many digits are printed, regardless of the precision of the number. If the precision is omitted, then the number of digits after the decimal point is obtained from the value of the special variable `*print-flo-digits*`, whose default value is three. If the precision is zero, then a decimal portion is truncated off entirely, including the decimal point.

When the resulting textual number passes to the second formatting step, the precision value, for the purposes of that step, is calculated by taking one less than the field width, or else zero if the field width is zero. This value is not related to the precision that had been used to determine the number of places past the decimal point.

`p` The `p` directive prints a numeric representation in hexadecimal of the bit pattern of the object, which is meaningful to someone familiar with the internals of **TXR**. If the object is a pointer to heaped data, that value has a correspondence to its address.

`!` The `!` directive establishes hanging indentation, and turns on the stream's indentation mode. Subsequent lines printed within the execution of the same `format` call will be automatically indented. If no width is specified, then the directive sets the hanging indentation to the current printing column position. If a width is specified, then it represents an offset (positive or negative). If the < prefix character is present, the hanging indentation is set to the specified offset relative to the current printing column. If the < prefix is present on the width field, then the offset is applied relative to the indentation which was saved on entry into the `format` function.

The indentation mode and indentation column are automatically restored to their previous values when `format` function terminates, naturally or via an exception or nonlocal jump.

The effect of a precision field (even if zero) combined with the `!` directive is currently not specified, and reserved for future extension. The precision field is processed

syntactically, and no error occurs, however.

9.44.9 Function `fmt`

Syntax:

```
(fmt format-string format-arg*)
```

Description:

The `fmt` function provides a shorthand for formatting to a string, according to the following equivalence which holds between `fmt` and `format`:

```
(fmt s arg ...)  <-->  (format nil s arg ...)
```

9.44.10 Macro `pic`

Syntax:

```
(pic format-string format-arg*)
```

Description:

The `pic` macro ("picture based formatting") provides a notation for constructing a character string under the control of *`format-string`* which indicates the insertion of zero or more *`format-arg`* argument values.

Like the `fmt` function or quasiliteral syntax, the `pic` macro returns a character string.

The `pic` macro's *`format-string`* notation is different from quasiliterals or from `fmt`.

The `pic` *`format-string`* argument isn't an evaluated expression, but syntax. It must be either a string literal or else a string quasiliteral. No other syntax is permitted.

If *`pic`* is a string, is scanned left to right in search of *pic patterns*. Any characters not belonging to a pic pattern are copied into the output string verbatim. When a pic pattern is found, it is removed from *`format-string`* and applied to the next successive *`format-arg`* to perform a conversion and formatting of that value to text. The resulting text is appended to the output string, and the process continues in search of the next pic pattern. When the *`format-string`* is exhausted, the constructed string is returned.

If *`format-string`* is a quasiliteral, then all of the text strings embedded within the quasiliteral are examined in the same way, in left to right order. Each such string is transformed into an expression which produces a character string according to the semantics of the pic patterns it contains, and the resulting expressions are substituted into the original quasiliteral to produce a transformed quasiliteral.

There must be exactly as many *`format-arg`* arguments as there are pic patterns in *`format-string`*.

The `pic` macro arranges for the left-to-right evaluation of the *`format-arg`* expressions. If *`format-string`* is a quasiliteral, the evaluation of these expressions is interleaved into the quasiliterals expressions and variables, in the order implied by the placement of the corresponding pic patterns relative to the quasiliteral elements. For instance, if *`format-string`* is `` `@(abc)<<<@(xyz)` `` then the function `abc` is called first, then the *`format-argument`* is evaluated which produces a value for the `<<<` pic pattern, after which the `xyz` function is called.

There are two kinds of pic patterns: alignment patterns, numeric patterns and escape patterns.

Escape patterns consist of a two-character sequence introduced by the ~ (tilde) character, which is followed by one of the characters that are special in pic pattern syntax:

```
< > | + - 0 # . ! ~ , ( )
```

An escape pattern produces the second character as its output. For instance ~~ encoded a single ~ character, and ~# encodes a literal # character that is not part of any pattern.

Alignment patterns are described next.

`<<...<<`
: A sequence of one or more < (less than) characters specifies that the corresponding argument is rendered left-aligned in a field whose width is given by the number of < characters. If the argument's textual representation doesn't fit into the field, it overflows.

`>>...>>`
: A sequence of one or more > (greater than) characters specifies that the corresponding argument is rendered right-aligned in a field whose width is given by the number of > characters. If the argument's textual representation doesn't fit into the field, it overflows.

`||...||`
: A sequence of one or more | (pipe) characters specifies that the corresponding argument is centered in a field whose width is given by the number of | characters. If the argument's textual representation doesn't fit into the field, it overflows. If the argument cannot be precisely centered, because the even-odd parity of its character count is different from the parity of the field width, it is centered slightly to the left: one less space appears on its left side in respect to its right side.

The numeric patterns, by means of their visual pattern and several optional prefix codes, specify the parameters for the conversion of a numeric argument, which is rendered right-aligned in a fixed-width field. Numeric patterns that do not contain any commas conform this simple rule:

```
[sign] [0] {#}+ [point {#}+ | !]
```

or else if they contain commas, the placement of these commas is governed by the more complicated rule:

```
[sign] [0 [,]] {#}+ {,{#}+}* [point {#}+ {,{#}+}* | !]
```

Commas may be placed anywhere within the pattern of hash characters, except at the beginning or end, or adjacent to the decimal point. If the leading zero is present, a comma may appear immediately after it, before the first hash.

A second form of both of the above patterns is supported, for specifying that negative numbers be shown in parentheses. Instead of the sign, an opening parenthesis may appear, which must be matched by a closing parenthesis which follows a valid pattern interior:

```
( [0] {#}+ [point {#}+ | !] )
```

With embedded commas:

```
( [0 [,]] {#}+ {,{#}+}* [point {#}+ {,{#}+}* | !] )
```

The pattern consists of an optional *sign* which is one of the characters + (plus) or − (minus), or else it may optionally begin with an opening parenthesis, indicating one of the two alternative forms.

This is followed by an optional leading zero. After this comes a sequence of one or more # (hash) characters, which may contain exactly one *point* element, which is defined as one of the characters . (period) or ! (exclamation mark). This *point* element may appear at most once, and must not be the first or last character, unless it is the exclamation mark, in which case it may appear last.

Except if ending in the exclamation mark, a numeric pattern specifies a field width which is equal to the number of characters occurring in the pattern itself. For instance, the patterns ####, +### and 0#.# all specify a field width of four. If the numeric pattern ends in an exclamation mark, that character is not counted toward the field width that it specifies. Thus the pattern ###! specifies a field width of three.

If the leading sign is present, it has the following meanings:

+ If the corresponding numeric argument is nonnegative, the + character shall appear before first digit. Otherwise the minus character will appear.

− Like + except that when the numeric argument is nonnegative, instead of a + character, a space appears before the first digit. This space counts toward the field width and therefore contributes to overflow.

If a leading sign is not present, then no extra character appears before the first digit of a positive value, which means that an extra character of field width is available for representing nonnegative values.

If the leading zero is present, it specifies that the number is padded with zeros on the left. In combination with the − sign, this shall not cause the leading space before a positive value to be overwritten with a zero; leading zeros, if any, begin after that space.

The remainder of the pattern specifies the number of digits of the fractional part which is indicated by number of # characters after the *point*. The number is rounded to that many fractional digits, which are all rendered, even if there are trailing zeros. If no *point* is not specified, then the number of fractional digits is zero. The same is true if *point* is specified as ! as the last character. In both cases, the numeric argument is rounded to integer, and rendered without any decimal point or fractional part.

There is a difference between *point* being specified using the ordinary decimal point character . versus the ! character. The ! character specifies that if the conversion of the numeric argument overflows the field, then instead of showing any digits, the field is filled with a pattern consisting of # (hash) characters, and possibly an embedded decimal point. In contrast, the . character permits the field's width to increase to accommodate overflowing output. If overflow takes place and the ! character appears other than as the rightmost character of the pattern, then the decimal point character . character appears at the position indicated by that ! character. If the ! character is the rightmost character of the pattern, then, just as in the case of normal, non-overflowing output, it doesn't contribute to the width of the hash fill, and only hash characters appear.

If commas appear in the numeric pattern according to the more complex syntactic rule, they count toward the field width and specify the insertion of digit-separating commas at the indicated locations. Digit separators may be specified on either side of the decimal point, but not adjacent to it. In the output, a digit separating comma shall not appear if it would be immediately preceded by a + or − sign or space. In this situations, the sign character or space appears in place of the digit separator. A digit separator that appears in a position occupied by a space is also suppressed in favor of the space. Digit separators are included among leading zeros. It is not logically possible for a digit separator to appear as the first character of a pattern's output, because it may not be the first character of a pattern. However, if a numeric pattern is preceded or followed by a comma, those commas are ordinary characters which are copied to the output.

When, due to the presence of `!`, an overflowing field is handled by the generation of a the hash character fill, the hash characters are treated as digits for the purpose of digit separation.

When the pattern uses parentheses to specify that negative numbers are to be shown with parentheses, the parentheses count toward the field width. The field portion between the parentheses is called the inner field. The parentheses appear in the output when the number is negative, and are placed immediately outside of the inner field, so that if leading zeros are not requested, there may be one or more spaces between the opening parenthesis and the first digit. If the number is nonnegative, then each parenthesis is replaced by one space, flanking the inner field in the same manner as parentheses.

Examples:

```
;; numeric formatting
(pic  "######"     1234.1) ->   "  1234"
(pic  "######.#"   1234.1) ->   "  1234.1"
(pic "#######.##"  1234.1) ->  "   1234.10"
(pic "#######.##" -1234.1) ->  "  -1234.10"
(pic "0######.##"  1234.1) ->  "0001234.10"
(pic "+######.##"  1234.1) ->  "  +1234.10"
(pic "-######.##"  1234.1) ->  "   1234.10"
(pic "+0#####.##"  1234.1) ->  "+001234.10"
(pic "-0#####.##"  1234.1) ->  " 001234.10"
(pic "#######.##" -1234.1) ->  "  -1234.10"

;; digit separation
(pic "0,###,###.##" 1234.1) ->  "0,000,123.10"
(pic "#,###,###.##" 1234.1) ->  "      123.10"

;; overflow with !
(pic   "#!#" 1234) ->    "###"
(pic   "#!#"  123) ->    "###"
(pic "-##!#" -123) ->  "#####"
(pic "+##!#"  123) ->  "#####"
(pic "###!"  1234) ->    "###"

;; negative parentheses
(pic "(#,###.##) 1234.56) -> " 1,234.56 "
(pic "(#,###.##) -234.56) -> "(  234.56)"

;; alignment, multiple arguments
(pic "<<<<<< 0#.# >>>>>>>" "foo" (+ 2 2) "bar")
-->  "foo    04.0     bar"

;; quasiliteral
(let ((a 2) (b "###") (c 13.5))
  (pic `abc@(+ a a)###.##@b>>>>` c "x"))
--> "abc4 13.50###   x"

;; filename generation
(mapcar (do pic "foo~-0##.jpg") (rlist 0..5 8 12))

--> ("foo-000.jpg" "foo-001.jpg" "foo-002.jpg" "foo-003.jpg"
     "foo-004.jpg" "foo-005.jpg" "foo-008.jpg" "foo-012.jpg")
```

9.44.11 Functions `print`, `pprint`, `prinl`, `pprinl`, `tostring` and `tostringp`

Syntax:

```
(print obj [stream [pretty-p]])
(pprint obj [stream])
(prinl obj [stream])
(pprinl obj [stream])
(tostring obj)
(tostringp obj)
```

Description:

The `print` and `pprint` functions render a printed character representation of the *`obj`* argument into *`stream`*.

If the *`stream`* argument is not supplied, then the destination is the stream currently stored in the `*stdout*` variable.

If Boolean argument *`pretty-p`* is not supplied or is explicitly specified as `nil`, then the `print` function renders in a way which strives for read-print consistency: an object is printed in a notation which is recognized as a similar object of the same kind when it appears in **TXR** source code. Floating-point objects are printed as if using the `format` function, with formatting controlled by the `*print-flo-format*` variable.

If *`pretty-p`* is true, then `print` does not strive for read-print consistency. Strings are printed by sending their characters to the output stream, as if by the `put-string` function, rather than being rendered in the string literal notation consisting of double quotes, and escape sequences for control characters. Likewise, character objects are printed via `put-char` rather than the `#\` notation. Buffer objects are printed by sending their bytes to the output stream using `put-byte` rather than being rendered in the `#b` notation. Symbols are printed without their package prefix, except that symbols from the keyword package are still printed with the leading colon. Floating-point objects are printed as if using the `format` function, with formatting controlled by the `*pprint-flo-format*` variable.

When aggregate objects like conses, ranges and vectors are printed, the notations of these objects themselves are unaffected by the `pretty-p` flag; however, that flag is distributed to the elements.

The `print` function returns *`obj`*.

The `pprint` ("pretty print") function is equivalent to `print`, with the *`pretty-p`* argument hardcoded true.

The `prinl` function ("print and new line") behaves like a call to `print` with *`pretty-p`* defaulting to `nil`, followed by issuing a newline characters to the stream.

The `pprinl` function ("pretty print and new line") behaves like `pprint` followed by issuing a newline to the stream.

The `tostring` and `tostringp` functions are like `print` and `pprint`, but they do not accept a stream argument. Instead they print to a freshly instantiated string stream, and return the resulting string.

The following equivalences hold between calls to the `format` function and calls to the above functions:

```
(format stream "~s" obj)  <-->  (print obj stream)
```

```
(format t "~s" obj)       <-->  (print obj)
(format t "~s\n" obj)     <-->  (prinl obj)
(format nil "~s" obj)     <-->  (tostring obj)
```

For `pprint`, `tostringp` and `pprinl`, the equivalence is produced by using `~a` in format rather than `~s`.

Notes:

For floating-point numbers, the above description of the behavior in terms of the format specifiers `~s` and `~a` only applies with respect to the default values of the variables `*print-flo-format*` and `*pprint-flo-format*`.

For characters, the print function behaves as follows: most control characters in the Unicode `C0` and `C1` range are rendered using the `#\x` notation, using two hex digits. Codes in the range `D800` to `DFFF`, and the codes `FFFE` and `FFFF` are printed in the `#\xNNNN` with four hexadecimal digits, and character above this range are printed using the same notation, but with six hexadecimal digits. Certain characters in the `C0` range are printed using their names such as `#\nul` and `#\return`, which are documented in the Character Literals section. The `DC00` character is printed as `#\pnul`. All other characters are printed as `#\`*`char`* where *`char`* is the actual character.

Caution: read-print consistency is affected by trailing material. If additional digits are printed immediately after a number without intervening whitespace, they extend that number. If hex digits are printed after the character `x`, which is rendered as `#\x`, they look like a hex character code.

9.44.12 Function `tprint`

Syntax:

```
(tprint obj [stream])
```

Description:

The `tprint` function prints a representation of *`obj`* on *`stream`*.

If the stream argument is not supplied, then the destination is the stream currently stored in the `*stdout*` variable.

For all object types except lists and vectors, `tprint` behaves like `pprinl`.

If `obj` is a list or vector, then `tprint` recurses: the `tprint` function is applied to each element. An empty list or vector results in no output at all. This effectively means that an arbitrarily nested structure of lists and vectors is printed flattened, with one element on each line.

9.44.13 Function `display-width`

Syntax:

```
(display-width char)
(display-width string)
```

Description:

The `display-width` function calculates the number of places occupied by the printed representation of *`char`* or *`string`* on a monospace display which renders certain characters, such as the East Asian kanji and other characters, using two places.

For a *string* argument, this value is the sum of the individual display width of the string's constituent characters. The display width of an empty string is zero.

Control characters are assigned a display width of zero, regardless of their display control semantics, if any.

Characters marked by Unicode as being wide or full width, have a display width of two. Other characters have a display width of one.

9.44.14 Function `streamp`

Syntax:

```
(streamp obj)
```

Description:

The `streamp` function returns `t` if *obj* is any type of stream. Otherwise it returns `nil`.

9.44.15 Function `real-time-stream-p`

Syntax:

```
(real-time-stream-p obj)
```

Description:

The `real-time-streamp-p` function returns `t` if *obj* is a stream marked as "real-time". If *obj* is not a stream, or not a stream marked as "real-time", then it returns `nil`.

Only certain kinds of streams accept the real-time attribute: file streams and tail streams. This attribute controls the semantics of the application of `lazy-stream-cons` to the stream. For a real-time stream, `lazy-stream-cons` returns a stream with "naive" semantics which returns data as soon as it is available, at the cost of generating spurious `nil` item when the stream terminates. The application has to recognize and discard that `nil` item. The ordinary lazy streams read ahead by one line and suppress this extra item, so their representation is more accurate.

When **TXR** starts up, it automatically marks the `*stdin*` stream as real-time, if it is connected to a TTY device (a device for which the POSIX function `isatty` reports true). This is only supported on platforms that have this function. The behavior is overridden by the `-n` command-line option.

9.44.16 Function `open-file`

Syntax:

```
(open-file path [mode-string])
```

Description:

The `open-file` function creates a stream connected to the file which is located at the given *path*, which is a string.

The *mode-string* argument is a string which uses the same conventions as the mode argument of the C language `fopen` function, with greater permissiveness, and some extensions.

The syntax of *mode-string* is described by the following grammar. Note that it permits no whitespace characters:

```
mode-string := [ mode ] [ options ]
```

```
mode := { selector [ + ] | + }
selector := { r | w | a | m }
options := { b | x | l | u | i | n | digit |
             z[digit] | redirection }
digit := { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }
```

If the *mode-string* argument is omitted, the behavior is the same as an empty mode string.

The *mode* part of the mode string generates the following possibilities:

`empty` If the *mode* is missing, then a default mode is implied. The default is specific to the particular stream-opening function. In the case of `open-file`, the default mode is `r`.

`+` A *mode* consisting of just the + character is equivalent to `r+`.

`r` This *mode* means that the file is opened for reading.

`r+` The file is opened for reading and writing. It is not created if it doesn't exist.

`w` The file is opened for writing. If it exists, it is truncated to zero length. If it doesn't exist, it is created.

`w+` The file is opened for reading and writing. If it exists, it is truncated to zero length. If it doesn't exist, it is created.

`m` The file is opened for modification. This is the same as `w` except that the file is not truncated if it exists.

`m+` The file is opened for reading and modification. This is the same as `w+` except that the file is not truncated if it exists.

`a` The file is opened for writing. If it doesn't exist, it is created. If it exists, the current position is advanced to one byte past the end of the file, so that newly written data are appended.

`a+` The file is opened for reading and writing. If it doesn't exist, it is created. The read position is at the beginning of the file, but writes are appended to the end regardless of the position.

The meanings of the option characters are:

`b` The file is opened in binary mode: no line-ending translation takes place. In the absence of this option, files are opened in text mode, in which newline characters in the stream are an abstract indication of the end of a line, translate to a system-specific way of terminating lines in text files.

`x` The file is created and opened only if it does not already exist. Otherwise, a `file-error` exception is thrown. This option is allowed only with the `w` and `w+` modes.

`l` Specifies that the stream will be line buffered. This means that an implicit flush operation takes place whenever the newline character is output.

`u` Specifies that the stream will be unbuffered. It is erroneous for both `l` and `u` to be specified.

`i` Specifies that the stream will have the real-time property set. For a description of the semantics, see the `real-time-stream-p` function. Briefly, this property affects the semantics of lazy lists which draw input from the stream. In addition, for a stream opened for writing or reading and writing, the `i` mode letter specifies that the stream will be line buffered, unless specified as unbuffered with `u`.

`n` Specifies that the operation shall not block.

`digit` A decimal digit specifies the stream buffer size as binary exponential buffer size order, such that `0` specifies 1024 bytes, `1` specifies 2048 and so forth up to `9` specifying 524288

bytes. If no such digit is specified, then the stream uses a default buffer size. It is erroneous for the size order digit to be present together with the option `u`.

`z` This option specifies `gzip` compression. A `gzip-stream` is opened for the file rather than an ordinary `stdio-stream`, which performs compression when writing and decompression when reading. The compression uses the Deflate algorithm, and a file format compatible with the `gzip` utility. If `z` is immediately followed by a digit, then that specifies the compression level. The default level is 6. A value of zero is an invalid level, silently accepted and treated as the default. When `z` is specified, special restrictions apply to the *`mode-string`*. If these are violated, an exception is thrown. A `gzip-stream` does not support an update mode; it may not be open simultaneously for reading and writing as, for example, requested by the `"r+"` mode. The options `l`, `u` and `i` are inapplicable. The digit option specifying the buffer size order also may not be used with `z`. The `seek-stream` function may only be used in the forward direction on `gzip` streams open for writing, and has the semantics of writing a region of zero bytes. Arbitrary seeking is supported in read mode, but via a costly emulation which decompresses data from the beginning of the file to the desired seek point.

`redirection`
This option refers to a special syntax that only has an effect in mode strings that are passed to the `open-process` function; the syntax performs I/O redirections in the child process created by that function, and is described in that function's documentation.

9.44.17 Function `open-tail`

Syntax:

```
(open-tail path [mode-string [seek-to-end-p]])
```

Description:

The `open-tail` function creates a tail stream connected to the file which is located at the given *`path`*. The *`mode-string`* argument is a string which uses the same conventions as the mode argument of the C language `fopen` function. If this argument is omitted, then `"r"` is used. See the `open-file` function for a discussion of modes.

The `seek-to-end-p` argument is a Boolean which determines whether the initial read/write position is at the start of the file, or just past the end. It defaults to `nil`. This argument only makes a difference if the file exists at the time `open-tail` is called. If the file does not exist, and is later created, then the tail stream will follow that file from the beginning. In other words, *`seek-to-end-p`* controls whether the tail stream reads all the existing data in the file, if any, or whether it reads only newly added data from approximately the time the stream is created.

A tail stream has special semantics with regard to reading at the end of file. A tail stream never reports an end-of-file condition; instead it polls the file until more data is added. Furthermore, if the file is truncated, or replaced with a smaller file, the tail stream follows this change: it automatically opens the smaller file and starts reading from the beginning (the *`seek-to-end-p`* flag only applies to the initial open). In this manner, a tail stream can dynamically grow rotating log files.

Caveat: since a tail stream can reopen a new file which has the same name as the original file, it behave incorrectly if the program changes the current working directory, and the pathname is relative.

9.44.18 Function `open-directory`

Syntax:

```
(open-directory path)
```

Description:

The `open-directory` function tries to create a stream which reads the directory given by the string argument *`path`*. If a filesystem object exists under the path, is accessible, and is a directory, then the function returns a stream. Otherwise, a file error exception is thrown.

The resulting stream supports the `get-line` operation. Each call to the `get-line` operation retrieves a string representing the next directory entry. The value `nil` is returned when there are no more directory entries. The `.` and `..` entries in Unix filesystems are not skipped.

9.44.19 Function `tmpfile`

Syntax:

```
(tmpfile)
```

Description:

The `tmpfile` function creates a new temporary binary file which is different from any existing file. It opens a stream for that file and returns the stream. The stream is created with the `open-file` mode `"w+b"`. When the stream is closed, or the **TXR** image terminates, the file is deleted.

Note: the `tmpfile` function is implemented using the same-named ISO C and POSIX library function. On POSIX systems of sufficient quality, `tmpfile` deletes the file before returning the open stream, such that the file object continues to exist while the stream is open, but is not known by any name in the file system. POSIX (IEEE Std 1003.1-2017) notes that in some implementations, "a permanent file may be left behind if the process calling tmpfile() is killed while it is processing a call to tmpfile".

Notes: if a unique file is required which exists in the file system under a known name until explicitly deleted, the `mkstemp` function may be used. If a unique directory needs to be created, the `mkdtemp` function may be used. These two functions are described in the Unix Filesystem Complex Operations section of the manual.

9.44.20 Function `make-string-input-stream`

Syntax:

```
(make-string-input-stream string)
```

Description:

The `make-string-input-stream` function produces an input stream object. Character read operations on the stream object read successive characters from *`string`*. Output operations and byte operations are not supported.

9.44.21 Function `make-string-byte-input-stream`

Syntax:

```
(make-string-byte-input-stream string)
```

Description:

The `make-string-byte-input-stream` function produces an input stream object. Byte read operations on this stream object read successive byte values obtained by encoding *`string`* into UTF-8. Character read operations are not supported, and neither are output operations.

9.44.22 Function `make-strlist-input-stream`

Syntax:

```
(make-strlist-input-stream list)
```

Description:

The `make-strlist-input-stream` function produces an input stream object based on a list of strings. Through the character read operations invoked on this stream, the list of strings appears as a list of newline-terminated lines. Output operations and byte operations are not supported.

9.44.23 Function `make-string-output-stream`

Syntax:

```
(make-string-output-stream)
```

Description:

The `make-string-output-stream` function, which takes no arguments, creates a string output stream. Data sent to this stream is accumulated into a string object. String output streams support both character and byte output operations. Bytes are assumed to represent a UTF-8 encoding, and are decoded in order to form characters which are stored into the string.

If an incomplete UTF-8 code is output, and a character output operation then takes place, that code is assumed to be terminated and is decoded as invalid bytes. The UTF-8 decoding machine is reset and ready for the start of a new code.

The `get-string-from-stream` function is used to retrieve the accumulated string.

If the null character is written to a string output stream, the behavior is unspecified. **TXR** strings cannot contain null bytes. The pseudo-null character `#\xDC00`, also notated `#\pnul`, will produce a null byte when converted to UTF-8 and thus serves as an effective internal representation of the null character in external data.

9.44.24 Function `get-string-from-stream`

Syntax:

```
(get-string-from-stream stream)
```

Description:

The *`stream`* argument must be a string output stream. This function finalizes the data sent to the stream and retrieves the accumulated character string.

If a partial UTF-8 code has been written to *`stream`*, and then this function is called, the byte stream is considered complete and the partial code is decoded as invalid bytes.

After this function is called, further output on the stream is not possible.

9.44.25 Function `make-strlist-output-stream`

Syntax:

```
(make-strlist-output-stream)
```

Description:

The `make-strlist-output-stream` function is similar to `make-string-output-stream`. However, the stream object produced by this function does not produce a string, but a

list of strings. The data is broken into multiple strings by newline characters written to the stream. Newline characters do not appear in the string list. Also, byte output operations are not supported.

9.44.26 Function `get-list-from-stream`

Syntax:

```
(get-list-from-stream stream)
```

Description:

The `get-list-from-stream` function returns the string list which has accumulated inside a string output stream given by *stream*. The string output stream is finalized, so that further output is no longer possible.

9.44.27 Macro `with-in-string-stream`

Syntax:

```
(with-in-string-stream (stream-var string)
   body-form*)
```

Description:

The `with-in-string-stream` macro binds the symbol *stream-var* as a variable, initializing it with a newly created string input stream. The string input stream is constructed from *string* as if by the `(make-string-input-stream string)` expression.

Then it evaluates the *body-form*s in the scope of the variable.

The value of the last *body-form* is returned, or else `nil` if no forms are present.

The *stream-var* argument must be a bindable symbol, as defined by the `bindable` function.

The *string* argument must be a form which evaluates to a character string value.

9.44.28 Macro `with-in-string-byte-stream`

Syntax:

```
(with-in-string-byte-stream (stream-var string)
   body-form*)
```

Description:

The `with-in-string-byte-stream` macro binds the symbol *stream-var* as a variable, initializing it with a newly created string byte input stream. The string input stream is constructed from *string* as if by the `(make-string-byte-input-stream string)` expression.

Then it evaluates the *body-form*s in the scope of the variable.

The value of the last *body-form* is returned, or else `nil` if no forms are present.

The *string* argument must be a form which evaluates to a character string value.

9.44.29 Macro `with-out-string-stream`

Syntax:

```
(with-out-string-stream (stream-var) body-form*)
```

Description:

The `with-out-string-stream` macro binds the symbol specified by the *stream-var* argument as a variable, initializing it with a newly created string output stream. The output stream is created as if by the `make-string-output-stream` function.

Then it evaluates *body-form*s in the scope of that variable.

After these forms are evaluated, the string is extracted from the string output stream, as if by the `get-string-from-stream` function, and returned as the result value of the form.

9.44.30 Macro `with-out-strlist-stream`

Syntax:

```
(with-out-strlist-stream (stream-var) body-form*)
```

Description:

The `with-out-strlist-stream` macro binds the symbol specified by the *stream-var* argument as a variable, initializing it with a newly created string list output stream. The output stream is created as if by the `make-strlist-output-stream` function.

Then it evaluates *body-form*s in the scope of that variable.

After these forms are evaluated, the string list is extracted from the string output stream, as if by the `get-strlist-from-stream` function, and returned as the result value of the form.

9.44.31 Function `make-byte-input-stream`

Syntax:

```
(make-byte-input-stream obj)
```

Description:

The `make-byte-input-stream` creates a stream which supports the `get-byte` operation for traversing a byte-wise representation of *obj*.

The function serves as a generic interface for calling one of several other stream constructing functions based on the type of the *obj* argument.

The *obj* argument must be either a buffer, in which case `make-byte-input-stream` behaves like `make-buf-stream`, or else a string, in which case the function behaves like `make-string-byte-input-stream`.

Note: the repertoire of types handled by `make-byte-input-stream` may expand in future language versions.

9.44.32 Function `close-stream`

Syntax:

```
(close-stream stream [throw-on-error-p])
```

Description:

The `close-stream` function performs a close operation on *stream*, whose meaning is depends on the type of the stream. For some types of streams, such as string streams, it does nothing. For streams which are connected to operating system files or devices, will perform a close of

the underlying file descriptor, and dissociate that descriptor from the stream. Any buffered data is flushed first.

`close-stream` returns a Boolean true value if the close has occurred without errors, otherwise `nil`.

For most streams, "without errors" means that any buffered output data is flushed successfully.

For command and process pipes (see `open-command` and `open-process`), success also means that the process terminates normally, with a successful error code, or an unsuccessful one. An abnormal termination is considered an error, as is the inability to retrieve the termination status, as well as the situation that the process continues running in spite of the close attempt. Detecting these situations is platform specific.

If the *`throw-on-error-p`* argument is specified, and isn't `nil`, then the function throws an exception if an error occurs during the close operation instead of returning `nil`.

If `close-stream` is called in such a way that it returns a value, without throwing an exception, that value is retained. Additional calls to the function with the same *`stream`* object return that same value without having any effect on the stream. These additional calls ignore the *`throw-on-error-p`* argument.

The *`stream`* may be associated with a process, in one of several ways: implicitly, by the functions `open-process` and `open-command` and related functions, or explicitly by the `open-fileno` function, if a *`pid`* argument is specified. In this situation, `close-stream` waits for the termination of that process, after closing the underlying file descriptor. If the process terminates normally, then `close-stream` returns its termination status, which is zero if the termination is successful. If the status of the process cannot be obtained, or is an abnormal termination, then the return value is `nil`. In that situation, if *`throw-on-error-p`* is true, an exception is thrown instead.

9.44.33 Macro `with-stream`

Syntax:

```
(with-stream (stream-var init-form)
   body-form*)
```

Description:

The `with-stream` macro binds the variable whose name is given by the *`stream-var`* argument, and macro arranges for the evaluation of *`body-form`*s in the scope of that variable.

The variable is initialized with the value produced by the evaluation of *`init-form`* which must be an expression which evaluates to a stream.

After each *`body-form`* is evaluated, the stream is closed, as if by the `(close-stream` *`stream-var`*`)` expression.

The value of the last *`body-form`* then becomes the result value of the form, or else `nil` if these forms are absent.

If the evaluation of the *`body-form`*s is abandoned, the stream is still closed. That is to say, the closure of the stream is a protected action, as if by the `unwind-protect` operator.

9.44.34 Functions `get-error`, `get-error-str` and `clear-error`

Syntax:

```
(get-error stream)
(get-error-str stream)
(clear-error stream)
```

Description:

When a stream operation fails, the `get-error` and `get-error-str` functions may be used to inquire about a more detailed cause of the error.

Not all streams support these functions to the same extent. For instance, string input streams have no persistent state. The only error which occurs is the condition when the string has no more data.

The `get-error` inquires *`stream`* about its error condition.

The function returns `nil` to indicate there is no error condition, `t` to indicate an end-of-data condition, or else a value which is specific to the stream type indicating the specific error type.

Note: for some streams, it is possible for the `t` value to be returned even though no operation has failed; that is to say, the streams "know" they are at the end of the data even though no read operation has failed. Code which depends on this will not work with streams which do not thus indicate the end-of-data *a priori*, but by means of a read operation which fails.

The `get-error-str` function returns a text representation of the error code. The `nil` error code is represented as the string `no error`; the `t` error code as `eof` and other codes have a stream-specific representation.

The `clear-error` function removes the error situation from a stream. On some streams, it does nothing. If an error has occurred on a stream, this function should be called prior to retrying any I/O or positioning operations. The return value is the previous error code, or `nil` if there was no error, or the operation is not supported on the stream.

9.44.35 Functions `get-line`, `get-char` and `get-byte`

Syntax:

```
(get-line [stream])
(get-char [stream])
(get-byte [stream])
```

Description:

These fundamental stream functions perform input. The *`stream`* argument is optional. If it is specified, it should be an input stream which supports the given operation. If it is not specified, then the `*stdin*` stream is used.

The `get-char` function pulls a character from a stream which supports character input. Streams which support character input also support the `get-line` function which extracts a line of text delimited by the end of the stream or a newline character and returns it as a string. (The newline character does not appear in the string which is returned).

Character input from streams based on bytes requires UTF-8 decoding, so that `get-char` may actually read several bytes from the underlying low-level operating system stream.

The `get-byte` function bypasses UTF-8 decoding and reads raw bytes from any stream which

supports byte input. Bytes are represented as integer values in the range 0 to 255.

Note that if a stream supports both byte input and character input, then mixing the two operations will interfere with the UTF-8 decoding.

These functions return `nil` when the end of data is reached. Errors are represented as exceptions.

See also: `get-lines`

9.44.36 Function `get-string`

Syntax:

```
(get-string [stream [count [close-after-p]]])
```

Description:

The `get-string` function reads characters from a stream, and assembles them into a string, which is returned. If the *`stream`* argument is omitted, then the `*stdin*` stream is used.

The stream is closed after extracting the data, unless *`close-after-p`* is specified as `nil`. The default value of this argument is `t`.

If the *`count`* argument is missing, then all of the characters from the stream are read and assembled into a string.

If present, the *`count`* argument should be a positive integer indicating a limit on how many characters to read. The returned string will be no longer than *`count`*, but may be shorter.

9.44.37 Functions `unget-char` and `unget-byte`

Syntax:

```
(unget-char char [stream])
(unget-byte byte [stream])
```

Description:

These functions put back, into a stream, a character or byte which was previously read. The character or byte must match the one which was most recently read. If the *`stream`* argument is omitted, then the `*stdin*` stream is used.

If the operation succeeds, the byte or character value is returned. A `nil` return indicates that the operation is unsupported.

Some streams do not support these operations; some support only one of them. In general, if a stream supports `get-char`, it supports `unget-char`, and likewise for `get-byte` and `unget-byte`.

Streams may require a pushed back byte or character to match the character which was previously read from that stream position, and may not allow a byte or character to be pushed back beyond the beginning of the stream.

Space may be available for only one byte of pushback under the `unget-byte` operation.

The number of characters that may be pushed back by `unget-char` is not limited.

Pushing both a byte and a character, in either order, is also unsupported. Pushing a byte and then

reading a character, or pushing a character and reading a byte, are unsupported mixtures of operations.

If the stream is binary, then pushing back a byte decrements its position, except if the position is already zero. At that point, the position becomes indeterminate.

The behavior of pushing back immediately after a `seek-stream` positioning operation is unspecified.

9.44.38 Functions `put-string`, `put-line`, `put-char` and `put-byte`

Syntax:

```
(put-string string [stream])
(put-line [string [stream]])
(put-char char [stream])
(put-byte byte [stream])
```

Description:

These functions perform output on an output stream. The *`stream`* argument must be an output stream which supports the given operation. If it is omitted, then `*stdout*` is used.

The `put-char` function writes a character given by `char` to a stream. If the stream is based on bytes, then the character is encoded into UTF-8 and multiple bytes are written. Streams which support `put-char` also support `put-line` and `put-string`.

The `put-string` function writes the characters of a string out to the stream as if by multiple calls to `put-char`. The *`string`* argument may be a symbol, in which case its name is used as the string.

The `put-line` function is like `put-string`, but also writes an additional newline character. The string is optional in `put-line`, and defaults to the empty string.

The `put-byte` function writes a raw byte given by the *`byte`* argument to *`stream`*, if *`stream`* supports a byte write operation. The byte value is specified as an integer value in the range 0 to 255.

All these functions return `t`. On failure, they do not return, but throw exceptions of type `file-error`.

9.44.39 Functions `put-strings` and `put-lines`

Syntax:

```
(put-strings sequence [stream]])
(put-lines sequence [stream]])
```

Description:

These functions assume *`sequence`* to be a sequence of strings, or of symbols, or a mixture thereof. These strings are sent to the stream. The *`stream`* argument must be an output stream. If it is omitted, then `*stdout*` is used.

The `put-strings` function iterates over *`sequence`* and writes each element to the stream as if using the `put-string` function.

The `put-lines` function iterates over `sequence` and writes each element to the stream as if

using the `put-line` function.

Both functions return `t`.

9.44.40 Function `flush-stream`

Syntax:

```
(flush-stream [stream])
```

Description:

The `flush-stream` function is meaningful for output streams which accumulate data which is passed on to the operating system in larger transfer units. Calling `flush-stream` causes all accumulated data inside *`stream`* to be passed to the operating system. If called on streams for which this function is not meaningful, it does nothing, and returns `nil`.

If *`stream`* is omitted, the current value of `*stdout*` is used.

9.44.41 Function `seek-stream`

Syntax:

```
(seek-stream stream offset whence)
```

Description:

The `seek-stream` function is meaningful for file streams. It changes the current read/write position within *`stream`*. It can also be used to determine the current position: see the notes about the return value below.

The *`offset`* argument is a positive or negative integer which gives a displacement that is measured from the point identified by the *`whence`* argument.

Note that for text files, there isn't necessarily a 1:1 correspondence between characters and positions due to line-ending conversions and conversions to and from UTF-8.

The *`whence`* argument is one of three keywords: `:from-start`, `:from-current` and `:from-end`. These denote the start of the file, the current position in the file and the end of the file.

If *`offset`* is zero, and *`whence`* is `:from-current`, then `seek-stream` returns the current absolute position within the stream, if it can successfully obtain it. Otherwise, it returns `t` if it is successful.

If a character has been successfully put back into a text stream with `unget-char` and is still pending, then the position value is unspecified. If a byte has been put back into a binary stream with `unget-byte`, and the previous position wasn't zero, then the position is decremented by one.

On failure, it throws an exception of type `stream-error`.

9.44.42 Function `truncate-stream`

Syntax:

```
(truncate-stream stream [length])
```

Description:

The `truncate-stream` causes the length of the underlying file associated with *`stream`* to be set to *`length`* bytes.

The stream must be a file stream, and must be open for writing.

If *`length`* is omitted, then it defaults to the current position, retrieved as if by invoking the `seek-stream` with an *`offset`* argument of zero and *`whence`* argument of `:from-current`. Hence, after the `truncate-stream` operation, that position is one byte past the end of the file.

9.44.43 Functions `stream-get-prop` and `stream-set-prop`

Syntax:

```
(stream-get-prop stream indicator)
(stream-set-prop stream indicator value)
```

Description:

These functions get and set properties on a stream. Only certain properties are meaningful with certain kinds of streams, and the meaning depends on the stream. If two or more stream types support a property of the same name, it is expected that the property has the same or similar meaning for both streams to the maximum extent that similarity is possible.

The `stream-set-prop` function sets a property on a stream. The *`indicator`* argument is a symbol, usually a keyword symbol, denoting the property, and *`value`* is the property value. If the stream understands and accepts the property, the function returns `t`. Otherwise it returns `nil`.

The `stream-get-prop` function inquires about the value of a property on a stream. If the stream understands the property, then it returns its current value. If the stream does not understand a property, nil is returned, which is also returned if the property exists, but its value happens to be `nil`.

The `:name` property is widely supported by streams of various types. It associates the stream with a name. This property is not always modifiable.

File, process and stream socket I/O streams have a `:fd` property which can be accessed, but not modified. It retrieves the same value as the `fileno` function.

The "real time" property supported by these streams, connected with the `real-time-stream-p` function, also appears as the `:real-time` property.

I/O streams also have a property called `:byte-oriented` which, if set, suppresses the decoding of UTF-8 on character input. Rather, each byte of the file corresponds directly to one character. Bytes in the range 1 to 255 correspond to the character code points U+0001 to U+00FF. Byte value 0 is mapped to the code point U+DC00.

The logging priority of the `*stdlog*` syslog stream is controlled by the `:prio` property.

If *`stream`* is a catenated stream (see the function `make-catenated-stream`) then these functions transparently operate on the current head stream of the catenation.

9.44.44 Functions `make-catenated-stream` and `cat-streams`

Syntax:

```
(make-catenated-stream stream*)
(cat-streams stream-list)
```

Description:

The `make-catenated-stream` function takes zero or more arguments which are input streams of the same type, and combines them into a single virtual stream called a catenated stream.

The `cat-streams` function takes a single list of input streams of the same type, and similarly combines them into a catenated stream.

A catenated stream does not support seeking operations or output, regardless of the capabilities of the streams in the list.

If the stream list is not empty, then the leftmost element of the list is called the head stream.

The `get-char`, `get-byte`, `get-line`, `unget-char` and `unget-byte` functions delegate to the corresponding operations on the head stream, if it exists. If the stream list is empty, they return `nil` to the caller.

If the `get-char`, `get-byte` or `get-line` operation on the head stream yields `nil`, and there are more lists in the stream, then the stream is closed, removed from the list, and the next stream, if any, becomes the head list. The operation is then tried again. If any of these operations fail on the last list, it is not removed from the list, so that a stream remains in place which can take the `unget-char` or `unget-byte` operations.

In this manner, the catenated streams appear to be a single stream.

Note that the operations can fail due to being unsupported. It is the caller's responsibility to make sure all of the streams in the list are compatible with the intended operations.

If the stream list is empty then an empty catenated stream is produced. Input operations on this stream yield `nil`, and the `unget-char` and `unget-byte` operations throw an exception.

9.44.45 Function `catenated-stream-p`

Syntax:

```
(catenated-stream-p obj)
```

Description:

The `catenated-stream-p` function returns `t` if *obj* is a catenated stream. Otherwise it returns `nil`.

9.44.46 Function `catenated-stream-push`

Syntax:

```
(catenated-stream-push new-stream cat-stream)
```

Description:

The `catenated-stream-push` function pushes *new-stream* to the front of the stream list inside *cat-stream*.

If an `unget-byte` or `unget-char` operation was successfully performed on *`cat-stream`* previously to a call to `catenated-stream-push`, those operations were forwarded to the front stream. If those bytes or characters are still pending, they are pending inside that stream, and thus are logically preceded by the contents of *`new-stream`*.

9.44.47 Functions `open-files` and `open-files*`

Syntax:

```
(open-files path-list [alternative-stream [mode-string]])
(open-files* path-list [alternative-stream [mode-string]])
```

Description:

The `open-files` and `open-files*` functions create a list of streams by invoking the `open-file` function on each element of *`path-list`*. By default, the mode string `"r"` is passed to `open-file`; if the *`mode-string`* argument specified, it overrides this default. In that situation, the specified mode should permit reading.

These streams are turned into a catenated stream as if they were the arguments of a call to `make-catenated-stream`.

The effect is that multiple files appear to be catenated together into a single input stream.

If the optional *`alternative-stream`* argument is supplied, then if *`path-list`* is empty, *`alternative-stream`* is returned instead of an empty catenated stream.

The difference between `open-files` and `open-files*` is that `open-files` creates all of the streams up-front. So if any of the paths cannot be opened, the operation throws. The `open-files*` variant is lazy: it creates a lazy list of streams out of the path list. The streams are opened as needed: before the second stream is opened, the program has to read the first stream to the end, and so on.

Example:

Collect lines from all files that are given as arguments on the command line. If there are no files, then read from standard input:

```
@(next (open-files *args* *stdin*))
@(collect)
@line
@(end)
```

9.44.48 Function `path-equal`

Syntax:

```
(path-equal left-path right-path)
```

Description:

The `path-equal` function determines whether the two paths *`left-path`* and *`right-path`* are equal under a certain definition of equivalence, whose requirements are given below. The function returns `t` if the paths are equal, otherwise `nil`.

If *`left-path`* and *`right-path`* are strings which are identical under the `equal` function, then they are considered equal paths.

Otherwise, the two paths are equal if the relative path from *left-path* to *right-path* is `"."` (dot), as would be determined by the `path-rel` function, if it were applied to *left-path* and *right-path* as its arguments. If `path-rel` would return the dot path, then the two paths are equal. If `path-rel` would return any other value, or throw an exception, then the paths are unequal.

Examples:

```
;; simple case
(path-equal "a" "a") -> t
(path-equal "a" "b") -> nil

;; trailing slashes don't matter
(path-equal "a" "a/") -> t
(path-equal "a/" "a/") -> t

;; .. components resolved:
(path-equal "a/b/../c" "a/c") -> t

;; . components resolved:
(path-equal "a" "a/././.") -> t
(path-equal "a/." "a/././.") -> t

;; (On Microsoft Windows)
;; different drive:
(path-equal "c:/a" "d:/b/../a") -> nil
;; same drive:
(path-equal "c:/a" "c:/b/../a") -> t
```

9.44.49 Functions `abs-path-p` and `portable-abs-path-p`

Syntax:

```
(abs-path-p path)
(portable-abs-path-p path)
```

Description:

The `abs-path-p` and `portable-abs-path-p` functions test whether the argument *path* is an absolute path, returning a `t` or `nil` indication.

The `portable-abs-path-p` function behaves in the same manner on all platforms, implementing a platform-agnostic definition of *absolute path*, as follows.

An absolute path is a string which either begins with a slash or backslash character, or which begins with an alphanumeric word, followed by a colon, followed by a slash or backslash.

The empty string isn't an absolute path.

Examples of absolute paths under `portable-abs-path-p`:

```
/etc
c:/tmp
ftp://user@server
disk0:/home
Z:\Users
```

Examples of strings which are not absolute paths:

```
.
abc
foo:bar/x
$:\abc
```

The `abs-path-p` is similar to `portable-abs-path-p` except that it reports false for paths which are not absolute paths according to the host platform. The following paths are not absolute on POSIX platforms:

```
c:/tmp
ftp://user@server
disk0:/home
Z:\Users
```

9.44.50 Function `pure-rel-path-p`

Syntax:

```
(pure-rel-path-p path)
```

Description:

The `pure-rel-path-p` function tests whether the string *`path`* represents a *pure relative path*, which is defined as a path which isn't absolute according to `abs-path-p`, which isn't the string **`"."`** (single period), which doesn't begin with a period followed by a slash or backslash, and which doesn't begin with an alphanumeric word terminated by a colon.

The empty string is a pure relative path.

Other examples of pure relative paths:

```
abc.d
.tmp/bar
1234
x
$:/xyz
```

Examples of strings which are not pure relative paths:

```
.
/
/etc
./abc
.\
foo:
$:\abc
```

9.44.51 Functions `dir-name` and `base-name`

Syntax:

```
(dir-name path)
(base-name path [suffix])
```

Description:

The `dir-name` and `base-name` functions calculate, respective, the directory part and base name part of a pathname.

The calculation is performed in a platform-dependent way, using the characters in the variable `path-sep-chars` as path component separators.

Both functions first remove from any further consideration all superfluous trailing occurrences of the directory separator characters from `path`. Thus input such as `"a////"` is reduced to just `"a"`, and `"///"` is reduced to `"/"`.

The resulting trimmed path is the *effective path*.

If the effective path is an empty string, then `dir-name` returns `"."` and `base-name` returns the empty string.

If the effective path is not empty, and contains no path separator characters, then `dir-name` returns `"."` and `base-name` returns the effective path.

Otherwise, the effective path is divided into two parts: the *raw directory prefix* and the remainder.

The raw directory prefix is the maximally long prefix of the effective path which ends in a separator character.

The `dir-name` function returns the raw directory prefix, if that prefix consists of nothing but a single directory separator character. Otherwise it returns the raw directory prefix, with the trailing path separator removed.

The `base-name` function returns the remaining part of the effective path, after the raw directory prefix.

If the *`suffix`* argument is given to `base-name`, it specifies a proper suffix to be removed from the returned base name. First, the base name is calculated according to the foregoing rules. Then, if *`suffix`* matches a trailing portion of the base name, but not the entire base name, it is removed from the base name.

The *`suffix`* parameter may be given a `nil`, argument, which is treated exactly as if it were absent. Note: this requirement allows for the following idiom to work correctly even in cases when `p` has no suffix:

```
;; calculate base name of p with short suffix removed
(base-name p (short-suffix p))

;; calculate base name of p with long suffix removed
(base-name p (long-suffix p))
```

Examples:

```
(base-name "") -> ""
(base-name "/") -> "/"
(base-name ".") -> "."
(base-name "./") -> "."
(base-name "a") -> "a"
(base-name "/a") -> "a"
```

```
(base-name "/a/") -> "a"
(base-name "/a/b") -> "b"
(base-name "/a/b/") -> "b"
(base-name "/a/b///") -> "b"

;; with suffix
(base-name "" "") -> ""
(base-name "/" "/") -> "/"
(base-name "/" "") -> "/"
(base-name "." ".") -> "."
(base-name "." "") -> "."
(base-name "./" "/") -> "."
(base-name "a" "a") -> "a"
(base-name "a" "") -> "a"
(base-name "a.b" ".b") -> "a"
(base-name "a.b/" ".b") -> "a"
(base-name "a.b/" ".b/") -> "a.b"
(base-name "a.b/" "a.b") -> "a.b"
```

9.44.52 Functions `long-suffix` and `short-suffix`

Syntax:

```
(long-suffix path [alt])
(short-suffix path [alt])
```

Description:

The `long-suffix` and `short-suffix` functions calculate the *long suffix* and *short suffix* of *`path`*, which must be a string.

If *`path`* does not contain any occurrences of the character `.` (period) in the role of a suffix delimiter, then *`path`* does not have a suffix. In this situation, both functions return the *`alt`* argument, which defaults to `nil` if it is omitted.

What it means for *`path`* to have a suffix delimiter is that the `.` character occurs somewhere in the last component of *`path`*, other than as the first character of that component. What constitutes the last component is specified in more detail below.

If a suffix delimiter is present, then the long or short suffix is the substring of *`path`* which includes the delimiting period and all characters which follow, except that if *`path`* ends in a sequence of one or more path separator characters, those characters are omitted from the returned suffix.

If multiple periods occur in the last component of the path, the delimiter for the long suffix is the leftmost period and the delimiter for the short suffix is the rightmost period.

If the delimiting period is the rightmost character of *`path`*, or occurs immediately before a trailing path separator, then the suffix delimited by that period is the period itself.

If *`path`* contains only one suffix delimiter, then its long and short suffix coincide.

For the purpose of identifying the last component of *`path`*, if *`path`* ends a sequence of one or more path-separator characters, then those characters are removed from consideration. If the remaining string contains path-separator characters, then the last component consists of that portion of it which follows the rightmost path-separator character. Otherwise, the last component is

the entire string. The suffix, if present, is identified and extracted from this last component.

Examples:

```
(short-suffix "") -> nil
(short-suffix ".") -> nil
(short-suffix "abc") -> nil
(short-suffix ".abc") -> nil
(short-suffix "/.abc") -> nil
(short-suffix "abc" "") -> ""
(short-suffix "abc.") -> "."
(short-suffix "abc.tar") -> ".tar"
(short-suffix "abc.tar///") -> ".tar"
(short-suffix "abc.tar.gz") -> ".gz"
(short-suffix "abc.tar.gz/") -> ".gz"
(short-suffix "x.y.z/abc.tar.gz/") -> ".gz"
(short-suffix "x.y.z/abc.tar.gz//") -> nil

(long-suffix "") -> nil
(long-suffix ".") -> nil
(long-suffix "abc") -> nil
(long-suffix ".abc") -> nil
(long-suffix "/.abc") -> nil
(long-suffix "abc.") -> "."
(long-suffix "abc.tar") -> ".tar"
(long-suffix "abc.tar///") -> ".tar"
(long-suffix "abc.tar.gz") -> ".tar.gz"
(long-suffix "abc.tar.gz/") -> ".tar.gz"
(long-suffix "x.y.z/abc.tar.gz/") -> ".tar.gz"
```

9.44.53 Functions `trim-long-suffix` and `trim-short-suffix`

Syntax:

```
(trim-long-suffix path)
(trim-short-suffix path)
```

Description:

The `trim-long-suffix` and `trim-short-suffix` functions calculate the portion of *`path` long suffix* and *short suffix* of the string argument *`path`*, and return a path with the suffix removed.

Respectively, `trim-long-suffix` and `trim-short-suffix` calculate the suffix in exactly the same manner as `long-suffix` and `short-suffix`.

If *`path`* is found not to contain a suffix, then it is returned.

If *`path`* contains a suffix, then a new string is returned from which the suffix is deleted. If the suffix is followed by one or more path separator characters, these are preserved in the return value.

Examples:

```
(trim-short-suffix "") -> ""
(trim-short-suffix "a") -> "a"
(trim-short-suffix ".") -> "."
```

```
(trim-short-suffix ".a") -> ".a"

(trim-short-suffix "a.") -> "a"
(trim-short-suffix "a.b") -> "a"
(trim-short-suffix "a.b.c") -> "a.b"

(trim-short-suffix "a./") -> "a/"
(trim-short-suffix "a.b/") -> "a/"
(trim-short-suffix "a.b.c/") -> "a.b/"

(trim-long-suffix "a.b.c") -> "a"
(trim-long-suffix "a.b.c/") -> "a/"
(trim-long-suffix "a.b.c///") -> "a///"
```

9.44.54 Function `add-suffix`

Syntax:

(`add-suffix` *`path suffix`*)

Description:

The `add-suffix` function combines the string arguments *`path`* and *`suffix`* in a way which harmonizes with the `long-suffix` and `short-suffix` functions.

If *`path`* does not end in a path separator character, that category being defined by the `path-sep-chars` variable, then `add-suffix` returns the trivial string catenation of *`path`* and *`suffix`*.

Otherwise, `add-suffix` returns a string formed by inserting *`suffix`* into *`path`* just prior to the sequence of trailing path separator characters. The returned string is a catenation of that portion of *`path`* which excludes the sequence of trailing path separators, followed by *`suffix`*, followed by the sequence of trailing path separators.

A path separator which occurs as a part of syntax that indicates an absolute pathname is not considered a trailing separator. A path which begins with a separator is absolute. Other platform-specific path patterns may constitute an absolute pathname.

Note: in cases when *`suffix`* does not begin with a period, or is inserted in such a way that it is the start of a path component, then the functions `long-suffix` and `short-suffix` will not recognize *`suffix`* in the resulting path.

Examples:

```
(add-suffix "" "") -> ""
(add-suffix "" "a") -> "a"
(add-suffix "." "a") -> ".a"
(add-suffix "." ".a") -> "..a"
(add-suffix "/" ".b") -> "/.b"
(add-suffix "//" ".b") -> "/.b/"
(add-suffix "//" "b") -> "/b/"
(add-suffix "a" "") -> "a"
(add-suffix "a" ".b") -> "a.b"
(add-suffix "a/" ".b") -> "a.b/"
(add-suffix "a//" ".b") -> "a.b//"
```

```
;; On MS Windows
(add-suffix "c://" "x") -> "c:/x/"
(add-suffix "host://" "x") -> "host://x"
(add-suffix "host:///" "x") -> "host://x/"
```

9.44.55 Function `path-cat`

Syntax:

```
(path-cat [dir-path {rel-path}*])
```

Description:

The `path-cat` function joins together zero or more paths, returning the combined path. All arguments are strings.

The following description defines the behavior when `path-cat` is given exactly two arguments, which are interpreted as *`dir-path`* and *`rel-path`*. A description of the variable-argument semantics follows.

Firstly, the two-argument `path-cat` is related to the functions `dir-name` and `base-name` in the following way: if *`p`* is some path denoting an object in the file system, then `(path-cat (dir-name p) (base-name p))` produces a path *`p*`* which denotes the same object. The paths *`p`* and *`p*`* might not be equivalent strings.

The `path-cat` function ensures that paths are joined without superfluous path-separator characters, regardless of whether *`dir-path`* ends in a separator.

If a separator must be added, the character `/` (forward slash) is always used, even on platforms where `\` (backslash) is also a pathname separator, and even if either argument includes backslashes.

The `path-cat` function eliminates trivial occurrences of the `.` (dot) path component. It preserves trailing separators in the following way: if *`rel-path`* ends in a path-separator character, then the returned string shall end in that character; and if *`rel-path`* vanishes entirely because it is equivalent to the dot, then the returned string is *`dir-name`* itself.

If *`dir-path`* is an empty string, then `rel-path` is returned, and vice versa.

The variadic semantics of `path-cat` are as follows.

If `path-cat` is called with no arguments at all, it returns the path `"."` (period) denoting the relative path of the current directory.

If `path-cat` is called with one argument, that argument is returned.

If `path-cat` is called with three or more arguments, a left-associative reduction takes place using the two-argument semantics. The first two arguments are catenated into a single path, which is then catenated with the third argument, and so on.

The above semantics imply that the following equivalence holds:

```
[reduce-left path-cat list]  <--> [apply path-cat list]
```

Examples:

```
(path-cat "" "")     -->  ""
(path-cat "" ".")    -->  "."
(path-cat "." "")    -->  "."
(path-cat "." ".")   -->  "."

(path-cat "abc" ".")  -->  "abc"
(path-cat "." "abc")  -->  "abc"

(path-cat "./" ".")   -->  "./"
(path-cat "."  "./")  -->  "./"

(path-cat "abc/" ".")  -->  "abc/"
(path-cat "./" "abc")  -->  "abc"

(path-cat "/" ".")    --> "/"

(path-cat "/" "abc")  --> "/abc"

(path-cat "ab/cd" "ef")  --> "ab/cd/ef"

(path-cat "a" "b" "c")  -->  "a/b/c"

(path-cat "a" "b" "" "c" "/")  -->  "a/b/c/"
```

9.44.56 Function `trim-path-seps`

Syntax:

```
(trim-path-seps path)
```

Description:

The `trim-path-seps` function removes a consecutive run of one or more trailing separators from the end of the input string *path*.

The function treats the
path in a system-independent way: both the backslash and forward slash are considered a trailing separator.

The function preserves any necessary trailing separators, such as that of the absolute path `"/"` or the trailing slashes in volume absolute paths such as `"c:/"`.

Examples:

```
(trim-path-seps "") -> ""
(trim-path-seps "/") -> "/"
(trim-path-seps "//") -> "/"
(trim-path-seps "a///") -> "a"
(trim-path-seps "/a///") -> "/a")

(trim-path-seps "\\") -> "\\"
(trim-path-seps "\\\\") -> "\\"
(trim-path-seps "\\a\\\\\\") -> "\\a")

(trim-path-seps "c:/") -> "c:/"
```

```
(trim-path-seps "c://") -> "c:/"
(trim-path-seps "c:///") -> "c:/"
(trim-path-seps "c:a///") -> "c:a"

;; not a volume prefix:
(trim-path-seps "/c:/a///") -> "/c:/a"
(trim-path-seps "/c://///") -> "/c:")

(trim-path-seps "c:\\") -> "c:\\"
(trim-path-seps "c:\\\\") -> "c:\\"
(trim-path-seps "c:a\\\\\\") -> "c:a"

;; mixtures
(trim-path-seps "c:/\\/\\/") -> "c:/"
```

9.44.57 Function `rel-path`

Syntax:

```
(rel-path from-path to-path)
```

Description:

The `rel-path` function calculates the relative path between two file system locations indicated by string arguments *`from-path`* and *`to-path`*. The *`from-path`* is assumed to be a directory. The return value is a relative path which could be used to access an object named by *`to-path`* if *`from-path`* were the current working directory.

The calculation performed by `rel-path` is a pure calculation; it has no interaction with the host operating system. No component of either input path has to exist. Symbolic links are not resolved. This can lead to incorrect results, as noted below.

Either both the inputs must be absolute paths, or must both be relative, otherwise an error exception is thrown.

On the MS Windows platform, if one input specifies a drive letter prefix, the other input must specify the same prefix, or else an error exception is thrown; there is no relative path between locations on different drives. The behavior is unspecified if the arguments are two UNC paths indicating different hosts.

The `rel-path` function first splits both paths into components according to the platform-specific pathname separators indicated by the `path-sep-chars` variable.

Next, it eliminates all empty components, `.` (dot) components and `..` (dotdot) components from both separated paths. All dot components are removed, and any component which is neither dot nor dotdot is removed if it is followed by dotdot.

Then, a common prefix is determined between the two component sequences, and a relative component sequence is calculated from them as follows:

If the component sequence corresponding to *`from-path`* is longer than the common prefix, then the excess part of that sequence after the common prefix must not contain any `..` (dotdot) components, or else an error exception is thrown. Otherwise, every component in this excess part of the *`from-path`* component sequence is converted to `..` in order to express the relative navigation from *`from-path`* up to the directory indicated by the common prefix.

Next, if the component sequence corresponding to *to-path* has any components in excess of the common prefix, those excess components are appended to this possibly empty sequence of dotdot components, in order to express navigation from the common prefix down to the *to-path* object. This excess sequence coming from *to-path* may include `..` components.

Finally, if the resulting sequence is nonempty, it is joined together using the leftmost path separator character indicated in `path-sep-chars` and returned. If it is empty, then the string `"."` is returned.

Note: because the function doesn't access the file system and in particular does not resolve symbolic links or other indirection devices, the result may be incorrect. For example, suppose that the current working directory contains a symbolic link called `up` which expands to `..` (dotdot). The expression `(rel-path "up/a" "../a")` is oblivious to this, and calculates `"../../../a"`. The correct result in light of `up` being an alias for `..` calls for a return value of `"."`. The exact problem is that any symbolic links in the excess part of *from-path* after the common prefix are assumed by `rel-path` to be simple subdirectory names, which can be navigated in reverse using a `..` link. This reverse navigation assumption is false for any symbolic link which which does not act as an alias for a subdirectory in the same location.

In situations where this possibility exists, it is recommended to use `realpath` function to canonicalize the input paths.

The following is an example of the algorithm being applied to arguments `"a/d/../b/x/y/"` and `"a/b/w"`, where the assumption is that this is on a POSIX platform where the leftmost character in `path-sep-chars` is `/`:

Firstly, both inputs are converted to component sequences, those respectively being:

```
("a" "d" ".." "b" "x" "y" "")
("a" "b" "w")
```

Next the `..` and empty components are removed:

```
("a" "b" "x" "y")
("a" "b" "w")
```

At this point, the common prefix is identified:

```
("a" "b")
```

The *from-path* has two components in excess of the prefix:

```
("x" "y")
```

which are each replaced by `".."`.

The *to-path* has one component in excess of the common prefix, `"w"`.

These two sequences are appended together:

```
(".." ".." "w")
```

The resulting path is then formed by joining these with the separator character, resulting in the relative path `"../../w"`.

Examples:

```
;; mixtures of relative and absolute
(rel-path "/abc" "abc") -> ;; error
(rel-path "abc" "/abc") -> ;; error

;; dotdot in excess part of from path:
(rel-path "../../x" "y") -> ;; error

(rel-path "." ".") -> "."
(rel-path "./abc" "abc") -> "."
(rel-path "abc" "./abc") -> "."
(rel-path "./abc" "./abc") -> "."
(rel-path "abc" "abc") -> "."
(rel-path "." "abc") -> "abc"
(rel-path "abc/def" "abc/ghi") -> "../ghi"
(rel-path "xyz/../abc/def" "abc/ghi") -> "../ghi"
(rel-path "abc" "d/e/f/g/h") -> "../d/e/f/g/h"
(rel-path "abc" "d/e/../g/h") -> "../d/g/h"
(rel-path "d/e/../g/h" ".") -> "../../.."
(rel-path "d/e/../g/h" "a/b") -> "../../../a/b"
(rel-path "x" "../../../y") -> "../../../../y"
(rel-path "x///" "x") -> "."
(rel-path "x" "x///") -> "."
(rel-path "///x" "/x") -> "."
```

9.44.58 Variable `path-sep-chars`

Description:

The `path-sep-chars` variable holds a string consisting of the characters which the underlying operating system recognizes as pathname separators.

If a particular of these characters is considered preferred on the host platform, that character is placed in the first position of `path-sep-chars`.

Altering the value of this variable has no effect on any **TXR Lisp** library function.

9.44.59 Functions `read` and `iread`

Syntax:

```
(read [source
       [err-stream [err-retval [name [lineno]]]]])
(iread [source
        [err-stream [err-retval [name [lineno]]]]])
```

Description:

The `read` function converts text denoting **TXR Lisp** structure, into the corresponding data structure. The *`source`* argument may be either a character string, or a stream. If it is omitted, then `*stdin*` is used as the stream.

The *`source`* must provide the text representation of one complete **TXR Lisp** object. If *`source`* and the function being applied is `read`, then if the object is followed by any non-whitespace material, the situation is treated as a syntax error, even if that material is a syntactically valid additional object. The `iread` function ignores this situation. Other differences between `read` and

iread are given below.

Multiple calls to read on the same stream will extract successive objects from the stream. To parse successive objects from a string, it is necessary to convert it to a string stream.

The optional *err-stream* argument can be used to specify a stream to which diagnostics of parse errors are sent. If absent, the diagnostics are suppressed.

The optional *name* argument can be used to specify the file name which is used for reporting errors. If this argument is missing, the name is taken from the name property of the *source* argument if it is a stream, or else the word string is used as the name if *source* is a string.

The optional lineno argument, defaulting to 1, specifies the starting line number. This, like the *name* argument, is used for reporting errors.

If there are no parse errors, the function returns the parsed data structure. If there are parse errors, and the *err-retval* parameter is present, its value is returned. If the *err-retval* parameter is not present, then an exception of type syntax-error is thrown.

The iread function ("interactive read") is similar to read except that it parses a modified version of the syntax. The modified syntax does not support the application of the dot and dotdot operators on a top-level expression. For instance, if the input is a.b or a .. b then iread will only read the a token whereas read will read the entire expression.

This modified syntax allows iread to return immediately when an expression is recognized, which is the expected behavior if the input is being read from an interactive terminal. By contrast, read waits for more input after seeing a complete expression, because of the possibility that the expression will be further extended by means of the dot or dotdot operators. An explicit end-of-input signal must be given from the terminal to terminate the expression.

The special variable *rec-source-loc* controls whether these functions record source location info similarly to load. Note: if these functions are used to scan data which is evaluated as Lisp code, it may be useful to set *rec-source-loc* true in order to obtain better diagnostics. However, source location recording incurs a performance and storage penalty.

9.44.60 Function parse-errors

Syntax:

```
(parse-errors stream)
```

Description:

The parse-errors function retrieves information, from a *stream*, pertaining to the status of the most recent parsing operation performed on that stream: namely, a previous call to read, iread or get-json.

If the *stream* object has not been used for parsing, or else the most recent parsing operation did not encounter errors, then parse-errors returns nil.

If the most recent parsing operation on *stream* encountered errors, then parse-errors function returns a positive integer value indicating the error count. Otherwise it returns nil.

If a parse error operation encounters a syntax error before obtaining any token from the stream, then the error count is zero and parse-errors returns nil. Consequently, parse-errors may be used after a failed parse operation to distinguish a true syntax error from an end-of-stream

condition.

9.44.61 Function `record-adapter`

Syntax:

```
(record-adapter regex [stream [include-match]])
```

Description:

The `record-adapter` function returns a new stream object which acts as an *adapter* to the existing *`stream`*.

If an argument is not specified for *`stream`*, then the `*std-input*` stream is used.

With the exception of *`get-line`*, all operations on the returned adapter transparently delegate to the original *`stream`* object.

When the `get-line` function is used on the adapter, it behaves differently. A string is extracted from *`stream`*, and returned. However, the string isn't a line delimited by a newline character, but rather a record delimited by *`regex`*. This record is extracted as if by a call to the `read-until-match` function, invoked with the *`regex`*, *`stream`* and *`include-match`* arguments.

All behavior which is built on the `get-lines` function is affected by the record-delimiting semantics of a record adapter's `get-line` implementation. Notably, the `get-lines` and `lazy-stream-cons` functions return a lazy list of delimited records rather than of lines.

9.45 Stream Output Indentation

TXR Lisp streams provide support for establishing hanging indentations in text output. Each stream which supports output has a built-in state variable called indentation mode, and another variable indicating the current indentation amount. When indentation mode is enabled, then prior to the first character of every line, the stream prepends the indentation: space characters equal in number to the current indentation value. This logic is implemented by the `put-char` and `put-string` functions, and all functions based on these. The `put-byte` function does not interact with indentation. The column position tracking will be incorrect if byte and character output are mixed, affecting the placement of indentation.

Indentation mode takes on four numeric values, given by the four variables `indent-off`, `indent-data`, `indent-code` and `indent-foff`. As far as stream output is concerned, the code and data modes represented by `indent-code` and `indent-data` behave the same way: both represent the "indentation turned on" state. The difference between them influences the behavior of the `width-check` function. This function isn't used by any lower-level stream output routines. It is used by the object printing functions like `print` and `pprint` to break up long lines. The `indent-off` and `indent-foff` modes are also treated the same way by lower level stream output, indicating "indentation turned off". The modes are distinguished by `print` and `pprint` in the following way: `indent-off` is a "soft" disable which allows these object-printing routines to temporarily turn on indentation while traversing aggregate objects. Whereas the `indent-foff` ("force off") value is a "hard" disable: the object-printing routines will not enable indentation and will not break up long lines.

9.45.1 Variables `indent-off`, `indent-data`, `indent-code` and `indent-foff`

Description:

These variables hold integer values representing output stream indentation modes. The value of `indent-off` is zero.

9.45.2 Functions `get-indent-mode` and `set-indent-mode`

Syntax:

```
(get-indent-mode stream)
(set-indent-mode stream new-mode)
(test-set-indent-mode stream compare-mode new-mode)
```

Description:

These functions retrieve and manipulate the stream indent mode. The `get-indent-mode` retrieves the current indent mode of *`stream`*. The `set-indent-mode` function sets the indent mode of *`stream`* to *`new-mode`* and returns the previous mode.

Note: it is encouraged to save and restore the indentation mode, and in a way that is exception safe. If a block of code sets up indentation on a stream such as `*stdout*` and is terminated by an exception, the indentation will remain in effect and affect subsequent output. The `with-resources` macro or `unwind-protect` operator may be used.

9.45.3 Functions `test-set-indent-mode` and `test-neq-set-indent-mode`

Syntax:

```
(test-set-indent-mode stream compare-mode new-mode)
(test-neq-set-indent-mode stream compare-mode new-mode)
```

Description:

The `test-set-indent-mode` function sets the indent mode of *`stream`* to *`new-mode`* if and only if its current mode is equal to *`compare-mode`*. Whether or not it changes the mode, it returns the previous mode.

The `test-neq-set-indent-mode` only differs in that it sets *`stream`* to *`new-mode`* if and only if the current mode is **not** equal to *`compare-mode`*.

9.45.4 Functions `get-indent`, `set-indent`, `inc-indent` and `inc-indent-abs`

Syntax:

```
(get-indent stream)
(set-indent stream new-indent)
(inc-indent stream indent-delta)
(inc-indent-abs stream indent-delta)
```

Description:

These functions manipulate the indentation value of the stream. The indentation takes effect the next time a character is output following a newline character.

The `get-indent` function retrieves the current indentation amount.

The `set-indent` function sets *`stream`*'s indentation to the value *`new-indent`* and returns the previous value. Negative values are clamped to zero.

The `inc-indent` function sets *`stream`*'s indentation relative to the current printing column position, and returns the old value. The indentation is calculated by adding *`indent-delta`* to the current column position. If a negative indentation results, it is clamped to zero.

The `inc-indent-abs` function sets *`stream`*'s indentation relative to the current indentation value. The indentation is calculated by adding *`indent-delta`* to the current indentation

amount. If a negative indentation results, it is clamped to zero.

9.45.5 Function `width-check`

Syntax:

```
(width-check stream alt-char)
```

Description:

The `width-check` function examines the state of the stream, taking into consideration the current printing column position, the indentation state, the indentation amount and an internal "force break" flag. It makes a decision either to introduce a line break by printing a newline character, or else to print the *`alt-char`* character.

If a decision is made not to emit a line break, but *`alt-char`* is `nil`, then the function has no effect at all.

The return value is `t` if the function has issued a line break, otherwise `nil`.

9.45.6 Function `force-break`

Syntax:

```
(force-break stream)
```

Description:

If the `force-break` function is called on a stream, it sets an internal "force break" flag which affects the future behavior of `width-check`. The `width-check` function examines this flag. If the flag is set, `width-check` clears it, and issues a line break without considering any other conditions.

The *`stream`*'s `force-break` flag is also cleared whenever a newline character is output.

The `force-break` function returns `stream`.

Note: the `force-break` is involved in line breaking decisions. Whenever a list or list-like syntax is being printed, whenever an element of that syntax is broken into multiple lines, a break is forced after that element, in order to avoid output which resembles the following diagonally-creeping pattern:

```
(a b c (d e f
        g h i) j (k l
                  m n) o)
```

but instead is rendered in a more horizontally compact pattern:

```
(a b c (d e f
        g h i)
 j (k l
    m n)
 o)
```

When the printer prints `(d e f g h i)` it uses the `width-check` function between the elements; that function issues the break between the `f` and `g`. The printer monitors the return value of `width-check`; it knows that since one of the calls returned `t`, the object had been broken into two or more lines. It then calls `force-break` after printing the last element `i` of that object.

Then, due to the force flag, the outer recursion of the printer which is printing `(a b c ...)` will experience a break when it calls `width-check` before printing `j`.

Custom `print` methods defined on structure objects can take advantage of `width-check` and `force-break` in the same way so that user-defined output integrates with the formatting algorithm.

9.46 Stream Output Limiting

Streams have two properties which are used by the **TXR Lisp** object printer to optionally truncate the output generated by aggregate objects.

A stream can specify a maximum length for aggregate objects via the `set-max-length` function. Using the `set-max-depth` function, the maximum depth can also be specified.

This feature is useful when diagnostic output is being produced, and the objects involved are so large that the diagnostic output overwhelms the output device or the user, so as to become uninformative. Output limiting also prevents the printer's non-termination on infinite, lazy structures.

It is recommended that functions which operate on streams passed in as parameters save and restore these parameters, if they need to manipulate them, for instance using `with-resources`:

```
(defun output-function (arg stream)
  ;; temporarily impose maximum width and depth
  (with-resources ((ml (set-max-length stream 42)
                       (set-max-length stream ml))
                   (mw (set-max-depth stream 12)
                       (set-max-depth stream mw)))
    (prinl arg stream)
    ...))
```

9.46.1 Function `set-max-length`

Syntax:

```
(set-max-length stream value)
```

Description:

The `set-max-length` function establishes the maximum length for aggregate object printing. It affects the printing of lists, vectors, hash tables, strings as well as quasiliterals and quasiword list literals (QLLs).

The default value is 0 and this value means that no limit is imposed. Otherwise, the value must be a positive integer.

When the list, vector or hash-table object being printed has more elements than the maximum length, then elements are printed only up to the maximum count, and then the remaining elements are summarized by printing the `...` (three dots) character sequence as if it were an additional element. This sequence is an invalid token; it cannot be read as input.

When a character string is printed, and the maximum length parameter is nonzero, a maximum character count is determined as follows. Firstly, if the maximum length value is less than 3, it is taken to be 3. Then it is multiplied by 8. Thus, a maximum length of 10 allows 80 characters, whereas a maximum length of 1 allows 24 characters.

If a string which exceeds the maximum number of characters is being printed with read-print consistency, as by the `print` function, then only a prefix of the string is printed, limited to the maximum number of characters. Then, the literal syntax is closed using the character sequence `\..."` (backslash, dot, dot, dot, double quote) whose leading invalid escape sequence `\.` (backslash, dot) ensures that the truncated object is not readable.

If a string which exceeds the maximum number of characters is being printed without read-print consistency, as by the `pprint` function, then only a prefix of the string is printed, limited to the maximum number of characters. Then the character sequence `...` is emitted.

Quasiliterals are treated using a combination of behaviors. Elements of a quasiliteral are literal sequence of text, and embedded variables and expressions. The maximum length specifies both the maximum number of elements in the quasiliteral, and the maximum number of characters in any element which is a sequence of text. When either limit is exceeded, the quasiliteral is immediately terminated with the sequence `\...` ` (escaped dot, dot, dot, backtick). The maximum character limit is applied to the units of text cumulatively, rather than individually. As in the case of string literals, the limit is determined by multiplying the length by 8, and clamping at a minimum value of 24.

When a QLL is printed, the space-separated elements of the literal are individually subject to the maximum character limit as if they were independent quasiliterals. Furthermore, the sequence of these elements is subject to the maximum length. If there are more elements in the QLL, then the sequence `\...` ` (escaped dot, dot, dot, backtick) is emitted and thus the QLL ends.

The `set-max-length` function returns the previous value.

9.46.2 Function `set-max-depth`

Syntax:

```
(set-max-depth stream value)
```

Description:

The `set-max-length` function establishes the maximum depth for the printing of nested objects. It affects the printing of lists, vectors, hash tables and structures. The default value is 0 and this value means that no limit is imposed. Otherwise, the value must be a positive integer.

The depth of an object not enclosed in any object is zero. The depth of the element of an aggregate is one greater than the depth of the aggregate itself. For instance, given the list `(1 (2 3))` the list itself has depth 0, the atom `1` has depth 1, as does the sublist `(2 3)`, and the `2` and `3` atoms have depth 2.

When an object is printed whose depth exceeds the maximum depth, then three dot character sequence `...` is printed instead of that object. This notation is an invalid token; it cannot be read as input.

Additionally, when a vector, list, hash table or structure is printed which itself doesn't exceed the maximum depth, but whose elements do exceed, then that object is summarized, respectively, as `(...)`, `#(...)`, `H#(...)` and `S#(...)`, rather than repeating the `...` sequence for each of its elements.

The `set-max-depth` function returns the previous value.

9.47 Coprocesses

9.47.1 Functions `open-command`, `open-process` and `open-subprocess`

Syntax:

```
(open-command system-command [mode-string])
(open-process program mode-string [argument-list])
(open-subprocess program mode-string
   [argument-list [function]])
```

Description:

These functions spawn external programs which execute concurrently with the **TXR** program. They all return a unidirectional stream for communicating with these programs: either an output stream, or an input stream, depending on the contents of *mode-string*.

In `open-command`, the *mode-string* argument is optional, defaulting to the value `"r"` if it is missing. See the `open-file` function for a discussion of modes. The `open-command` function is implemented using POSIX `popen`. Those elements of *mode-string* which are applicable to `popen` are passed to it, and hence their semantics follows from their processing in that function.

The `open-command` function accepts, via the *system-command* string parameter, a system command, which is in a system-dependent syntax. On a POSIX system, this would be in the POSIX Shell Command Language.

The `open-process` function specifies a program to invoke via the *command* argument. This is subject to the operating system's search strategy. On POSIX systems, if it is an absolute or relative path, it is treated as such, but if it is a simple base name, then it is subject to searching via the components of the PATH environment variable. If `open-process` is not able to find *program*, or is otherwise unable to execute the program, the child process will exit, using the value of the C variable `errno` as its exit status. This value can be retrieved via `close-stream`.

The *argument-list* argument is a list of strings which specifies additional optional arguments to be passed to the program. The *program* argument becomes the first argument, and *argument-list* becomes the second and subsequent arguments. If *argument-list* is omitted, it defaults to empty.

If a coprocess is open for writing (*mode-string* is specified as `"w"`), then writing on the returned stream feeds input to that program's standard input file descriptor. Indicating the end of input is performed by closing the stream.

If a coprocess is open for reading (*mode-string* is specified as `"r"`), then the program's output can be gathered by reading from the returned stream. When the program finishes output, it will close the stream, which can be detected as normal end of data.

The standard input and error file descriptors of an input coprocess are obtained from the streams stored in the `*stdin*` and `*stderr*` special variables, respectively. Similarly, the standard output and error file descriptors of an output coprocess are obtained from the `*stdout*` and `*stderr*` special variables. These variables must contain streams on which the `fileno` function is meaningful, otherwise the operation will fail. What this functionality means is that rebinding the special variables for standard streams has the effect of redirection. For example, the following two expressions achieve the same effect of creating a stream which reads the output of the `cat` program, which reads and produces the contents of the file `text-file`.

```
;; redirect input by rebinding *stdin*
```

```
(let ((*stdin* (open-file "text-file")))
  (open-command "cat"))

;; redirect input using POSIX shell redirection syntax
(open-command "cat < text-file")
```

The following is erroneous:

```
;; (let ((*stdin* (make-string-input-stream "abc")))
     (open-command "cat"))
```

A string input or output stream doesn't have an operating system file descriptor; it cannot be passed to a coprocess.

The streams `*stdin*`, `*stdout*` and `*stderr*` are not synchronized with their underlying file descriptors prior to the execution of a coprocess. It is up to the program to ensure that previous output to `*stdout*` or `*stderr*` is flushed, so that the output of the coprocess isn't reordered with regard to output produced by the program. Similarly, input buffered in `*stdin*` is not available to the coprocess, even though it has not yet been read by the program. The program is responsible for preventing this situation also.

If a coprocess terminates abnormally or unsuccessfully, an exception is raised.

The *`mode-string`* argument of `open-process` supports a special *`redirection`* syntax. This syntax specifies I/O redirections which are done in the context of the child process, before the specified program is executed. Instances of the syntax are considered options; if *`mode-string`* specifies a mode such as `r` that mode must precede the redirections. Redirections may be mixed with other options.

Up to four redirections may be specified using one of two forms: a short form or the long form. If more than four redirections are specified, the *`mode-string`* is considered ill-formed.

The short form of the syntax consists of three characters: the prefix character >, a single decimal digit indicating the file descriptor to be redirected, and then a third character which is either another digit, or else one of the two characters `n` or `x`. If the third character is a digit, it indicates the target file descriptor of the redirection. For instance `>21` indicates that file descriptor 2 is to be redirected to 1 (so that material written to standard error goes to the same destination as that written to standard output). If the third character is `n`, it means that the file descriptor will be redirected to the file `/dev/null`. For instance, `>2n` indicates that descriptor 2 (standard error) will be redirected to the null device. If the third character is `x`, it indicates that the file descriptor shall be closed. For instance `>0x` means to close descriptor 0 (standard input).

The long form of the syntax allows file descriptors that require more than one decimal digit. It consists of the same prefix character > which is immediately followed by an open parenthesis `(`. The parenthesis is immediately followed by one or more digits which give the to-be-redirected file descriptor. This is followed by one or more whitespace characters, and then either another multi-digit decimal file descriptor or one of the two letters `n` or `x`. This second element must be immediately followed by the closing parenthesis `)`. Thus `>21` and `>2n` may be written in the long form, respectively, as `>(2 1)` and `>(2 n)`, while `>(32 47)` has no short form equivalent. Multiple redirections may be specified, in any mixture of the long and short form. For instance `r>21>0n>(27 31)` specifies a process pipe that is open for reading, capturing the output of the process. In that process, standard error is redirected to standard output, standard input is connected to the null device, and descriptor 27 is redirected to descriptor 31.

The `open-subprocess` function is a variant of `open-process`. This function has all the same argument conventions and semantics as `open-process`, adding the *`function`* argument. If this argument isn't `nil`, then it must specify a function which can be called with no arguments. This function is called in the child process after any redirections are established, just before the program specified by the *`program`* argument is executed. Moreover, the `open-subprocess` function allows *`program`* to be specified as `nil` in which case *`function`* must be specified. When *`function`* returns, the child process terminates as if by a call to `exit*` with an argument of zero.

9.48 I/O-Related Convenience Functions

The functions in this group create a stream, perform an I/O operation on it, and ensure that it is closed, in one convenient operation. They operate on files or command streams.

Several other functions in this category exist, which operate with buffers. They are documented in the Buffer Functions subsection under the FOREIGN FUNCTION INTERFACE section.

Many of the functions described in this section take an optional *`mode-opts`* argument. If this is specified, it must be a string which follows the *`options`* portion of the *`mode-string`* syntax described for the `open-file` function. This string must not specify the `mode` part. If specified, the *`mode-opts`* must be compatible with the implied *`mode`*. Functions that write a file have an implied mode of `"w"`, those which append have an implied mode of `"a"`, and those which read have an implied mode of `"r"`. For instance, a *`mode-opts`* value of `"x"` is useful with `file-put-string` but not `file-get-string`,

9.48.1 Functions `file-get`, `file-get-string` and `file-get-lines`

Syntax:

```
(file-get name [mode-opts])
(file-get-string name [mode-opts])
(file-get-lines name [mode-opts])
```

Description:

The `file-get` function opens a text stream over the file indicated by the string argument *`name`* for reading, reads the printed representation of a **TXR Lisp** object from it, and returns that object, ensuring that the stream is closed.

The `file-get-string` is similar to `file-get` except that it reads the entire file as a text stream and returns its contents in a single character string.

The `file-get-lines` function opens a text stream over the file indicated by *`name`* and returns produces a lazy list of strings representing the lines of text of that file as if by a call to the `get-lines` function, and returns that list. The stream remains open until the list is consumed to the end, as indicated in the description of `get-lines`.

9.48.2 Functions `file-put`, `file-put-string` and `file-put-lines`

Syntax:

```
(file-put name obj [mode-opts])
(file-put-string name string [mode-opts])
(file-put-lines name list [mode-opts])
```

Description:

The `file-put`, `file-put-string` and `file-put-lines` functions open a text stream over the file indicated by the string argument *`name`*, write the argument object into the file in their

specific manner, and then close the file.

If the file doesn't exist, it is created. If it exists, it is truncated to zero length and overwritten.

The `file-put` function writes a printed representation of *obj* using the `prinl` function. The return value is that of `prinl`.

The `file-put-string` function writes *string* to the stream using the `put-string` function. The return value is that of `put-string`.

The `file-put-lines` function writes *list* to the stream using the `put-lines` function. The return value is that of `put-lines`.

9.48.3 Functions `file-append`, `file-append-string` and `file-append-lines`

Syntax:

```
(file-append name obj [mode-opts])
(file-append-string name string [mode-opts])
(file-append-lines name list [mode-opts])
```

Description:

The `file-append`, `file-append-string` and `file-append-lines` functions open a text stream over the file indicated by the string argument *name*, write the argument object into the stream in their specific manner, and then close the stream.

These functions are close counterparts of, respectively, `file-get`, `file-append-string` and `file-append-lines`.

These functions behave differently when the indicated file already exists. Rather than being truncated and overwritten, the file is extended by appending the new data to its end.

9.48.4 Functions `command-get`, `command-get-string` and `command-get-lines`

Syntax:

```
(command-get cmd [mode-opts])
(command-get-string cmd [mode-opts])
(command-get-lines cmd [mode-opts])
```

Description:

The `command-get` function opens text stream over an input command pipe created for the command string *cmd*, as if by the `open-command` function. It reads the printed representation of a **TXR Lisp** object from it, and returns that object, ensuring that the stream is closed.

The `command-get-string` is similar to `command-get` except that it reads the entire file as a text stream and returns its contents in a single character string.

The `command-get-lines` function opens a text stream over an input command pipe created for the command string *cmd* and returns produces a lazy list of strings representing the lines of text of that file as if by a call to the `get-lines` function, and returns that list. The stream remains open until the list is consumed to the end, as indicated in the description of `get-lines`.

9.48.5 Functions `command-put`, `command-put-string` and `command-put-lines`

Syntax:

```
(command-put cmd obj [mode-opts])
(command-put-string cmd string [mode-opts])
(command-put-lines cmd list [mode-opts])
```

Description:

The `command-put`, `command-put-string` and `command-put-lines` functions open an output text stream over an output command pipe created for the command specified in the string argument *cmd*, as if by the `open-command` function. They write the argument object into the stream in their specific manner, and then close the stream.

The `command-put` function writes a printed representation of *obj* using the `prinl` function. The return value is that of `prinl`.

The `command-put-string` function writes *string* to the stream using the `put-string` function. The return value is that of `put-string`.

The `command-put-lines` function writes *list* to the stream using the `put-lines` function. The return value is that of `put-lines`.

9.49 Buffer streams

A stream type exists which allows `buf` objects to be manipulated through the stream interface. A buffer stream is created using the `make-buf-stream` function, which can either attach the stream to an existing buffer, or create a new buffer that can later be retrieved from the stream using `get-buf-from-stream`.

Operations on the buffer stream treat the underlying buffer much like if it were a memory-based file. Unless the underlying buffer is a "borrowed buffer" referencing the storage belonging to another object (such as the buffer object produced by the `buf-d` FFI type's get semantics) the stream operations can change the buffer's size. Seeking beyond the end of the buffer an then writing one or more bytes extends the buffer's length, filling the newly allocated area with zero bytes. The `truncate-stream` function is supported also. Buffer streams also support the `:byte-oriented` property.

Macros `with-out-buf-stream` and `with-in-buf-stream` are provided to simplify the steps involved in using buffer streams in some common scenarios. Note that in spite of the naming of these macros there is only one buffer stream type, which supports bidirectional I/O.

9.49.1 Function `make-buf-stream`

Syntax:

```
(make-buf-stream [buf])
```

Description:

The `make-buf-stream` function return a new buffer stream. If the *buf* argument is supplied, it must be a `buf` object. The stream is then associated with this object. If the argument is omitted, a buffer of length zero is created and associated with the stream.

9.49.2 Function `get-buf-from-stream`

Syntax:

```
(get-buf-from-stream buf-stream)
```

Description:

The `get-buf-from-stream` returns the buffer object associated with *`buf-stream`* which must be a buffer stream.

9.49.3 Macros `with-out-buf-stream` and `with-in-buf-stream`

Syntax:

```
(with-out-buf-stream (var [buf-expr])
   body-form*)
(with-in-buf-stream (var buf-expr)
   body-form*)
```

Description:

The `with-out-buf-stream` and `with-in-buf-stream` macros both bind variable *`var`* to an implicitly created buffer stream, and evaluate zero or more *`body-form`*s in the environment where the variable is visible.

The *`buf-expr`* argument, which may be omitted in the use of the `with-out-buf-stream` macro, must be an expression which evaluates to a `buf` object.

The *`var`* argument must be a symbol suitable for naming a variable.

The implicitly allocated buffer stream is connected to the buffer specified by *`buf-expr`* or, when *`buf-expr`* is omitted, to a newly allocated buffer.

The code generated by the `with-out-buf-stream` macro, if it terminates normally, yields the buffer object as its result value.

The `with-in-buf-stream` returns the value of the last *`body-form`*, or else `nil` if no forms are specified.

Examples:

```
(with-out-buf-stream (*stdout* (make-buf 24))
  (put-string "Hello, world!"))
-> #b'48656c6c6f2c2077 6f726c6421000000 0000000000000000'

(with-out-buf-stream (*stdout*) (put-string "Hello, world!"))
-> #b'48656c6c6f2c2077 6f726c6421'
```

9.50 Foreign Pointers

9.50.1 The `cptr` type

Objects of type `cptr` are Lisp values which contain a foreign pointer ("C pointer"). This data type is used by the `dlopen` function and is generally useful in conjunction with the Foreign Function Interface (FFI). An arbitrary pointer emanating from a foreign function can be captured as a `cptr` value, which can be passed back into foreign code. For this purpose, there exits also a matching FFI type called `cptr`.

The `cptr` type supports a symbolic type tag, which defaults to `nil`. The type tag plays a role in FFI. The FFI `cptr` type supports a tag attribute. When a `cptr` object is converted to a foreign pointer under the control of the FFI type, and that FFI type has a tag other than `nil`, the object's tag must exactly match that of the FFI type, or the conversion throws an error. In the reverse direction, when a foreign pointer is converted to a `cptr` object under control of the FFI `cptr` type, the object inherits the type tag from the FFI

type.

Although `cptr` objects are conceptually non-aggregate values, corresponding to pointers, they are de facto aggregates due to their implementation as references to heap objects. When a `cptr` object is passed to a foreign function by pointer, for instance using a parameter of type `(ptr cptr)`, its internal pointer is potentially updated to the new value coming from the function.

9.50.2 Function `cptr-int`

Syntax:

```
(cptr-int integer [type-symbol])
```

Description:

The `cptr-int` function converts *`integer`* into a pointer in a system-specific way which is consistent with the system's addressing structure. Then it returns that pointer contained in a `cptr` object.

The *`integer`* argument must be an integer which is in range for a pointer value. Note: this range is wider than the `fixnum` range; a portion of the range of `bignum` integers can denote pointers.

An extended range of values is accepted. The entire addressable space may be expressed by non-negative values. A range of negative values also expresses a portion of the address space, in accordance with the platform's concept of a signed integer.

For instance, on a system with 32-bit addresses, the values 0 to 4294967295 express all of the addresses as a pure binary value. Furthermore, the values -2147483648 to -1 also express the upper part of this range, corresponding, respectively, to the addresses 2147483648 to 4294967295. On that platform, values of *`integer`* outside of the range -2147483648 to 4294967295 are invalid.

The *`type-symbol`* argument should be a symbol. If omitted, it defaults to `nil`. This symbol becomes the `cptr` object's type tag.

9.50.3 Function `cptr-obj`

Syntax:

```
(cptr-obj object [type-symbol])
```

Description:

The `cptr-obj` function converts *`object`* object directly to a `cptr`.

The *`object`* argument may be of any type.

The raw representation of *`object`* is simply stored in a new instance of `cptr` and returned.

The *`type-symbol`* argument should be a symbol. If omitted, it defaults to `nil`. This symbol becomes the `cptr` object's type tag.

The lifetime of the returned `cptr` object is independent from that of *`object`*. If the lifetime of *`object`* reaches its end before that of the `cptr`, the pointer stored inside the `cptr` becomes invalid.

9.50.4 Function `int-cptr`

Syntax:

```
(int-cptr cptr)
```

Description:

The `int-cptr` function retrieves the pointer value of the *cptr* object as an integer.

If an integer *n* is in a range convertible to `cptr` type, then the expression `(int-cptr (cptr-int n))` reproduces *n*.

9.50.5 Function `cptr-buf`

Syntax:

```
(cptr-buf buf [type-symbol])
```

Description:

The `cptr-buf` function returns a `cptr` object which holds a pointer to a buffer object's storage area. The *buf* argument must be of type `buf`.

The *type-symbol* argument should be a symbol. If omitted, it defaults to `nil`. This symbol becomes the `cptr` object's type tag.

The lifetime of the returned `cptr` object is independent from that of *buf*. If the lifetime of *buf* reaches its end before that of the `cptr`, the pointer stored inside the `cptr` becomes invalid.

9.50.6 Function `cptr-cast`

Syntax:

```
(cptr-cast type-symbol cptr)
```

Description:

The `cptr-cast` function produces a new `cptr` object which has the same pointer as *cptr* but whose type is given by *type-symbol*.

Casting *cptr* objects with `cptr-cast` circumvents the safety mechanism which `cptr` type tagging provides.

9.50.7 Function `copy-cptr`

Syntax:

```
(cptr-copy cptr)
```

Description:

The `copy-cptr` function creates a new `cptr` object similar to *cptr*, which has the same address and type symbol as *cptr*.

9.50.8 Function `cptr-zap`

Syntax:

```
(cptr-zap cptr)
```

Description:

The `cptr-zap` function changes the pointer value of the *`cptr`* object to the null pointer.

The *`cptr`* argument must be of `cptr` type.

The return value is *`cptr`* itself.

Note: it is recommended to use `cptr-zap` when the program has taken some action which invalidates the pointer value stored in a `cptr` object, where a risk exists that the value may be subsequently misused.

9.50.9 Function `cptr-free`

Syntax:

```
(cptr-free cptr)
```

Description:

The `cptr-free` function passes the *`cptr`* object's pointer to the C library `free` function. After this action, it behaves exactly like `cptr-zap`.

The *`cptr`* argument must be of `cptr` type.

The return value is *`cptr`* itself.

Note: this function is unsafe. If the pointer didn't originate from the `malloc` family of memory allocation functions, or has already been freed, or copies of the pointer exist which are still in use, the consequences are likely catastrophic.

9.50.10 Function `cptrp`

Syntax:

```
(cptrp value)
```

Description:

The `cptrp` function tests whether *`value`* is a `cptr`. It returns `t` if this is the case, `nil` otherwise.

9.50.11 Function `cptr-type`

Syntax:

```
(cptr-type cptr)
```

Description:

The `cptr-type` function retrieves the *`cptr`* object's type tag.

9.50.12 Function `cptr-get`

Syntax:

```
(cptr-get cptr [type])
```

Description:

The `cptr-get` function extracts a Lisp value by converting a C object at the memory location denoted by *`cptr`*, according to the FFI type *`type`*. The external representation at the specified

memory location is is scanned according to the *type* and converted to a Lisp value which is returned.

If the *type* argument is specified, it must be a FFI type object. If omitted, then the `cptr` object's type tag is interpreted as a FFI type symbol and resolved to a type; the resulting type, if one is found is substituted for *type*. If the lookup fails an error exception is thrown.

The *cptr* object must be of type `cptr` and point to a memory area suitably aligned for, and large enough to hold a foreign representation of *type*, at the byte offset indicated by the *offset* argument.

If *cptr* is a null pointer, an exception is thrown.

The `cptr-get` operation is similar to the "get semantics" performed by FFI in order to extract the return value of foreign function calls, and by the FFI callback mechanism to extract the arguments coming into a callback.

The *type* argument may not be a variable length type, such as an array of unspecified size.

Note: the functions `cptr-get` and `cptr-out` are useful in simplifying the interaction with "semi-opaque" foreign objects: objects which serve as API handles that are treated as opaque pointers in API argument calls, but which expose some internal members that the application must access directly. The `cptr` objects pass through the foreign API without undergoing conversion, as usual. The application uses these two functions to perform conversion as necessary. Under this technique, the description of the foreign object need not be complete. Structure members which occur after the last member that the application is interested in need not be described in the FFI type.

9.50.13 Function `cptr-out`

Syntax:

```
(cptr-out cptr obj [type])
```

Description:

The `cptr-out` function converts a Lisp value into a C representation, which is stored at the memory location denoted by *cptr*, according to the FFI type *type*. The function's return value is *obj*.

If the *type* argument is specified, it must be a FFI type object. If omitted, then the `cptr` object's type tag is interpreted as a FFI type symbol and resolved to a type; the resulting type, if one is found is substituted for *type*. If the lookup fails an error exception is thrown.

The *obj* argument must be an object compatible with the conversions implied by *type*.

The *cptr* object must be of type `cptr` and point to a memory area suitably aligned for, and large enough to hold a foreign representation of *type*, at the byte offset indicated by the *offset* argument.

If *cptr* is a null pointer, an exception is thrown.

It is assumed that *obj* is an object which was returned by an earlier call to `cptr-get`, and that the *cptr* and *type* arguments are the same objects that were used in that call.

The `cptr-out` function performs the "out semantics" encoding action, similar to the treatment

applied to the arguments of a callback prior to returning to foreign code.

9.50.14 Variable `cptr-null`

Description:

The `cptr-null` variable holds a null pointer as a `cptr` instance.

Two `cptr` objects may be compared for equality using the `equal` function, which tests whether their pointers are equal.

The `cptr-null` variable compares `equal` to values which have been subject to `cptr-zap` or `cptr-free`.

A null `cptr` may be produced by the expression `(cptr-obj nil)`; however, this creates a freshly allocated object on each evaluation.

The expression `(cptr-int 0)` also produces a null pointer on all platforms where **TXR** is found.

9.50.15 Function `cptr-size-hint`

Syntax:

```
(cptr-size-hint cptr bytes)
```

Description:

The `cptr-size-hint` function indicates to the garbage collector that the given *`cptr`* object is associated with *`bytes`* of foreign memory that are otherwise invisible to the garbage collector.

Note: this function should be used if the foreign memory is indirectly managed by the *`cptr`* object in cooperation with the garbage collector. Specifically, *`cptr`* should have a finalizer registered against it which will liberate the foreign memory.

9.51 User-Defined Streams

In **TXR Lisp**, stream objects aren't structure types, and therefore lie outside of the object-oriented programming system. However, **TXR Lisp** supports a delegation mechanism which allows a structure which provides certain methods to be used as a stream.

The function `make-struct-delegate-stream` takes as an argument the instance of a structure, which is referred to as the *stream interface object*. The function returns a stream object such that when stream operations are invoked on this stream, it delegates these operations to methods of the stream interface object.

A structure type called `stream-wrap` is provided, whose instances can serve as stream interface objects. This structure has a slot called *`stream`* which holds a stream, and it provides all of the methods required for the delegation mechanism used by `make-struct-delegate-stream`. This `stream-wrap` operations simply invoke the ordinary stream operations on the *`stream`* slot. The `stream-wrap` type can be used as a base class for a derived class which intercepts certain operations on a stream (by defining the corresponding methods) while allowing other operations to transparently pass to the stream (via the base methods inherited from `stream-wrap`).

9.51.1 Function `make-struct-delegate-stream`

Syntax:

```
(make-struct-delegate-stream object)
```

Description:

The `make-struct-delegate-stream` function returns a stream whose operations depend on the *`object`*, a stream interface object.

The *`object`* argument must be a structure which implements certain subsets of, or all of, the following methods: `put-string`, `put-char`, `put-byte`, `get-line`, `get-char`, `get-byte`, `unget-char`, `unget-byte`, `put-buf`, `fill-buf`, `close`, `flush`, `seek`, `truncate`, `get-prop`, `set-prop`, `get-error`, `get-error-str`, `clear-error` and `get-fd`.

Implementing `get-prop` is mandatory, and that method must support the `:name` property.

Failure to implement some of the other methods will impair the use of certain stream operations on the object.

9.51.2 Method `put-string`

Syntax:

```
stream.(put-string str)
```

Description:

The `put-string` method is implemented on a stream interface object. It should behave in a manner consistent with the description of the `put-string` stream I/O function.

9.51.3 Method `put-char`

Syntax:

```
stream.(put-char chr)
```

Description:

The `put-char` method is implemented on a stream interface object. It should behave in a manner consistent with the description of the `put-char` stream I/O function.

9.51.4 Method `put-byte`

Syntax:

```
stream.(put-byte byte)
```

Description:

The `put-byte` method is implemented on a stream interface object. It should behave in a manner consistent with the description of the `put-byte` stream I/O function.

9.51.5 Method `get-line`

Syntax:

```
stream.(get-line)
```

Description:

The `get-line` method is implemented on a stream interface object. It should behave in a manner consistent with the description of the `get-line` stream I/O function.

9.51.6 Method `get-char`

Syntax:

stream.(get-char)

Description:

The `get-char` method is implemented on a stream interface object. It should behave in a manner consistent with the description of the `get-char` stream I/O function.

9.51.7 Method `get-byte`

Syntax:

stream.(get-byte)

Description:

The `get-byte` method is implemented on a stream interface object. It should behave in a manner consistent with the description of the `get-byte` stream I/O function.

9.51.8 Method `unget-char`

Syntax:

stream.(unget-char *chr*)

Description:

The `unget-char` method is implemented on a stream interface object. It should behave in a manner consistent with the description of the `unget-char` stream I/O function.

9.51.9 Method `unget-byte`

Syntax:

stream.(unget-byte *byte*)

Description:

The `unget-byte` method is implemented on a stream interface object. It should behave in a manner consistent with the description of the `unget-byte` stream I/O function.

9.51.10 Method `put-buf`

Syntax:

stream.(put-buf *buf pos*)

Description:

The `put-buf` method is implemented on a stream interface object. It should behave in a manner consistent with the description of the `put-buf` stream I/O function.

Note: there is a severe restriction on the use of the *buf* argument. The buffer object denoted by the *buf* argument may be specially allocated and have a lifetime which is scoped to the method invocation. The `put-buf` method shall not permit the *buf* object to be used beyond the duration

of the method invocation.

9.51.11 Method `fill-buf`

Syntax:

```
stream.(fill-buf buf pos)
```

Description:

The `fill-buf` method is implemented on a stream interface object. It should behave in a manner consistent with the description of the `fill-buf` stream I/O function.

Note: there is a severe restriction on the use of the *buf* argument. The buffer object denoted by the *buf* argument may be specially allocated and have a lifetime which is scoped to the method invocation. The `fill-buf` method shall not permit the *buf* object to be used beyond the duration of the method invocation.

9.51.12 Method `close`

Syntax:

```
stream.(close throw-on-error-p)
```

Description:

The `close` method is implemented on a stream interface object. It should behave in a manner consistent with the description of the `close-stream` stream I/O function.

9.51.13 Method `flush`

Syntax:

```
stream.(flush offs whence)
```

Description:

The `flush` method is implemented on a stream interface object. It should behave in a manner consistent with the description of the `flush-stream` stream I/O function.

9.51.14 Method `seek`

Syntax:

```
stream.(seek offs whence)
```

Description:

The `seek` method is implemented on a stream interface object. It should behave in a manner consistent with the description of the `seek-stream` stream I/O function.

9.51.15 Method `truncate`

Syntax:

```
stream.(truncate len)
```

Description:

The `truncate` method is implemented on a stream interface object. It should behave in a manner consistent with the description of the `truncate-stream` stream I/O function.

9.51.16 Method `get-prop`

Syntax:

```
stream.(get-prop sym)
```

Description:

The `get-prop` method is implemented on a stream interface object. It should behave in a manner consistent with the description of the `get-prop` stream I/O function.

9.51.17 Method `set-prop`

Syntax:

```
stream.(set-prop sym nval)
```

Description:

The `set-prop` method is implemented on a stream interface object. It should behave in a manner consistent with the description of the `set-prop` stream I/O function.

9.51.18 Method `get-error`

Syntax:

```
stream.(get-error)
```

Description:

The `get-error` method is implemented on a stream interface object. It should behave in a manner consistent with the description of the `get-error` stream I/O function.

9.51.19 Method `get-error-str`

Syntax:

```
stream.(get-error-str)
```

Description:

The `get-error-str` method is implemented on a stream interface object. It should behave in a manner consistent with the description of the `get-error-str` stream I/O function.

9.51.20 Method `clear-error`

Syntax:

```
stream.(clear-error)
```

Description:

The `clear-error` method is implemented on a stream interface object. It should behave in a manner consistent with the description of the `clear-error` stream I/O function.

9.51.21 Method `get-fd`

Syntax:

```
stream.(get-fd)
```

Description:

The `get-fd` method is implemented on a stream interface object. It should behave in a manner consistent with the description of the `fileno` stream I/O function.

9.51.22 Structure `stream-wrap`

Syntax:

```
(defstruct stream-wrap nil
  stream
  (:method put-string (me str)
    (put-string str me.stream))
  (:method put-char (me chr)
    (put-char chr me.stream))
  (:method put-byte (me byte)
    (put-byte byte me.stream))
  (:method get-line (me)
    (get-line me.stream))
  (:method get-char (me)
    (get-char me.stream))
  (:method get-byte (me)
    (get-byte me.stream))
  (:method unget-char (me chr)
    (unget-char chr me.stream))
  (:method unget-byte (me byte)
    (unget-byte byte me.stream))
  (:method put-buf (me buf pos)
    (put-buf buf pos me.stream))
  (:method fill-buf (me buf pos)
    (fill-buf buf pos me.stream))
  (:method close (me throw-on-error)
    (close-stream me.stream throw-on-error))
  (:method flush (me)
    (flush-stream me.stream))
  (:method seek (me offs whence)
    (seek-stream me.stream offs whence))
  (:method truncate (me len)
    (truncate-stream me.stream len))
  (:method get-prop (me sym)
    (stream-get-prop me.stream sym))
  (:method set-prop (me sym nval)
    (stream-set-prop me.stream sym nval))
  (:method get-error (me)
    (get-error me.stream))
  (:method get-error-str (me)
    (get-error-str me.stream))
  (:method clear-error (me)
    (clear-error me.stream))
  (:method get-fd (me)
    (fileno me.stream)))
```

Description:

The `stream-wrap` class provides a trivial implementation of a stream interface. It has a single slot, `stream` which should be initialized with a stream object. Each methods of *`stream-wrap`*, shown in its entirety in the above Syntax section, simply invoke the corresponding stream I/O library functions, passing the method arguments, and the value of the `stream` slot to that function, and consequently returning whatever that function returns.

Note: the `stream-wrap` function is intended to useful as an inheritance base. A user-defined structure can inherit from `stream-wrap` and provide its own versions of some of the methods,

thereby intercepting those operations to customize the behavior.

For instance, a function equivalent to the `record-adapter` function could be implemented by constructing an object derived from `stream-wrap` which overrides the behavior of the `get-line` method, and then using the `make-struct-delegate-stream` to return a stream based on this object.

Example:

```
;;; Implementation of my-record-adapter,
;;; a function resembling
;;; the record-adapter implementation

(defstruct rec-input stream-wrap
  regex
  include-match-p

  ;; get-line overridden to use regex-based
  ;; extraction using read-until-match
  (:method get-line (me)
    (read-until-match me.regex me.stream
                      me.include-match-p))))

(defun my-record-adapter (regex stream include-match-p)
  (let ((recin (new rec-input
                    stream stream
                    regex regex
                    include-match-p include-match-p)))
   (make-struct-delegate-stream recin)))
```

9.52 Symbols and Packages

TXR Lisp has a package system inspired by the salient features of ANSI Common Lisp, but substantially simpler.

Each symbol has a name, which is a string.

A package is an object which serves as a container of symbols; the package associates the name strings with symbols.

A symbol which exists inside a package is said to be interned in that package. A symbol can be interned in more than one package.

A symbol may also have a home package. A symbol which has a home package is always interned in that package.

A symbol which has a home package is called an *interned symbol*.

A symbol which is interned in one or more packages, but has no home package, is a *quasi-interned symbol*. When a quasi-interned symbol is printed, if it is not interned in the package currently held in the `*package*` variable, it will appear in uninterned notation denoted by a `#:` prefix, even though it is interned in one or more packages. This is because in any situation when a symbol is printed with a package prefix, that prefix corresponds to the name of its home package. The reverse isn't true: when a symbol token is read bearing a package prefix, the token denotes any interned symbol in the indicated package, whether or not the package is the home package of that symbol.

Packages are held in a global list which can be used to search for a package by name. The `find-package` function performs this lookup. A package may be deleted from the list with the `delete-package` function, but it continues to exist until the program loses the last reference to that package. When a package is deleted with `delete-package`, its symbols are uninterned from all other packages.

An existing symbol can be brought into a package via the `use-sym` function, causing it to be interned in that package. A symbol which thus exists inside a package which is not its home package is called a *foreign symbol*, relative to that package. The contrasting term with *foreign symbol* is *local symbol*, which refers to a symbol, relative to a package, which is interned in that package and that package is also its home. Every symbol interned in a package is either foreign or local.

If a foreign symbol is introduced into a package, and has the same name as an existing local symbol, the local symbol continues to exist, but is hidden: it is not accessible via a name lookup on that package. While hidden, a symbol loses its home package and is thus degraded to either quasi-interned or uninterned status, depending on whether that symbol is interned in other packages.

When a foreign symbol is removed from a package via `unuse-sym`, then if a hidden symbol exists in that package of the same name, that hidden symbol is reinterned in that package and reacquires that package as its home package, becoming an interned symbol again.

Finally, packages have a *fallback package list*: a list of associated packages, which may be empty. The fallback package list is manipulated with the functions `package-fallback-list` and `set-package-fallback-list`, and with the `:fallback` clause of the `defpackage` macro. The fallback package list plays a role only in three situations: one in the **TXR Lisp** parser, one in the printer, and one in the interactive listener. Besides that, two library functions refer to it: `intern-fb` and `find-symbol-fb`.

The parser situation involving the fallback list occurs when the **TXR Lisp** parser resolves an unqualified symbol token: a symbol token not carrying a package prefix. Such a symbol name is resolved against the current package (the package currently stored in the `*package*` special variable). If a symbol matching the token is not found in the current package, then the packages in its fallback package list are searched for the symbol. The first matching symbol which is found in the fallback list is returned. If no matching symbol is found in the fallback list, then the token is interned as a new symbol is interned in the current package. The packages in the current package's fallback list may themselves have fallback lists. Those fallback lists are not involved; no such recursion takes place.

The printer situation involving the fallback list is as follows. If a symbol is being printed in a machine-readable way (not "pretty"), has a home package and is not a keyword symbol, then a search takes place through the current package first and then its fallback list. If the symbol is found anywhere in that sequence of locations, and is not occluded by a same-named symbol occurring earlier in that sequence, then the symbol is printed without a package prefix.

The listener situation involving the fallback list is a follows. When tab completion is used on a symbol without a package prefix, the listener searches for completions not only in the current package, but in the fallback list also.

Dialect Notes:

> The **TXR Lisp** package system doesn't support the ANSI Common Lisp concept of package use, replacing that concept with fallback packages.
>
> Though the `use-package` and `unuse-package` functions exist and are similar to the ones in ANSI CL, they actually operate on individual foreign symbols, bringing them in or removing them, respectively. These functions effectively iterate over the local symbols of the used or unused package, and invoke `use-sym` or `unuse-sym`, respectively.

The **TXR Lisp** package system consequently doesn't support the concept of shadowing symbols, and conflicts do not exist. When a foreign symbol is introduced into a package which already has a symbol by that name, that symbol is silently removed from that package if it is itself foreign, or else hidden if it is local.

The **TXR Lisp** package system also doesn't feature the concept of internal and external symbols. The rationale is that this distinction divides symbols into subsets in a redundant way. Packages are already subsets of symbols. A module can use two packages to simulate private symbols. An example of this is given in the Package Examples section below.

The **TXR Lisp** fallback package list mechanism resembles ANSI CL package use, and satisfies similar use scenarios. However, this mechanism does not cause a symbol to be considered visible in a package. If a package `foo` contains no symbol `bar`, but one of the packages in `foo`'s fallback list does contain `bar`, that symbol is nevertheless not considered visible in `foo`. The syntax `foo:bar` will not resolve. The fallback mechanism only comes into play when a package is installed as the current package in the `*package*` variable. It then allows unqualified symbol references to refer across the fallback list.

9.52.1 Package Examples

The following example illustrates a simple scenario of a module whose identifies are in a package, and which also has private identifiers in a private package.

```
;; Define three packages.
(defpackage mod-priv
  (:fallback usr))

(defpackage mod)

(defpackage client
  (:fallback mod usr)
  (:use-from mod-priv other-priv))

;; Switch to mod-priv package
(in-package mod-priv)

(defun priv-fun (arg)
  (list arg))

;; Another function with a name in the mod-priv package.
(defun other-priv (arg)
  (cons arg arg))

;; Define a function in mod; a public function.
;; Note that we don't have to change to the mod package,
;; to define functions with names in that package.
;; We rely on interning being allowed for the qualified
;; mod:public-fun syntax.
(defun mod:public-fun (arg)
  (priv-fun arg)) ;; priv-fun here is mod-priv:priv-fun

;; Switch to client package
(in-package client)

(priv-fun) ;; ERROR: refers to client:priv-fun, not defined
```

```
(mod:priv-fun) ;; ERROR: mod-priv:priv-fun not used in mod
(mod-priv:priv-fun 3) ;; OK: direct reference via qualifier
(public-fun 3) ;; OK: mod:public-fun symbol via fallback
(other-priv 3) ;; OK: foreign symbol mod-priv:other-priv
               ;; present in client due to :use-from
```

The following example shows how to create a package called `custom` in which the + symbol from the `usr` package is replaced with a local symbol. A function is then defined using the local symbol, which allows strings to be catenated with +:

```
(defpackage custom
  (:fallback usr)
  (:local + - * /))

(defmacro outside-macro (x) ^(+ ,x 42))

(in-package custom)

(defun binary-+ (: (left 0) (right 0))
  (if (and (numberp left) (numberp right))
    (usr:+ left right)
    `@left@right`))

(defun + (. args)
  [reduce-left binary-+ args])

(+) -> 0
(+ 1) -> 1
(+ 1 "a") -> "1a"
(+ 1 2) -> 3
(+ "a") -> "a"
(+ "a" "b" "c") -> "abc"

;; macro expansions using usr:+ are not affected
(outside-macro "a") -> ;; error: + invalid operands "a" 42
```

9.52.2 Packages and the Extraction Language

The **TXR** extraction language has a syntax in which certain Lisp symbolic expressions denoting directives `@(collect ...)` or `@(end)` behave as if they were the tokens of a phrase structure. As a matter of implementation, these are processed specially in the parser and lexical analyzer, and are not read in the same way as ordinary Lisp forms.

On the other hand, some directives are not this way. For instance the `@(bind ...)`, syntax is processed as a true Lisp expression, in which the `bind` token is subject to the usual rules for interning a symbol, sensitive to `*package*` in the usual way.

The following notes describe the treatment of "special" directives that are involved in phrase structure syntax. It applies to all directives which head off a block that must be terminated by `@(end)`, all "punctuation" directives like `@(and)` or `@(end)` and all subphrase indicators like `@(last)` or `@(elif)`.

Firstly, each such directive may have a package prefix on its main symbol, yet is still recognized as the same token. That is to say, `@(foo:collect)` is still treated by the tokenizer and parser as the `@(collect)` token, regardless of the package prefix, and regardless of whether `foo:end` is the same symbol as the `usr:end` symbol.

However, this doesn't mean that any `foo:collect` is allowed to denote the `collect` directive.

A qualified symbol such as `foo:collect` must correspond to (be the same object as) precisely one of two symbols: either the same-named symbol in the `usr` package, or else the same-named symbol in the `keyword` package. If this condition isn't satisfied, the situation is a syntax error. Note that this check uses the original `usr` and `keyword` packages, not the packages which are currently named `"usr"` or `"keyword"` in the current `*package-alist*`.

A check is also performed for an unqualified symbol. An unqualified symbol like `collect` must also resolve, in the context of the current value of the `*package*` variable, to the same named-symbol in either the original `usr` or `keyword` package. Thus if the current package isn't `usr`, and `@(collect)` is being processed, the current package must be such that `collect` resolves to `usr:collect`. either because that symbol is present in the current pack via import, or else visible via the fallback list.

These rules are designed to approximate what the behavior would be if these directives were actually scanned as Lisp forms in the usual way and then recognized as phrase structure tokens according to the identity of their leading symbol. The additional restriction is added that that the directive symbol names are treated as reserved. If there exists a user-defined pattern function called `mypackage:end` it may not be invoked using the syntax `@(mypackage:end)`, which is erroneous; though it is invocable indirectly via the `@(call)` directive.

9.52.3 Package Library Conventions

Various functions in the package and symbol area of the library have a *`package`* parameter. When the argument is optional, it defaults to the current value of the `*package*` special variable.

If specified, the argument may be a character string, which is taken as the name of a package. It may also be a symbol, in which case the symbol's name, which is a character string, is used. Thus the objects `:sys`, `usr:sys`, `abc:sys` and `"sys"` all refer to the same package, the system package which is named `"sys"`.

A `package` parameter may also simply be a package object.

Some functions, like `use-package` and `unuse-package` functions accept a list of packages as their first argument. This may be a list of objects which follow the above conventions: strings, symbols or package objects. Also, instead of a list, an atom may be passed: a string, symbol or package object. It is treated as a singleton list consisting of that object.

9.52.4 Variables `user-package`, `keyword-package` and `system-package`

Description:

> These variables hold predefined packages. The `user-package` contains all of the public symbols in the **TXR Lisp** library. The `keyword-package` holds keyword symbols, which are printed with a leading colon. The `system-package` is for internal symbols, helping the implementation avoid name clashes with user code in some situations.
>
> These variables shouldn't be modified. If they are modified, the consequences are unspecified.
>
> The names of these packages, respectively, are `"usr"`, `"sys"`, and `"keyword"`.

9.52.5 Special variable `*package*`

Description:

This variable holds the current package. The global value of this variable is initialized to a package called `"pub"`. The `pub` package has the `usr` package in its fallback list; thus when `pub` is current, all of the `usr` symbols, comprising the content of the **TXR Lisp** library, are visible.

All forms read and evaluated from the **TXR** command line, in the interactive listener, from files via `load` or `compile-file` or from the **TXR** pattern language are processed in this default `pub` package, unless arrangement are made to change to a different package.

The current package is used as the default package for interning symbol tokens which do not carry the colon-delimited package prefix.

The current package also affects printing. When a symbol is printed whose home package matches the current package, it is printed without a package prefix. (Keyword symbols are always printed with the colon prefix, even if the keyword package is current.)

9.52.6 Function `make-sym`

Syntax:

```
(make-sym name)
```

Description:

The `make-sym` function creates and returns a new symbol object. The argument *`name`*, which must be a string, specifies the name of the symbol. The symbol does not belong to any package (it is said to be "uninterned").

Note: an uninterned symbol can be interned into a package with the `rehome-sym` function. Also see the `intern` function.

9.52.7 Function `gensym`

Syntax:

```
(gensym [prefix])
```

Description:

The `gensym` function is similar to `make-sym`. It creates and returns a new symbol object. If the *`prefix`* argument is omitted, it defaults to `"g"`.

The difference between `gensym` and `make-sym` is that `gensym` creates the symbol's name by combining the *`prefix`* with a numeric suffix. The suffix is obtained by incrementing the `*gensym-counter*` and taking the new value. The name string then calculated from the prefix and the counter value as if by evaluating a form similar to `(fmt "~a~,04d" prefix counter)`. From this it can be inferred that *`prefix`* can be an object of any kind.

Note: the generated symbol's name, though varying thanks to the incrementing counter, is not the basis of its uniqueness. The basis of the symbol's uniqueness is that it is a freshly created object, distinct from any other object. The related function `make-sym` still returns unique symbols even if repeatedly called with the same string argument.

9.52.8 Special variable `*gensym-counter*`

Description:

This variable is initialized to 0. Each time the `gensym` function is called, it is incremented. The incremented value forms the basis of the numeric suffix which `gensym` uses to form the name of the new symbol.

9.52.9 Function `make-package`

Syntax:

```
(make-package name [weak])
```

Description:

The `make-package` function creates and returns a package named *name*, where *name* is a string. It is an error if a package by that name exists already. Note: ordinary creation of packages for everyday program modularization should be performed with the `defpackage` macro rather than by direct use of `make-package`.

If the *weak* parameter is given an argument which is a Boolean true, then the resulting package holds symbols weakly, from a garbage collection point of view. If the only reference to a symbol is that which occurs inside the weak package, then that symbol may be removed from the package and reclaimed by the garbage collector.

Note: weak packages address the following problem. The application creates a package for the purpose of reading Lisp data. Symbols occurring in that data therefore are interned into the package. Subsequently, the application retains references to some of the symbols, discarding the others. If the package isn't weak, then because the application is retaining some of the symbols, and those symbols hold a reference to the package, and the package holds a reference to all symbols that were interned in it, all of the symbols are retained. If a weak package is used, then the uninterested symbols are eligible for garbage collection.

9.52.10 Function `delete-package`

Syntax:

```
(delete-package package)
```

Description:

The `delete-package` breaks the association between a package and its name. After `delete-package`, the *package* object continues to exist, but cannot be found using `find-package`.

Furthermore, `delete-package` iterates over all remaining packages. For each remaining package *p*, it performs the semantic action of the `(unuse-package package p)` expression. That is to say, all of the remaining packages are scrubbed of any foreign symbols which are the local symbols of the deleted *package*.

9.52.11 Function `merge-delete-package`

Syntax:

```
(merge-delete-package dst-package [src-package])
```

Description:

The `merge-delete-package` iterates over all of the local symbols of *src-package* and rehomes each symbol into *dst-package*. Then, it deletes *src-package*.

Note: the local symbols are identified as if using `package-local-symbols`, rehoming is performed as if using `rehome-sym`, and deleting *`src-package`* is performed as if using `delete-package`.

9.52.12 Function `packagep`

Syntax:

```
(packagep obj)
```

Description:

The `packagep` function returns `t` if *`obj`* is a package, otherwise it returns `nil`.

9.52.13 Function `find-package`

Syntax:

```
(find-package name)
```

Description:

The argument *`name`* should be a string. If a package called *`name`* exists, then it is returned. Otherwise `nil` is returned.

9.52.14 Special variable `*package-alist*`

Description:

The `*package-alist*` variable contains the master association list which contains an entry about each existing package.

Each element of the list is a cons cell whose `car` field is the name of a package and whose `cdr` is a package object.

Note: the **TXR Lisp** application can overwrite or rebind this variable to manipulate the active package list. This is useful for *sandboxing*: safely evaluating code that is obtained as an input from an untrusted source, or calculated from such an input.

The contents of `*package-alist*` have security implications because textual source code can refer to any symbol in any package by invoking a package prefix. For instance, even if the `open` function's name is not available in the current package (established by the `*package*` variable) that symbol can easily be obtained using the syntax `usr:open`.

However, the entire `usr` package itself can be removed from `*package-alist*`. In that situation, the syntax `usr:open` is no longer valid.

At the same time, selected symbols from the original `usr` can be nevertheless made available via some intermediate package, which is present in `*package-alist*` and which contains a subset of the `usr` symbols that has been curated for safety. That curated package may even be called `usr`, so that if for instance `cons` is present in that package, it may be referred to as `usr:cons` in the usual way.

9.52.15 Function `package-alist`

Syntax:

```
(package-alist)
```

Description:

The `package-alist` function retrieves the value of `*package-alist*`.

Note: this function is obsolescent. There is no reason to use it in new code instead of just accessing `*package-alist*` directly.

9.52.16 Function `package-name`

Syntax:

```
(package-name package)
```

Description:

The `package-name` function retrieves the name of a package.

9.52.17 Function `package-symbols`

Syntax:

```
(package-symbols [package])
```

Description:

The `package-symbols` function returns a list of all the symbols which are interned in *package*.

9.52.18 Functions `package-local-symbols` and `package-foreign-symbols`

Syntax:

```
(package-local-symbols [package])
(package-foreign-symbols [package])
```

Description:

The `package-local-symbols` function returns a list of all the symbols which are interned in *package*, and whose home package is that package.

The `package-foreign-symbols` function returns a list of all the symbols which are interned in *package*, which do not have that package as their home package, or do not have a home package at all.

The union of the local and foreign symbols contains exactly the same elements as the list returned by `package-symbols`: the symbols interned in a package are partitioned into local and foreign.

9.52.19 Functions `package-fallback-list` and `set-package-fallback-list`

Syntax:

```
(package-fallback-list package)
(set-package-fallback-list package package-list)
```

Description:

The `package-fallback-list` returns the current *fallback package list* associated with *package*.

The `set-package-fallback-list` replaces the fallback package list of *package* with *package-list*.

The *package-list* argument must be a list which is a mixture of symbols, strings or package objects. Strings are taken to be package names, which must resolve to existing packages. Symbols are reduced to strings via `symbol-name`.

9.52.20 Functions `intern` and `intern-fb`

Syntax:

```
(intern name [package])
(intern-fb name [package])
```

Description:

The argument *name* must be a string. The optional argument *package* must be a package. If *package* is not supplied, then the value taken is that of `*package*`.

The `intern` function searches *package* for a symbol called *name*. If that symbol is found, it is returned. If that symbol is not found, then a new symbol called *name* is created and inserted into *package*, and that symbol is returned. In this case, the package becomes the symbol's home package.

The `intern-fb` function is very similar to `intern` except that if the symbol is not found in *package* then the packages listed in the fallback list of *package* are searched, in order. Only these packages themselves are searched, not their own fallback lists. If a symbol called *name* is found, the search terminates and that symbol is returned. Only if nothing is found in the fallback list will `intern-fb` create a new symbol and insert it into *package*, exactly like `intern`.

9.52.21 Function `unintern`

Syntax:

```
(unintern symbol [package])
```

Description:

The `unintern` function removes *symbol* from *package*.

The `nil` symbol may not be removed from the `usr` package; an error exception is thrown in this case.

If `symbol` isn't `nil`, then *package* is searched to determine whether it contains *symbol* as an interned symbol (either local or foreign), or a hidden symbol.

If *symbol* is a hidden symbol, then it is removed from the hidden symbol store. Thereafter, even if a same-named foreign symbol is removed from the package via `unuse-sym` or `unuse-package`, those operations will no longer restore the hidden symbol to interned status. In this case, *unintern* returns the hidden symbol that was removed from the hidden store.

If *symbol* is a foreign symbol, then it is removed from the package. If the package has a hidden symbol of the same name, that hidden symbol is reinterned in the package, and the package once again becomes its home package. In this case, *symbol* is returned.

If *symbol* is a local symbol, the symbol is removed from the package. In this case also, *symbol* is returned.

If *symbol* is not found in the package as either an interned or hidden symbol, then the function has no effect and returns `nil`.

9.52.22 Functions `find-symbol` and `find-symbol-fb`

Syntax:

```
(find-symbol name [package [notfound-val]])
(find-symbol-fb name [package [notfound-val]])
```

Description:

The `find-symbol` and `find-symbol-fb` functions search *package* for a symbol called *name*. That argument must be a character string.

If the *package* argument is omitted, the parameter defaults to the current value of `*package*`.

If the symbol is found in *package* then it is returned.

If the symbol is not found in *package*, then the function `find-symbol-fb` also searches the packages listed in the fallback list of *package* are searched, in order. Only these packages themselves are searched, not their own fallback lists. If a symbol called *name* is found, the search terminates and that symbol is returned.

The function `find-symbol` only searches *package*, ignoring its fallback list.

If a symbol called *name* isn't found, then these functions return *notfound-val* is returned, which defaults to `nil`.

Note: an ambiguous situation exists when *notfound-val* is a symbol, such as its default value `nil`, because if that symbol is successfully found, it is indistinguishable from *notfound-val*.

9.52.23 Function `rehome-sym`

Syntax:

```
(rehome-sym symbol [package])
```

Description:

The arguments *symbol* and *package* must be a symbol and package object, respectively, and *symbol* must not be the symbol `nil`.

The `rehome-sym` function moves *symbol* into *package*. If *symbol* is already interned in a package, it is first removed from that package.

If a symbol of the same name exists in *package*, that symbol is first removed from *package*.

Also, if a symbol of the same name exists in the hidden symbol store of *package*, that hidden symbol is removed.

Then `symbol` is interned into *package*, and *package* becomes its home package, making it a local symbol of *package*.

Note: if `symbol` is currently the hidden symbol of some package, it is not removed from the hidden symbol store of that package. This is a degenerate case. The implication is that if that hidden symbol is ever restored in that package, it will once again have that package as its home package, and consequently it will turn into a foreign symbol of *package*.

9.52.24 Function `symbolp`

Syntax:

```
(symbolp obj)
```

Description:

The `symbolp` function returns `t` if *`obj`* is a symbol, otherwise it returns `nil`.

9.52.25 Function `symbol-name`

Syntax:

```
(symbol-name symbol)
```

Description:

The `symbol-name` function returns the name of *`symbol`*.

9.52.26 Function `symbol-package`

Syntax:

```
(symbol-package symbol)
```

Description:

The `symbol-package` function returns the home package of *`symbol`*. If *`symbol`* has no home package, it returns `nil`.

9.52.27 Function `keywordp`

Syntax:

```
(keywordp obj)
```

Description:

The `keywordp` function returns `t` if *`obj`* is a keyword symbol, otherwise it returns `nil`.

9.52.28 Function `bindable`

Syntax:

```
(bindable obj)
```

Description:

The `bindable` function returns `t` if *`obj`* is a bindable symbol, otherwise it returns `nil`.

All symbols are bindable, except for keyword symbols, and the special symbols `t` and `nil`.

9.52.29 Function `use-sym`

Syntax:

```
(use-sym symbol [package])
```

Description:

The `use-sym` function brings an existing `symbol` into *`package`*.

In all cases, the function returns `symbol`.

If *symbol* is already interned in *package*, then the function has no effect.

Otherwise *symbol* is interned in *package*.

If a symbol having the same name as *symbol* already exists in *package*, then it is replaced. If that replaced symbol is a local symbol of *package*, then the replaced symbol turns into a hidden symbol associated with the package. It is placed into a special hidden symbol store associated with *package* and is stripped of its home package, becoming quasi-interned or uninterned.

An odd case is possible whereby *symbol* is already a hidden symbol of *package*. In this case, the hidden symbol replaces some foreign symbol and is interned in *package*. Thus it simultaneously exists as both an interned foreign symbol and as a hidden symbol of *package*.

9.52.30 Function `unuse-sym`

Syntax:

```
(unuse-sym symbol [package])
```

Description:

The `unuse-sym` function removes *symbol* from *package*.

If *symbol* is not interned in *package*, the function does nothing and returns `nil`.

If *symbol* is a local symbol of *package*, an error is thrown: a package cannot "unuse" its own symbol. Removing a symbol from its own home package requires the `unintern` function.

Otherwise *symbol* is a foreign symbol interned in *package* and is removed.

If the package has a hidden symbol of the same name as *symbol*, that symbol is reinterned into *package* as a local symbol. In this case, that previously hidden symbol is returned.

If the package has no hidden symbol matching the removed *symbol*, then *symbol* itself is returned.

9.52.31 Functions `use-package` and `unuse-package`

Syntax:

```
(use-package package-list [package])
(unuse-package package-list [package])
```

Description:

The *use-package* and *unuse-package* are convenience functions which perform a mass import of symbols from one package to another, or a mass removal, respectively.

The `use-package` function iterates over all of the local symbols of the packages in *package-list*. For each symbol *s*, it performs the semantic action implied by the `(use-sym s package)` expression.

Similarly `unuse-package` iterates *package-list* in the same way, performing, effectively, the semantic action of the `(unuse-sym s package)` expression.

The *package-list* argument must be a list which is a mixture of symbols, strings or package objects. Strings are taken to be package names, which must resolve to existing packages. Symbols are reduced to strings via `symbol-name`.

9.52.32 Macro `defpackage`

Syntax:

```
(defpackage name clause*)
```

Description:

The `defpackage` macro provides a convenient means to create a package and establish its properties in a single construct. It is intended for the ordinary situations in which packages support the organization of programs into modules.

The `name` argument, giving the package name, may be a symbol or a character string. If it is a symbol, then the symbol's name is taken to be name for the package.

If a package called `name` already exists, then `defpackage` selects that package for further operations. Otherwise, a new, empty package is created. In either case, this package is referred to as the *present package* in the following descriptions.

The `name` may be optionally followed by one or more clauses, which are processed in the order that they appear. Each clause is a compound form headed by a keyword. The supported clauses are as follows:

`(:fallback package-name*)`
: The `:fallback` clause specifies the packages to comprise the fallback list of the present package. If this clause is omitted, or if it is present with no *package-name* arguments, then the present package has an empty fallback list. Each *package-name* may be a string or symbol naming an existing package. It is permitted for the present package itself to appear in its own fallback list. This is useful for creating a package with a nonempty fallback list which doesn't actually provide access to any other package.

`(:use package-name*)`
: The `:use` clause specifies packages whose local symbols are to be interned into the present package as foreign symbols. Each *package-name* may be a string or symbol naming an existing package. The list of package names is processed as if by a call to `use-package`.

`(:use-syms symbol*)`
: The `:use-syms` clause specifies individual symbols to be interned in the present package. The arguments are symbols.

`(:use-from package-name symbol-name*)`
: The `:use-from` clause specifies the names of local symbols in a package denoted by *package-name* to be used in the present package. All arguments of `:use-from` are either strings or symbols which are reduced to strings by mapping to their names. Each *symbol-name* is interned in the package identified by *package-name*, which may have the effect of creating that symbol. This symbol is expected to be a local symbol of that package. If that is so, the symbol is brought into the present package via `use-sym`. Otherwise if the symbol is foreign to package identified by *package-name*, then an error exception is thrown.

`(:local symbol-name*)`
: The `:local` clause specifies the names of symbols to be interned in the new package as local symbols. Each *symbol-name* argument must be either a character string or a symbol. If it is a symbol, its name is taken, thereby reducing the argument to a character string. The arguments are processed in the order in which they appear. Each name is first interned in the newly created package using the `intern` function. Then, if the resulting symbol is foreign to the package, it is removed with `unuse-sym` and the name is interned again.

9.52.33 Macro `in-package`

Syntax:

```
(in-package name)
```

Description:

The `in-package` macro causes the `*package*` special variable to take on the package denoted by *`name`*. The macro checks, at expansion time, that *`name`* is either a string or symbol. An error is thrown if this isn't the case.

The *`name`* argument expression isn't evaluated, and so must not be quoted.

The code generated by the macro performs a search for the package. If the package is not found at the time when the macro's expansion is evaluated, an error is thrown.

9.53 Pseudorandom Numbers

9.53.1 Special variable `*random-state*`

Description:

The `*random-state*` variable holds an object which encapsulates the state of a pseudorandom number generator. This variable is the default argument value for the `random-fixnum` and `random functions`, for the convenience of writing programs which are not concerned about the management of random state.

On the other hand, programs can create and manage random states, making it possible to obtain repeatable sequences of pseudorandom numbers which do not interfere with each other. For instance objects or modules in a program can have their own independent streams of random numbers which are repeatable, independently of other modules making calls to the random number functions.

When **TXR** starts up, the `*random-state*` variable is initialized with a newly created random state object, which is produced as if by the call `(make-random-state 42)`.

9.53.2 Special variable `*random-warmup*`

Description:

The `*random-warmup*` special variable specifies the value which is used by `make-random-state` in place of a missing *`warmup-period`* argument.

To "warm up" a pseudorandom number generator (PRNG) means to obtain some values from it which are discarded, prior to use. The number of values discarded is the *warm-up period*.

The WELL512a PRNG used in **TXR** produces 32-bit values, natively. Thus each warm-up iteration retrieves and discards a 32-bit value. The PRNG has a state space consisting of a vector of sixteen 32-bit words, making the state space 4096 bits wide.

Warm up is required because PRNG-s, in particular PRNG-s with large state spaces and long periods, produce fairly predictable sequences of values in the beginning, before transitioning into chaotic behavior. This problem is worse for low complexity seeds, such as small integer values.

The sequences are predictable in two ways. Firstly, some initial values extracted from the PRNG may exhibit patterns ("problem 1"). Secondly, the initial values from sequences produced from similar seeds (for instance consecutive integers) may be similar or identical ("problem 2").

Notes:

The default value of `*random-warmup*` is only 8. This is insufficient to ensure good initial PRNG behavior for seeds even as large as 64 bits or more. That is to say, even if as many as eight bytes' worth of true random bits are used as the seed, the PRNG will exhibit predictable behaviors, and a poor distribution of values.

Applications which critically depend on good PRNG behavior should choose large warm-up periods into the hundreds or thousands of iterations. If a small warm-up period is used, it is recommended to use larger seeds which initialize more of the 4096-bit state space.

TXR's PRNG implementation addresses "problem 1" by padding the unseeded portions of the state space with random values (from a static table that doesn't change). For instance, if the integer 1 is used to seed the space, then one 32-bit word of the space is set to the value 1. The remaining 15 are populated from the random table. This helps to ensure that a good PRNG sequence is obtained immediately. However, it doesn't address "problem 2": that similar seed values generate similar sequences, when the warm-up period is small. For instance, if 65536 different random state objects are created, from each of the 16-bit seeds in the range [0, 65536), and then a random 16-bit value is extracted from each state, only 1024 unique values result.

9.53.3 Function `make-random-state`

Syntax:

```
(make-random-state [seed [warmup-period])
```

Description:

The `make-random-state` function creates and returns a new random state, an object of the same kind as what is stored in the `*random-state*` variable.

The seed, if specified, must be an integer value, a buffer, an existing random state object, or else a vector returned from a call to the function `random-state-get-vec`.

Note that the sign of the seed is ignored, so that negative seed values are equivalent to their additive inverses.

If seed is not specified, then `make-random-state` produces a seed based on some information in the process environment, such as current time of day. It is not guaranteed that two calls to `make-random-state` that are separated by less than some minimum increment of real time produce different seeds. The minimum time increment depends on the platform.

On a platform with a millisecond-resolution real-time clock, the minimum time increment is a millisecond. Calls to `make-random-state` less than a millisecond apart may predictably produce the same seed.

If an integer or buffer seed is specified, then the integer value is mapped to a pseudorandom sequence, in a platform-independent way.

If an existing random state is specified as a seed, then it is duplicated. The returned random state object is a distinct object which is in the same state as the input object. It will produce the same remaining pseudorandom number sequence, as will the input object.

If a vector is specified as a seed, then a random state is constructed which duplicates the random state object which was captured in that vector representation by the `random-state-get-vec` function.

The *warm-up-period* argument specifies the number of values which are immediately obtained and discarded from the newly-seeded generator before it is returned. This procedure is referred to as PRNG *warm-up*.

Warm-up is not performed if *seed* is a vector or random state object. In this situation, if the *warm-up-period* is present, it may still be required to be an integer, even though it is ignored.

If warm-up is performed, but the *warm-up-period* argument is missing, then the value of the `*random-warmup*` special variable is used. Note: this variable has a default value which may be too small for some applications of pseudorandom numbers; see the Notes under `*random-warmup*`.

9.53.4 Function `random-state-p`

Syntax:

```
(random-state-p obj)
```

Description:

The `random-state-p` function returns `t` if *obj* is a random state, otherwise it returns `nil`.

9.53.5 Function `random-state-get-vec`

Syntax:

```
(random-state-get-vec [random-state])
```

Description:

The `random-state-get-vec` function converts a random state into a vector of integer values. If the *random-state* argument, which must be a random state object, is omitted, then the value of the `*random-state*` is used.

9.53.6 Functions `random-fixnum`, `random` and `rand`

Syntax:

```
(random-fixnum [random-state])
(random random-state modulus)
(rand modulus [random-state])
```

Description:

All three functions produce pseudorandom numbers, which are positive integers.

The numbers are obtained from a WELL512a PRNG, whose state is stored in the random state object.

The `random-fixnum` function produces a random fixnum integer: a reduced range integer which fits into a value that does not have to be heap-allocated.

The `random` and `rand` functions produce a value in the range [0, *modulus*). They differ only in the order of arguments. In the `rand` function, the random state object is the second argument and is optional. If it is omitted, the global `*random-state*` object is used.

The *modulus* argument must be a positive integer. If *modulus* is 1, then the function returns zero without altering the state of the pseudorandom number generator.

9.53.7 Functions `random-float` and `random-float-incl`

Syntax:

```
(random-float [random-state])
(random-float-incl [random-state])l
```

Description:

The `random-float` function produces a pseudorandom floating-point value in the range [0.0, 1.0).

The `random-float-incl` produces a pseudorandom floating-point value in the range [0.0, 1.0], thus differing from `random-float` by including the 1.0 limit value.

The numbers are obtained from a WELL512a PRNG, whose state is stored in the random state object given by the argument to the optional *`random-state`* parameter, which defaults to the value of `*random-state*`.

Because the floating-point type does not provide a representation of every real value in the range 0.0 to 1.0, it is not possible to impose the requirement that every value shall occur with equal likelihood.

Rather, these functions are intended to produce an a uniform distribution of values according to the following pragmatic requirements. A subset S of the real values in the specified range, [0.0, 1.0) or [0.0, 1.0] is identified whose elements are representable in the floating-point type and which are uniformly spaced along the interval. Then, a random element is chosen from S and returned, such that every element is equally likely to be selected.

Note that these requirements do not correspond to the more mathematically ideal concept of uniformly choosing actual real numbers in the [0, 1] interval of the real number line, and then finding the closest floating-point representation. Such a requirement would mean that the boundary values 0.0 and 1.0 appear in the output half as frequently as all the interior values, because each of these two floating-point values is a representations of a range of numbers, half of which lies outside of the [0, 1] interval.

9.53.8 Function `random-buf`

Syntax:

```
(random-buf size [random-state])
```

Description:

The `random-buf` function creates a `buf` object of the specified *`size`* fills it with pseudorandom bytes, and returns it.

The bytes are obtained from the random state object given by the optional *`random-state`* parameter, which defaults to the value of `*random-state*`.

See the section **Buffers** for a description of `buf` objects.

9.53.9 Function `random-sample`

Syntax:

```
(random-sample size seq [random-state])
```

Description:

The `random-sample` function returns a vector of *`size`* randomly selected elements from the sequence *`seq`*, using reservoir sampling.

If the number of elements in *`seq`* is equal to or smaller than *`size`*, then the function returns a vector of all the elements of *`seq`* in their original order.

In other cases, the selected elements are not required to appear in their original order.

No element of sequence *`seq`* is selected more than once; duplicate values can appear in the output only if *`seq`* itself contains duplicates.

9.54 Time

9.54.1 Functions `time`, `time-usec` and `time-nsec`

Syntax:

```
(time)
(time-usec)
(time-nsec)
```

Description:

The `time` function returns the number of seconds that have elapsed since midnight, January 1, 1970, in the UTC timezone: a point in time called *the epoch*.

The `time-usec` function returns a cons cell whose `car` field holds the seconds measured in the same way, and whose `cdr` field extends the precision by giving number of microseconds as an integer value between 0 and 999999.

The `time-nsec` function is similar to `time-usec` except that the returned cons cell's `cdr` field gives a number of nanoseconds as an integer value between 0 and 999999999.

Note: on hosts where obtaining nanosecond precision is not available, the `time-nsec` function obtains a microseconds value instead, and multiplies it by 1000.

9.54.2 Functions `time-string-local` and `time-string-utc`

Syntax:

```
(time-string-local time format)
(time-string-utc time format)
```

Description:

These functions take the numeric time returned by the `time` function, and convert it to a textual representation in a flexible way, according to the contents of the *`format`* string.

The `time-string-local` function converts the time to the local timezone of the host system. The `time-string-utc` function produces time in UTC.

The *`format`* argument is a string, and follows exactly the same conventions as the format string of the C library function `strftime`.

The *`time`* argument is an integer representing seconds obtained from the time function or from the `car` field of the cons returned by the `time-usec` function.

9.54.3 Functions `time-fields-local` and `time-fields-utc`

Syntax:

```
(time-fields-local time)
(time-fields-utc time)
```

Description:

These functions take the numeric time returned by the time function, and convert it to a list of seven fields.

The `time-string-local` function converts the time to the local timezone of the host system. The `time-string-utc` function produces time in UTC.

The fields returned as a list consist of six integers, and a Boolean value. The six integers represent the year, month, day, hour, minute and second. The Boolean value indicates whether daylight savings time is in effect (always `nil` in the case of `time-fields-utc`).

The *time* argument is an integer representing seconds obtained from the `time` function or from the `time-usec` function.

9.54.4 Structure `time`

Syntax:

```
(defstruct time nil
   year month day hour min sec
   wday yday
   dst gmtoff zone)
```

Description:

The `time` structure represents a time broken down into individual fields. The structure almost directly corresponds to the `struct tm` type in the ISO C language. There are differences. Whereas the `struct tm` member `tm_year` represents a year since 1900, the `year` slot of the `time` structure represents the absolute year, not relative to 1900. Furthermore, the `month` slot represents a one-based numeric month, such that 1 represents January, whereas the C member `tm_mon` uses a zero-based month. The `dst` slot is a **TXR Lisp** Boolean value. The slots `hour`, `min`, `sec`, `wday` and `yday` correspond directly to `tm_hour`, `tm_min`, `tm_sec`, `tm_wday` and `tm_yday`.

The slot `gmtoff` represents the number of seconds east of UTC, and `zone` holds a string giving the abbreviated time zone name. On platforms where the C type `struct tm` has fields corresponding to these slots, values for these slots are calculated and stored into them by the `time-struct-local` and `time-struct-utc` functions, and also the related `time-local` and `time-utc` methods. On platforms where the corresponding fields are not present in the C language `struct tm`, these slots are unaffected by those functions, retaining the default initial value `nil` or a previously stored value, if applicable. Lastly, the values of `gmtoff` and `zone` are not ignored by functions which accept a `time` structure as a source of input values.

9.54.5 Functions `time-struct-local` and `time-struct-utc`

Syntax:

```
(time-struct-local time)
(time-struct-utc time)
```

Description:

These functions take the numeric time returned by the time function, and convert it to an instance of the `time` structure.

The `time-struct-local` function converts the time to the local timezone of the host system. The `time-struct-utc` function produces time in UTC.

The *`time`* argument is an integer representing seconds obtained from the `time` function or from the `time-usec` function.

9.54.6 Functions `time-parse`, `time-parse-local` and `time-parse-utc`

Syntax:

```
(time-parse format string)
(time-parse-local format string)
(time-parse-utc format string)
```

Description:

The `time-parse` function scans a time description in *`string`* according to the specification given in the *`format`* string. If the scan is successful, a structure of type `time` is returned, otherwise `nil`.

The *`format`* argument follows the same conventions as the POSIX C library function `strptime`.

Prior to obtaining the time from *`format`* and *`string`* the returned structure is created and initialized with a time which represents time 0 ("the epoch") if interpreted in the UTC timezone as by the *`time-utc`* method.

The `time-parse-local` and `time-parse-utc` functions return an integer time value: the same value that would be returned by the `time-local` and `time-utc` methods, respectively, when applied to the structure object returned by `time-parse`. Thus, these equivalences hold:

```
(time-parse-local f s)  <-->  (time-parse f s).(time-local)
(time-parse-utc f s)  <-->  (time-parse f s).(time-utc)
```

Note: the availability of these three functions depends on the availability of `strptime`.

Note: on some platforms, like the GNU C Library, the `strptime` function supports the parsing of numeric and symbolic time zones. The `gmtoff` slot of the structure ends up being set accordingly. The `time-local` and `time-utc` functions take the `gmtoff` field into account, adjusting the returned time accordingly.

If these are specified.

9.54.7 Methods `time-local` and `time-utc`

Syntax:

```
time-struct.(time-local)
time-struct.(time-utc)
```

Description:

The `time` structure has two methods called `time-local` and `time-utc`.

The `time-local` function considers the slots of the `time` structure instance *`time-struct`* to be local time, and returns its integer representation as the number of seconds since the epoch.

The `time-utc` function is similar, except it considers the slots of *`time-struct`* to be in the UTC time zone.

Note: these functions work by converting the slots into arguments which are applied to `make-time` or `make-time-utc`.

Note: if the `gmtoff` slot is not `nil`, its value is added to the returned result.

9.54.8 Method `time-string`

Syntax:

```
time-struct.(time-string format)
```

Description:

The `time` structure has a method called `time-string`.

This method accepts a *`format`* string argument, which it uses to convert the fields to a character string representation which is returned.

The *`format`* argument is a string, and follows exactly the same conventions as the format string of the C library function `strftime`.

9.54.9 Method `time-parse`

Syntax:

```
time-struct.(time-parse format string)
```

Description:

The `time-parse` method scans a time description in *`string`* according to the specification given in the *`format`* string.

If the scan is successful, the structure is updated with the parsed information, and the remaining unmatched portion of *`string`* is returned. If all of *`string`* is matched, then an empty string is returned. Slots of *`time-struct`* which are originally `nil` are replaced with zero, even if these zero values are not actually parsed from *`string`*.

If the scan is unsuccessful, then `nil` is returned and the structure is not altered.

The *`format`* argument follows the same conventions as the POSIX C library function `strptime`.

Note: the `time-parse` method may be unavailable if the host system does not provide the `strptime` function. In this case, the `time-parse` static slot of the `time` struct is `nil`.

9.54.10 Functions `make-time` and `make-time-utc`

Syntax:

```
(make-time year month day
           hour minute second dst-advice)
(make-time-utc year month day
               hour minute second dst-advice)
```

Description:

The `make-time` function returns a time value, similar to the one returned by the `time` function. The `time` value is constructed not from the system clock, but from a date and time specified as arguments. The *`year`* argument is a calendar year, like 2014. The *`month`* argument ranges from 1 to 12. The *`hour`* argument is a 24-hour time, ranging from 0 to 23. These arguments represent a local time, in the current time zone.

The *`dst-advice`* argument specifies whether the time is expressed in daylight savings time (DST). It takes on three possible values: `nil`, the keyword `:auto`, or else the symbol `t`. Any other value has the same interpretation as `t`.

If *`dst-advice`* is `t`, then the time is assumed to be expressed in DST. If the argument is `nil`, then the time is assumed not to be in DST. If *`dst-advice`* is `:auto`, then the function tries to determine whether DST is in effect in the current time zone for the specified date and time.

The `make-time-utc` function is similar to `make-time`, except that it treats the time as UTC rather than in the local time zone. The *`dst-advice`* argument is supported by `make-time-utc` for function call compatibility with `make-time`. It may or may not have any effect on the output (since the UTC zone by definition doesn't have daylight savings time).

9.55 Data Integrity

9.55.1 Function `crc32-stream`

Syntax:

```
(crc32-stream stream [nbytes [crc-prev]])
```

Description:

The `crc32-stream` calculates the CRC-32 sum over the bytes read from *`stream`*, starting at the stream's current position.

If the *`nbytes`* argument is specified, it should be a nonnegative integer. It gives the number of bytes which should be read and included in the sum. If the argument is omitted, then bytes are read until the end of the stream.

The optional *`crc-prev`* argument defaults to zero. It is fully documented under the `crc32` function.

The `crc32-stream` functions returns the calculated CRC-32 as a nonnegative integer.

9.55.2 Function `crc32`

Syntax:

```
(crc32 obj [crc-prev])
```

Description:

The `crc32` function calculates the CRC-32 sum over *`obj`*, which may be a character string or a buffer.

If *obj* is a buffer, then the sum is calculated over all of the bytes contained in that buffer, according to its current length.

If *obj* is a character string, then the sum is calculated over the bytes which constitute its UTF-8 representation.

The optional *crc-prev* argument defaults to zero. If specified, it should be a nonnegative integer in the 32-bit range. This argument is useful when a single CRC-32 must be calculated in multiple operations over several objects. The first call should specify a value of zero, or omit the argument. To continue the checksum, each subsequent call to the function should pass as the *crc-prev* argument the CRC-32 obtained from the previous call.

The `crc32` function returns the calculated CRC-32 as a nonnegative integer.

Examples:

```
;; Single operation
(crc32 "ABCD") --> 3675725989

;; In two steps, demonstrating crc-prev argument:
(crc32 "CD" (crc32 "AB")) -> 3675725989
```

9.55.3 Functions `sha256-stream` and `md5-stream`

Syntax:

```
(sha256-stream stream [nbytes [buf]])
(md5-stream stream [nbytes [buf]])
```

Description:

The `sha256-stream` calculates the NIST SHA-256 digest over the bytes read from *stream*, starting at the stream's current position.

The `md5-stream` function calculates the MD5 digest, using the RSA Data Security, Inc. MD5 Message-Digest Algorithm.

If the *nbytes* argument is specified, it should be a nonnegative integer. It gives the number of bytes which should be read and included in the digest. If the argument is omitted, then bytes are read until the end of the stream.

If the *buf* argument is omitted, the digest value is returned as a new, buffer object. This buffer is 32 bytes long in the case of SHA-256, holding a 256-bit digest, and 16 bytes long in the case of MD5, holding a 128-bit digest. If the *buf* argument is specified, it must be a buffer that is at least 16 bytes long in the case of MD5, and at least 32 bytes long in the case of SHA-256. The hash is placed into that buffer, which is then returned.

9.55.4 Functions `sha256` and `md5`

Syntax:

```
(sha256 obj [buf])
(md5 obj [buf])
```

Description:

The `sha256` function calculates the NIST SHA-256 digest over *obj*, which may be a character string or a buffer.

Similarly, the `md5` functions calculates the MD5 digest over *obj*, using the RSA Data Security, Inc. MD5 Message-Digest Algorithm.

If *obj* is a buffer, then the digest is calculated over all of the bytes contained in that buffer, according to its current length.

If *obj* is a character string, then the digest is calculated over the bytes which constitute its UTF-8 representation.

If the *buf* argument is omitted, the digest value is returned as a new, buffer object. This buffer is 32 bytes long in the case of SHA-256, holding a 256-bit digest, and 16 bytes long in the case of MD5, holding a 128-bit digest. If the *buf* argument is specified, it must be a buffer that is at least 16 bytes long in the case of MD5, and at least 32 bytes long in the case of SHA-256. The hash is placed into that buffer, which is then returned.

9.55.5 Functions `sha256-begin`, `sha256-hash` and `sha256-end`

Syntax:

```
(sha256-begin)
(sha256-hash ctx obj)
(sha256-end ctx [buf])
```

Description:

The three functions `sha256-begin`, `sha256-hash` and `sha256-end` implement a stateful computation of SHA256 digest which allows multiple input sources to contribute to the result. Furthermore, the context object may be serially reused for calculating multiple digests.

The `sha256-begin` function, which takes no arguments, returns a new SHA256 digest-producing context object.

The `sha256-hash` updates the state of the SHA256 digest object *ctx* by including *obj* into the digest calculation. The *obj* argument may be: a character or character string, whose UTF-8 representation is digested; a buffer object, whose contents are digested; or an integer, representing a byte value in the range 0 to 255 included in the digest. The `sha256-hash` function may be called multiple times to include any mixture of strings and buffers into the digest calculation.

The `sha256-end` function finalizes the digest calculation and returns the digest in a buffer. If the *buf* argument is omitted, then a new 32-byte buffer is created for this purpose. Otherwise, *buf* must specify a `buf` object that is at least 32 bytes long. The digest is stored into this buffer and that the buffer is returned.

The `sha256-end` function additionally resets the *ctx* object into the initial state of a newly created context object, so that it may be used for another digest session.

9.55.6 Functions `md5-begin`, `md5-hash` and `md5-end`

Syntax:

```
(md5-begin)
(md5-hash ctx obj)
(md5-end ctx [buf])
```

Description:

The three functions `md5-begin`, `md5-hash` and `md5-end` implement a stateful computation of MD5 digest which allows multiple input sources to contribute to the result. Furthermore, the

context object may be serially reused for calculating multiple digests.

The `md5-begin` function, which takes no arguments, returns a new MD5 digest-producing context object.

The `md5-hash` updates the state of the MD5 digest object *ctx* by including *obj* into the digest calculation. The *obj* argument may be: a character or character string, whose UTF-8 representation is digested; a buffer object, whose contents are digested; or an integer, representing a byte value in the range 0 to 255 included in the digest. The `md5-hash` function may be called multiple times to include any mixture of strings and buffers into the digest calculation.

The `md5-end` function finalizes the digest calculation and returns the digest in a buffer. If the *buf* argument is omitted, then a new 16-byte buffer is created for this purpose. Otherwise, *buf* must specify a `buf` object that is at least 16 bytes long. The digest is stored into this buffer and that the buffer is returned.

The `md5-end` function additionally resets the *ctx* object into the initial state of a newly created context object, so that it may be used for another digest session.

9.56 The Awk Utility

The **TXR Lisp** library provides a macro called `awk` which is inspired by the Unix utility Awk. The macro implements a processing paradigm similar to that of the utility: it scans one or more input streams, which are divided into records or fields, under the control of user-settable regular-expression-based delimiters. The records and fields are matched against a sequence of programmer-defined conditions (called "patterns" in the original Awk), which have associated actions. Like in Awk, the default action is to print the current record.

Unlike Awk, the `awk` macro is a robust, self-contained language feature which can be used anywhere where a **TXR Lisp** expression is called for, cleanly nests with itself and can produce a return value when done. By contrast, a function in the Awk language, or an action body, cannot instantiate a local Awk processing machine.

The `awk` macro implements some of the most important Awk conventions and semantics, in Lisp syntax, while eschewing others. It does not implement the Awk convention that variables become defined upon first mention; variables must be defined to be used. It doesn't implement Awk's weak type system. A character string which looks like a number isn't a number, and an empty string or undefined variable doesn't serve as zero in arithmetic expressions enclosed in the macro. All expression evaluation within `awk` is the usual **TXR Lisp** evaluation.

The `awk` macro also does not provide a library of functions corresponding to those in the Awk library, nor does it provide counterparts various global variables in Awk such as the `ENVIRON` and `PROCINFO` arrays, or `RSTART` and `RLENGTH`. Such features of Awk are extraneous to its central paradigm.

9.56.1 Macro `awk`

Syntax:

```
(awk {(condition action*)}*)
```

Description:

The `awk` macro processes one or more input sources, which may be streams or files. Each input source is scanned into records, and each record is broken into fields. For each record, the sequence of condition-action clauses (except for certain special clauses) is processed. Every *condition* is evaluated, and if it yields true, the corresponding *action*s are evaluated.

The *condition* and *action* forms are understood to be in a scope in which certain local identifiers exist in the variable namespace as well as in the function namespace. These are called *awk functions* and *awk macros*.

If *condition* is one of the following keyword symbols, then it is a special clause, with special semantics: `:name`, `:let`, `:inputs`, `:output`, `:begin`, `:set`, `:end`, `:begin-file`, `:set-file` and `:end-file`. These clause types are explained below. In such a clause, the *action* expressions are not necessarily forms to be evaluated; the treatment of these expressions depends on the clause. Otherwise, if *condition* is not one of the above keyword symbols, the clause is an ordinary condition-action clause, and *condition* is a **TXR Lisp** expression, evaluated to determine a Boolean value which controls whether the *action* forms are evaluated. In every ordinary condition-action clause which contains no *action* forms, the `awk` macro substitutes the single action equivalent to the form `(prn)`: a call to the local `awk` function `prn`. The behavior of this macro, when called with no arguments, as above, is to print the current record (contents of the variable `rec`) followed by the output record terminator from the variable `ors`.

While the processing loop in `awk` scans an input source, it also binds the special variable `*stdin*` to the open stream associated with that source. This binding is in effect across all ordinary clauses, as well as across the special clauses `:begin-file` and `:end-file`.

The following is a description of the special clauses:

`(:name` *obj*`)`

The `:name` clause establishes the name of the implicit block contained within the expansion of the `awk` macro to be the object *obj*, usually a symbol. Forms enclosed in the macro can use `return-from` to abandon the `awk` form, specifying the same object as the argument.

If the `:name` form is omitted, the implicit block is named `awk`.

It is an error for two or more `:name` forms to appear.

Note: in **TXR** 255 and older, the `:name` clause must have an argument which is a symbol. The symbol `nil` is not permitted.

`(:let {`*sym* | `(`*sym init-form*`)}*)`

Regardless of what order they appear in relation to other clauses in the same `awk` macro, `:let` clauses are evaluated first before the macro takes any other action. The argument forms of this clause are variables or variable-init forms. They are treated the same way as analogous forms in the `let*` special form. Note that these are not enclosed in an extra list as they are in the that form. The bindings established by the `:let` clause have a scope which extends over all the other clauses in the `awk` macro.

If multiple `:let` clauses are present, they are effectively consolidated into a single clause, in the order they appear.

Note that the lexical variables, functions and macros established by the `awk` macro (called, respectively, *awk macros*, *awk functions* and *awk variables*) are in an inner scope relative to `:let` bindings. For instance if `:let` creates a binding for a variable called `fs`, that variable will be visible only to subsequent forms appearing in the same `:let` clause or later `:let` clauses, and also visible in `:inputs` and `:output` clauses. In `:begin`, `:set`, `:end`, and ordinary clauses, it will be shadowed by the `awk` variable `fs`, which holds the field-separator regular expression or string.

(:inputs *source-form**)

The :inputs clause is evaluated by the awk macro after processing the :let clauses. Each *source-form* is evaluated and the values of these forms are gathered into a list. This list then comprises the list of input sources for the awk processing task.

Each input source must be one of three kinds of objects. It may be a stream object, which must be capable of character input. It may be a list of strings, which awk will convert to an input stream as if by the make-strlist-input-stream function. Or else it must be a character string, which denotes a filesystem pathname which awk will open for reading.

If the :inputs clause is omitted, then a defaulting behavior occurs for obtaining the list of input sources. If the special variable *args* isn't the empty list, then *args* is taken as the input sources. Otherwise, the *stdin* stream is taken as the one and only input source.

If the awk macro uses *args* via the above defaulting behavior, it copies *args* and sets that variable to nil. This is done in order that if awk is used from the **TXR** command line, for example using the -e command-line option, after awk terminates, **TXR** will not try to open the next argument as a script file or treat it as an option. Note: programs which want awk not to modify *args* can explicitly specify *args* as the argument to the :inputs keyword, rather than allow *args* to be used through the defaulting behavior. Only the defaulting behavior consumes the arguments by overwriting *args* with nil.

It is an error to specify more than one :inputs clause.

(:output *output-form*)

The :output clause is processed just after the :inputs clause. It must have exactly one argument, which is an expression that evaluates to a string, or else to an output stream. If it evaluates to a string, then that string is used as the name of a file to open for writing, and the resulting stream is taken in place of that string.

The :output clause, if present, has the effect of creating a local binding for the *stdout* special variable. This new value of *stdout* is visible to all forms within the macro. If a :let clause is present, it establishes bindings in a scope which is nested within the scope established by :output. Therefore, *init-form*s in the :let may refer to the new value of *stdout* established by :output. Furthermore, :let can rebind *stdout*, causing the definition provided by :output to be shadowed.

In the case when the :output argument is a string such that a new stream is opened on the file, the awk macro will close that stream when it finishes executing. Moreover, that stream is treated uniformly as a member of the set of streams that are implicitly managed by the redirection macros in the same awk macro invocation. In brief, the implication is that if :output creates a stream for the file pathname "out.txt" and somewhere in the same awk macro, there is a redirection of the form, or equivalent to (-> "out.txt") then this redirection shall refer to the same stream that was established by :output. Note also that in this example situation, the expression (-> "out.txt" :close) has the effect of closing the :output stream.

(:begin *form**)

All :begin clauses are processed in the order in which they appear, before input processing begins. Each form is evaluated. These forms have in their scope the local awk variables and macros.

`(:set {place new-value}*)`
The `:set` clause provides a shorthand which allows the frequently occurring pattern `(:begin (set ...))` to be condensed to `(:set ...)`.

`(:end form*)`
All `:end` clauses are processed, in the order in which they appear, when the input processing loop terminates. This termination occurs when all records from all input sources are either processed or skipped, or else by an explicit termination such as a dynamic nonlocal transfer, such as `return-from`, or the throwing of an exception.

Upon termination, the end clauses are processed in the order they appear. Each `form` is evaluated, left to right.

In the normal termination case, the value of the last *form* of the last end clause appears as the return value of the `awk` macro.

Note that only termination of the `awk` macro initiated from condition-action clauses, `:begin-file` clauses, or `:end-file` clauses triggers `:end` clause processing. If termination of the `awk` macro is initiated from within a `:let`, `:inputs`, `:output` or `:begin` clause, then end clauses are not processed. If an `:end` clause performs a nonlocal transfer, the remaining `:end` forms in that clause and `:end` clauses which follow are not evaluated.

`(:begin-file form*)`
All `:begin-file` clauses are processed in the order in which they appear, before `awk` switches to each new input.

If both `:begin` and `:begin-file` forms are specified, then before the first input is processed, `:begin` clauses are processed first, then the `:begin-file` clauses.

`(:set-file {place new-value}*)`
The `:set-file` clause is a shorthand which translates `(:set-file ...)` to `(:begin-file (set ...))`.

`(:end-file form*)`
All `:end-file` clauses are processed after the processing of an input source finishes.

If both `:end` and `:end-file` forms are specified, then before after the last input is processed, `:end-file` clauses are processed first, then the `:end` clauses.

The `:end-file` clauses are processed unconditionally, no matter how the processing of an input source terminates, whether terminated naturally by running out of records, prematurely by invocation of the `next-file` macro, or via a dynamic nonlocal control transfer such as a block return or exception throw.

If a `:begin-file` clause performs a nonlocal transfer, `:end-file` processing is not triggered, because the processing of the input source is deemed not to have taken place.

`(:fields {sym | (sym [fun]) | -}*)`
The `:fields` clause may be specified in order to give symbolic names to fields, and optionally specify conversions for them. Every argument must be one of three expressions. It may be a bindable symbol other than − (minus). It may be a list whose first element is a symbol other than − optionally followed the name of a function. Or else it may be the − symbol, which has a special meaning. Symbols other than − may not be repeated, and the `:fields` clause may appear at most once in a given instance of the `awk` macro. Each argument is understood to correspond to a field expression for a

successive field, starting with the leftmost *sym* corresponding with the first field, `[f 0]`. Each *sym* other than − becomes the name of a symbol macro which denotes its corresponding field expression, expanded over the scope of the `awk` macro. The − symbol is a placeholder which doesn't bind a symbol macro to the corresponding field. Additionally, every two-element entry which associates the field symbol *sym* with a function name *fun* specifies a field conversion. After each record is read and divided into fields, those fields for which *fun* is specified are updated by passing their value to this function and replacing them by the returned value. The *fun* symbol may also be one of the shorthand symbols available in the `fconv` macro, such as `i`, `x` and others. If at least one such conversion is specified in a `:fields` clause, then the value of `rec` is updated from the converted fields in the usual manner, as if the fields had been assigned. Furthermore, it is ensured that every field for which a `:fields` clause specifies a conversion exists. Fields with an empty string value are automatically added so that a field exists for the rightmost conversion, and the value of `nf` is updated to include these fields.

`(`*condition action*`*`*`)`

Clauses which do not have one of the specially recognized keywords in the first position are ordinary condition-action clauses. After processing the `:begin` clauses, `awk` enters a loop in which it extracts successive records from the input sources according to the `rs` (record separator) variable. Each record is divided into fields according to the `fs` (field separator) variable, and various `awk` variables are updated. Then, the condition-action clauses are processed, in the order in which they appear. Each *condition* is evaluated. If the resulting value is a regular expression or a function, then this regular expression or function is invoked on the value stored in the record variable `rec`, and the result is taken to be the truth value of *condition*. Otherwise, if the resulting value of *condition* is other than a function or regular expression, it is taken directly to be the truth value. If the condition is true, then its associated *action* forms are evaluated. These forms have access to the truth value via the `res` variable, which is freshly bound for each execution of the *action* forms of that specific clause. For each input record, all condition-action clauses are processed in the order they appear, regardless of which of them have a true action, except in cases when some *action* invokes the `next` or `next-file` macro, or abandons the execution of the `awk` macro entirely via a non-local exit. When an input source runs out of records, `awk` switches to the next input source. When there are no more input sources, the macro terminates.

9.56.2 Variables `rec` and `orec`

Description:

The `awk` variable `rec` holds the current record. It is automatically updated prior to the processing of the condition-action clauses. Prior to the extraction of the first record, its value is `nil`.

It is possible to assign to `rec`. The value assigned to `rec` must be a character string. Immediately upon the assignment, the character string is delimited into fields according to the field separator `awk` variable `fs`, and these fields are assigned to the field list `f`. At the same time, the `nf` variable is updated to reflect the new number of fields. Likewise, modification of these variables causes `rec` to be reconstructed by a catenation of the textual representation of the fields in `f` separated by copies of the output field separator `ofs`.

The `orec` variable ("original record") also holds the current record. It is automatically updated prior to the processing of the condition-action clauses at the same time as `rec` with the same contents. Like `rec`, it is initially `nil` before the first record is read. The `orec` variable is unaffected by modification of the variables `rec`, `f` and `nf`. It may be assigned. Doing so has no effect on any other variable.

9.56.3 Variable `f`

Description:

The `awk` variable `f` holds the list of fields. Prior to the first record being read, its value is `nil`. Whenever a new record is read, it is divided into fields according to the field separator variable `fs`, and these fields are stored in `f` as a list of character strings.

If the variable `f` is assigned, the new value must be a sequence. The variable `nf` is automatically updated to reflect the length of this sequence. Furthermore, the `rec` variable is updated by catenating a string representation of the elements of this sequence, separated by the contents of the `ofs` (output field separator) `awk` variable.

Note that assigning to a DWIM bracket form which indexes `f`, such as for instance `[f 0]` constitutes an implicit modification of `f`, and triggers the recalculation of `rec`. Modifications of the `f` list which do not involve an implicit or explicit assignment to the variable `f` itself do not have this recalculating effect.

Unlike in Awk, assigning to the nonexistent field `[f` *m*`]` where *m* >= `nf` is erroneous.

9.56.4 Variable `nf`

Description:

The `awk` variable `nf` holds the current number of fields in the sequence `f`. Prior to the first record being read, it is initially zero.

If `nf` is assigned, then `f` is modified to reflect the new number of fields. Fields are deleted from `f` if the new value of `nf` is smaller. If the new value of `nf` is larger, then fields are added. The added fields are empty strings, which means that `f` must be a sequence of a type capable of holding elements which are strings.

If `nf` is assigned, then `rec` is also recalculated, in the same way as described in the documentation for the `f` variable.

9.56.5 Variable `nr`

Description:

The `awk` variable `nr` holds the current absolute record number. Record numbers start at 1. Absolute means that this value does not reset to 1 when `awk` switches to a new input source; it keeps incrementing for each record. See the `fnr` variable.

Prior to the first record being read, the value of `nr` is zero.

9.56.6 Variable `fnr`

Description:

The `awk` variable `fnr` holds the current record number within the file. The first record is 1.

Prior to the first record being read from the first input source, the value of `fnr` is zero. Thereafter, it resets to 1 for the first record of each input source and increments for the remaining records of the same input source.

9.56.7 Variable `arg`

Description:

The `awk` variable `arg` is an integer which indicates what input source is being processed. Prior to input processing, it holds the value zero. When the first record is extracted from the first input source, it is set to 1. Thereafter, it is incremented whenever `awk` switches to a new input source.

9.56.8 Variable `fname`

Description:

The `awk` variable `fname` provides access to a character string which, if the current input is a file stream, is the name of the underlying file. Assigning to this variable changes its value, but has no effect on the input stream. Whenever a new input source is used by `awk`, this variable is set from the file name on which it is opening a stream. When using an existing stream rather than opening a file, `awk` sets this variable from the `:name` property of the stream.

Note that the redirection macros `<-` and `<!` have no effect on this variable. Within their scope, `fname` retains its value.

9.56.9 Variable `rs`

Description:

The `awk` variable `rs` specifies a string or regular expression which is used for delimiting characters read from the inputs into pieces called records.

Note: the record extraction is internally implemented using record streams instantiated by the `record-adapter` function.

The regular-expression pattern stored in `rs` is used to matches substrings in the input which separate or terminate records. Unless the `krs` variable is set true, the substrings which match `rs` are discarded and the records consist of the nonmatching extents between them.

The initial value of `rs` is `"\n"`: the newline character. This means that, by default, records are lines.

If `rs` is changed to the value `nil`, then record separation operates in *paragraph mode*, which is described below.

If a match for the record separator occurs at the end of the stream, it is not considered to delimit an empty record, but acts as the terminator for the previous record.

When a new value is assigned to `rs`, it has no effect on the most recently scanned and delimited record which is still current, or previous records. The new value applies to the next, not yet read record.

In paragraph mode, records are separated by a newline character followed by one or more blank lines (empty lines or lines containing only a mixture of tabs and spaces). This means that, effectively, the record-separating sequences match the regular expression `/\n[ \n\t]*\n/`.

There are two differences between paragraph mode and simply using the above regular expression as `rs`. The first difference is that if the first record which is read upon entering paragraph mode is empty (because the input begins with a match for the separator regex), then that record is thrown away, and the next record is read. The second difference is that, if field separation based on the `fs` variable is in effect, then regardless of the value of `fs`, newline characters separate fields.

Therefore, the programmer-defined `fs` doesn't have to include a match for newline. Moreover, if it is a simple fixed string, it need not be converted to a regular expression which also matches a newline.

9.56.10 Variable `krs`

Description:

The `awk` variable `krs` stands for "keep record separator". It is a Boolean variable, initialized to `nil`.

If it is set to a true value, then the separating text matched by the pattern in the `rs` variable is retained as part of the preceding record rather than removed.

When a new value is assigned to `krs`, it has no effect on the most recently scanned and delimited record which is still current, or previous records. The new value applies to the next, not yet read record.

9.56.11 Variables `fs` and `ft`

Description:

The `awk` variable `fs` and `ft` each specify a string or regular expression which is used for each record that is stored in the `rec` variable into fields.

Both variables are initialized to `nil`, in which case a default behavior is in effect, described below.

Use of these variable is mutually exclusive; it is an error for both of these variables to simultaneously have a value other than `nil`. The value stored in either variable must be `nil`, a character string or a regular expression. If it contains a string or regex, it is said to contain a pattern. A string value effectively behaves as a fixed regular expression which matches the sequence of characters in the string verbatim, without treating any of them as regex operators.

The splitting of `rec` into fields is influenced by the Boolean `kfs` ("keep field separators") variable, whose effect is discussed in its description. If `kfs` is false, the splitting is carried out as follows.

If `fs` contains a pattern, then `rec` is treated specially when it is the empty string: in that case, the pattern in `fs` is ignored, and no fields are produced: the field list `f` is the empty list, and `nf` is zero. A nonempty record is split by searching it for matches for the `fs` pattern. If a match does not occur, then the entire record is a field. If one match occurs, then the record is split into two fields, either of which, or both, might be empty. If two matches occur, the record is split into three fields, and so on. If `fs` finds only an empty string match in the record, then it is considered to match each of the empty strings between two consecutive characters of the record. Consequently, the record is split into its individual characters, each one becoming a field. Note: all of these behaviors, except for the special treatment of the empty record, are accomplished by a call to the `split-str` function.

If the variable `ft` ("field tokenize") contains a pattern, that pattern is used to positively recognize tokens within the input record, rather than to match separating material between them. Those matching tokens then constitute the fields. The tokenizing is performed using the `tok-str` function.

If `fs` and `ft` are both `nil`, as is initially the case, then the splitting into fields is performed as if the `ft` variable held the regular expression `/[^\n\t ]+/`. This means that, by default, fields are sequences of consecutive characters which are not spaces, tabs or newlines. Newlines are

excluded from fields (and thus separate them) because they can occur in a record when the value of the record separator `rs` is customized.

9.56.12 Variable `kfs`

Description:

The `awk` variable `kfs` is a Boolean flag which is initialized to `nil`.

If it is set to any other value, it indicates a request to retain the pieces of the record which separate the fields (even when they are empty strings). The retained pieces appear as fields, interspersed among the regular fields so that all of the fields appear in the order in which they were extracted from the record.

When `kfs` is set, and tokenization-style delimiting is in effect due to `ft` being set, there is always at least one field, even if the record is empty. If the record doesn't match the tokenizing regular expression in `ft` then a single field is generated, then the entire record is taken as one field, denoting the nonmatching space, even if the record is the empty string.

If the record matches one or more tokens, then the first and last field will always contain the nonmatching material before the first and last token, respectively. This is true even if the material is empty. Thus `[f 0]` always has the material before the first token, whether or not the first token is matched immediately at the first character position in the record. This behavior follows from the semantics of the `keep-sep` parameter of the `tok-str` function.

Similarly, when splitting based on `fs` is in effect and `kfs` is set, there is always at least one field, even if the record is empty. If `fs` finds no match in the record, then the entire record, even if empty, is taken as one field. In that case, there are no separator to retain. When `fs` finds one or more matches, then these are included as fields. Separators are always between the fields. If a separator finds a nonempty match at the beginning of a record, that causes an empty field to be split off: the separator is understood as intervening between an empty string before the first character of the record, and subsequent material which follows the text matched by the separator. Thus the first field is an empty field, and the second is the matched text which is included due to `kfs` being set. An analogous situation occurs at the end of the record: if `fs` matches a nonempty string at the tail of the record, it splits off an empty last field, preceded by a field holding the matched separator portion. Empty matches are only permitted to occur between the characters of the record, not before the first character of after the last. If `fs` matches the entire record, then there will be three fields: the first and last of these three will be empty strings, and the middle field, the separator, will be a copy of the record. Under `kfs`, empty matches cause empty string to be included among the fields. All of this follows from the semantics of the `keep-sep` parameter of the `split-str` function.

9.56.13 Variable `fw`

Description:

The `awk` variable `fw` controls the fixed-width-based delimiting of records into fields.

The variable is initialized to `nil`. In that state, it has no effect. When this variable holds a non-`nil` value, it is expected to be a list of integers. The use of the `fs` or `ft` variables is suppressed, and fields are extracted according to the widths indicated by the list. The fields are consecutive, such that if the list is `(5 3)` then the first five characters of the record are identified as field `[f 0]` and the next three characters after that as `[f 1]`.

Only complete fields are extracted from the record. If, after the extraction of the maximum possible complete fields, more characters remain, those characters are assigned to an extra field.

An empty record produces an empty list of fields regardless of the integers stored in fw.

A zero width extracts a zero length field, except when no more characters remain in the record.

If `nil` is stored into `fw` then control over field separation is relinquished to the `fs` or `ft` variables, according to their current values.

If `fw` holds a value other than `nil` or else a list of nonnegative integers, the behavior is unspecified.

Examples

The following table shows how various combinations of the value the input record `rec` and field widths in the variable `fw` give rise to field values `f`:

```
rec    fw      f
---------------------------------
"abc"  (0)     ("" "abc")
"abc"  (2)     ("ab" "c")
"abc"  (1 2)   ("a" "bc")
"abc"  (1 3)   ("a" "bc")
"abc"  (1 1)   ("a" "b" "c")
"abc"  (3)     ("abc")
"abc"  (4)     ("abc")
""     (4)     nil
""     (0)     nil
```

9.56.14 Variable `ofs`

Description:

The `awk` variable `ofs` hold the output field separator. Its initial value is a string consisting of a single space character.

When the `prn` function prints two or more arguments, or fields, the value of `ofs` is used to separate them.

Whenever `rec` is implicitly updated due to a change in the variable `f` or `nf`, `ofs` is used to separate the fields, as they appear in `rec`.

9.56.15 Variable `ors`

Description:

The `awk` variable `ors`, though it stands for "output record separator" holds what is in fact the output record terminator. It is named after the `ORS` variable in Awk.

Each call to the `prn` function terminates its output by emitting the value of `ors`.

The initial value of `ors` is a character string consisting of a single newline, and so the `prn` function prints lines.

9.56.16 Variable `res`

The `awk` variable `res` is implicitly bound over the scope of the action forms of every condition-action clause. It holds the result of the condition form.

Because the action forms execute only if the condition yields true, it follows that `res` is never observed with a value of `nil` unless the program explicitly assigns that value.

Note: this is an original feature in the **TXR Lisp** `awk` macro, which has no counterpart in POSIX or GNU Awk.

Example:

```
(awk
  (:inputs '("carpet"))
  (#/a.*p/ (prn res)))

Output:

arp
```

In this example, the result of the `#/a.*p/` regular expression being applied to the input `carpet` is the string `"arp"` and so over that clause, `prn` takes on that string as its value. Thus, thanks to `prn`, the action has access to the matching part of the record.

9.56.17 Function `prn`

Syntax:

```
(prn form*)
```

Description:

The `awk` function `prn` performs output into the `*stdout*` stream. The `:output` clause affects the destination by rebinding `*stdout*`.

If called with no arguments, `prn` prints `rec` followed by `ors`.

Otherwise, it prints the values of the arguments, separated by `ofs`, followed by `ors`.

When a condition-action clause specifies no action forms, then a call to `prn` with no arguments is the default action.

Each argument *form* is printed by conversion to a string, as if by the expression `` `@val` `` where `val` is some variable which holds the value produced by the evaluation of *form*. Thus if the value is `nil`, the output for that argument is an empty string, rather than the text `"nil"`.

9.56.18 Macro `next`

Syntax:

```
(next)
```

Description:

The `awk` macro `next` may be invoked in a condition-pattern clause. It terminates the processing of that clause, and all subsequent clauses, causing `awk` to process the next record, if there is one. If there is no next record, `awk` terminates.

9.56.19 Macro `again`

Syntax:

```
(again)
```

Description:

The `awk` macro `again` may be invoked in a condition-pattern clause. It terminates the processing of that clause, and all subsequent clauses. Then, the current value of the record, namely the datum stored in the `awk` variable `rec`, is delimited into fields, and all of the condition-pattern clauses are processed again.

No other state is modified. In particular, the record number `nr` and the `orec` variable holding the original record both retain their current values.

Note: this is an original feature in the **TXR Lisp** `awk` macro, which has no counterpart in POSIX or GNU Awk.

9.56.20 Macro `next-file`

Syntax:

```
(next-file)
```

Description:

The `awk` macro `next-file` may be invoked in a condition-pattern clause. It terminates the processing of that clause and all subsequent clauses. Then `awk` abandons the current input source and moves to the next one. If there is no next input source, `awk` terminates.

9.56.21 Macros `rng`, `-rng`, `rng-`, `-rng-`, `--rng`, `--rng-`, `rng+`, `-rng+` and `--rng+`

Syntax:

```
(rng from-condition to-condition)
(-rng from-condition to-condition)
(rng- from-condition to-condition)
(-rng- from-condition to-condition)
(--rng from-condition to-condition)
(--rng- from-condition to-condition)
(rng+ from-condition to-condition)
(-rng+ from-condition to-condition)
(--rng+ from-condition to-condition)
```

Description:

The nine `awk` macros in the `rng` family may be used anywhere within an ordinary condition-pattern `awk` clause.

Each provides a Boolean test which is true if the current record lands within a range of records delimited by conditions. Each provides its own distinct, useful nuance, which is identified by the mnemonic characters prefixed or suffixed to the name.

The basic `rng` macro inclusively matches ranges of records. Each such range begins with a record for which *`from-condition`* yields true, and ends on the record for which *`to-condition`* is true. What it means to match is that the `rng` expression yields a Boolean true value when it is evaluated in the context of processing any of the records which are included in the range.

The table below summarizes the semantic variations of these nine range macro operators. The leftmost column labeled `DATA` represents the stream of records being processed. Each entry in this column gives the literal piece of text which comprises the content of one record in the stream. The remaining nine columns, labeled with the nine range operators, inform about the behavior of these operators with respect to these records. In each of these columns the letter `X` marks those records

for which the column's range operator yields true, if it is invoked with the arguments `#/H/` and `#/T/` as its *from-condition* and *to-condition*, respectively. For example, the `rng` column shows the values of the `(rng #/H/ #/T/)` expression, indicating that the expression starts being true when the `H1` record is seen, stays true for the `T1` record, and then reverts to false:

```
DATA    rng -rng rng- -rng- --rng --rng- rng+ -rng+ --rng+
-------------------------------------------------------------
PROLOG
H1      X        X                        X
H2      X    X   X     X                  X     X
H3      X    X   X     X                  X     X
B1      X    X   X     X      X     X     X     X      X
B2      X    X   X            X     X     X     X      X
T1      X    X                X           X     X      X
T2                                        X     X      X
T3                                        X     X      X
EPILOG
```

The prefix and suffix characters of the operator names are intended to be mnemonic. A single `-` (dash) indicates the exclusion of one record. A double `--` (dash dash) indicates the exclusion of all leading records which match *from-condition*; this appears on the left side only. The `+` character, appearing on the right only, indicates that all consecutive records which match *to-condition* are included in the range, not only the first one.

Ranges are oblivious to the division between successive sources of input; a range can start in one file of records and terminate in another. To prevent a range from spanning input transitions, additional complexity is required in the expression.

Ranges expressed using the `rng` family macros may combine with other expressions, including other ranges, and allow arbitrary nesting: the *from-condition* or *to-condition* can be a range, or an expression containing ranges.

The expressions *from-condition* and *to-condition* are ordinary expressions which are evaluated. However, their evaluation is unusual in two ways.

Firstly, if either expression produces, as its result, a function or regular-expression object, then that function or regular-expression object is applied to the current record (value of the `rec` variable), and the result of that application is then taken as the result of the condition. This allows for expressions like `(rng (f^ #/start/) #/end/)` which denotes a range which begins with a record which begins with the prefix `"start"` and ends with a record which contains `"end"` as a substring.

Secondly, the conditions are evaluated out of order with respect to the surrounding expression in which they occur. Ranges and their constituent *from-condition* and *to-condition* are evaluated just prior to the processing of the condition-action clauses. Each `rng` expression is reduced to a Boolean value. Then, when the condition-action clauses are processed and their *condition* and *action* forms are evaluated, each occurrence of a `rng` expression simply denotes its previously evaluated Boolean value.

Therefore, it is not possible for expressions to short circuit the evaluation of ranges. Ranges cannot "miss" their starting or terminating conditions; every range occurring anywhere in the condition-action clauses is tested against every record that is processed.

Because of this perturbed evaluation order, code which happens to place side effects into ranges

may produce surprising results.

For instance, the expression `(if nil (rng (prinl 'hello) (prinl 'world)))` will produce output even though the `if` condition is `nil`, and, moreover, this output will happen before the clauses are processed in which this `if` expression appears. At the time when the `if` itself is evaluated, the `rng` expression merely fetches a previously computed Boolean value which indicates whether the range is active for this record.

Also, the behavior is unspecified if range expressions attempt to modify any of the special `awk` variables `rec`, `f`, `fs`, `ft` and `kfs`. It is not recommended to place any side effects into range expressions.

A more detailed description of the range operators follows.

`(rng` *`from to`*`)`
: This type of range becomes active when a record is encountered for which the *`from`* expression yields true. While the range is active, the expression evaluates true. If, when the range is active, a record is encountered for which the *`to`* expression yields true, the range remains active for that record and is deactivated after the completion of processing for that record. If the range is inactive and a record is encountered or which both *`from`* and *`to`* are true, then the range is activated for that record and then deactivated when that record is processed. Records for which *`from`* and *`to`* are not true do not affect the range's activation state.

`(-rng` *`from to`*`)`
: This type of range is active under the same conditions as the `rng` type. However, the expression yields a Boolean false value for the first record which begins a range. That is to say, when the range is inactive, and a record is scanned for which *`from`* is true, the range activates, but the range expression yields `nil`. This is true regardless of whether the *`to`* expression yields true for that record. If there are additional records in the range, the expression yields a true value for those records.

`(rng-` *`from to`*`)`
: This type of range is active under the same conditions as the `rng` type. However, the range expression yields `nil` for the record for which `to` yields true which terminates the range. This occurs even if that is the same record which activated the range by triggering the *`from`* condition. Note that if a range terminates abruptly due to no more records being available, the range expression still yields true for the last record.

`(-rng-` *`from to`*`)`
: This type of range is active under the same conditions as the `rng` type. However, the range expression yields `nil` for the first record which activates the range, and for the last record which deactivates the range by activating the `to` condition. If the range is active over fewer than three records, then the expression never yields true for that range. If the range terminates abruptly due to no more records being available, and if the last record processed isn't the one which activated the range due to triggering the `from` condition, the expression yields true for that record.

`(--rng` *`from to`*`)`
: This type of range is active under the same conditions as `rng`. However, the range expression yields `nil` for the entire leading sequence of consecutive records for which *`from`* is true. If *`from`* is true of the *`to`* record which terminates the range, `nil` is returned for that record also.

`(--rng-` *`from to`*`)`
: This type of range is active under the same conditions as `rng`. However, the range expression yields `nil` for the entire leading sequence of consecutive records for which *`from`* is true, and also yields nil for the last record which triggers the *`to`* condition.

`(rng+ from to)`
: This range is active under different conditions compared to `rng`. Though it becomes active in the same way, when the *from* expression yields true, the deactivation logic is different. The range is deactivated when a record for which *to* is true is followed by a record for which *to* is not true. That record is excluded from the range; if the *from* expression happens to be true for that record, a new range begins at that record. Thus, effectively, the range is terminated not by single record which triggers *to* but by a sequence of one or more such consecutive records.

`(-rng+ from to)`
: This range is active under the same conditions as `rng+`. However, the range expression yields `nil` for the first record in the range. If the range contains only one record, then it returns `nil` for that record.

`(--rng+ from to)`
: This range is active under the same conditions as `rng+`. However, the range expression yields `nil` for the entire leading sequence of consecutive records for which *from* is true. This is the case even for those for which the *to* expression is true.

9.56.22 Macro `ff`

Syntax:

```
(ff opip-arg *)
```

Description:

The `awk` macro `ff` (filter fields) provides a shorthand for filtering the field list `f` trough a pipeline of chained functions expressed using `opip` argument syntax.

The following equivalence holds, except that `f` refers to the `awk` variable even if the `ff` invocation occurs in code which establishes a binding which shadows `f`.

```
(ff a b c ...)  <-->  (set f [(opip a b c ...) f])
```

Example:

```
;; convert all fields from string to floating-point
(ff (mapcar flo-str))
```

9.56.23 Macro `mf`

Syntax:

```
(mf opip-arg *)
```

Description:

The `awk` macro `mf` (map fields) provides a shorthand for mapping each field individually trough a pipeline of chained functions expressed using `opip` argument syntax.

The following equivalence holds, except that `f` refers to the `awk` variable even if the `mf` invocation occurs in code which establishes a binding which shadows `f`.

```
(mf a b c ...)  <-->  (set f (mapcar (opip a b c ...) f))
```

Example:

```
;; convert all fields from string to floating-point
```

```
(mf flo-str)
```

9.56.24 Macro `fconv`

Syntax:

```
(fconv {clause | : | - }*)
```

Description:

The `awk` macro `fconv` provides a succinct way to request conversions of the textual fields. Conversions are expressed by clauses which correspond with fields.

Each *`clause`* is an expression which must evaluate to a function. The clause is evaluated in the same manner as the argument a `dwim` operator, using Lisp-1-style name lookup. Thus, functions may be specified simply by using their name as a *`clause`*.

Furthermore, several local functions exist in the scope of each *`clause`*, providing a shorthand notation. These are described below.

Conversion proceeds by applying the function produced by a clause to the field to which that clause corresponds, positionally. The return value of the function applied to the field replaces the field.

When a clause is specified as the symbol `-` (minus) it has a special meaning: this minus clause occupies a field position and corresponds to a field, but performs no conversion on its field.

The `:` (colon) keyword symbol isn't a clause and does not correspond to a field position. Rather, it acts as a separator among clauses. It need not appear at all. If it appears, it may appear at most twice. Thus, the clauses may be separated into up to three sequences.

If the colon does not appear, then all the clauses are *prefix clauses*. Prefix clauses line up with fields from left to right. If there are fewer fields than prefix clauses, the values of the excess clauses are evaluated, but ignored. Vice versa, if there are fewer prefix clauses than fields, then the excess fields are not subject to conversions.

If the colon appears once, then the clauses before the colon, if any, are prefix clauses, as described in the previous paragraph. Clauses after the colon, if any, are *interior clauses*. Interior clauses apply to any fields which are left unconverted by the prefix clauses. All interior clauses are evaluated. If there are fewer fields than interior clauses, then the values of the excess interior clauses are ignored. If there are more fields than clauses, then the clause values are cycled: reused from the beginning against the excess fields, enough times to convert all the fields.

If the colon appears twice, then the clauses before the first colon, if any, are prefix clauses, the clauses between the two colons are interior clauses, and those after the second colon are *suffix clauses*. The presence of suffix clauses change the behavior relative to the one-colon case as follows. After the conversions are performed according to the prefix clauses, the remaining fields are counted. If there are are only as many fields as there are suffix clauses, or fewer, then the interior clauses are evaluated, but ignored. The remaining fields are processed against the suffix clauses. If after processing the prefix clauses there are more fields remaining than suffix clauses, then a number of rightmost fields equal to the number of suffix clauses is reserved for those clauses. The interior fields are applied only to the unreserved middle fields which precede these reserved rightmost fields, using the same repeating behavior as in the one-colon case. Finally, the previously reserved rightmost fields are processed using the suffix clauses.

The following special convenience functions are in scope of the clauses, effectively providing a

shorthand for commonly-needed conversions:

`i` Provides conversion to integer. It is identical to the `toint` function, with the default radix.

`o` Converts a string value holding an octal representation to the integer which it denotes. It is equivalent to `toint` with a *`radix`* argument of 8.

`x` Converts a string value holding a hexadecimal representation to the integer which it denotes. It is equivalent to `toint` is equivalent to with a *`radix`* argument of 16.

`b` Converts a string value holding a binary (base two) representation to the integer which it denotes. It is equivalent to `toint` with a *`radix`* argument of 2.

`c` Converts a string value holding a C-language-style representation to the integer which it denotes, meaning that the `0x` prefix denotes a hexadecimal value, a leading zero octal, otherwise decimal. These prefixes follow the + or − sign, if present. The `c` function is equivalent to `toint` invoked with a *`radix`* argument of `#\c`.

`r` Converts a string holding a floating-point representation to the floating-point value which it denotes. It is equivalent to `tofloat`.

`iz`, `oz`, `xz`, `bz`, `cz` and `rz`
Conversion similar to `i`, `o`, `x`, `b`, `c` and `r`, but equivalent to using the functions `tointz` and `tofloatz`. Thus fields which are non-numeric strings or the object `nil` get converted to 0, or 0.0 in the case of `rz`.

− Performs no conversion: the corresponding field is taken as-is.

Because `fconv` macro destructively operates on the elements of the field list `f`, it has the same effect as an assignment to the fields: the value of `rec` is updated.

The return value of `fconv` is `f`.

Note: because `f` is `nil` when no fields have been extracted, a `fconv` expression can be used as the condition in an `awk` clause which triggers the action if one or more fields have been extracted, and performs conversions on them.

Note: although `fconv` is intended for converting textual fields, and the semantic descriptions below consequently make references to string inputs, the behavior of `fconv` with respect to non-string fields can be inferred. For instance if a field actually holds the floating-point value 3.14, and the `i` conversion is applied to it, it will produce 3, because it works by means of the `toint` function.

Note: a somewhat less flexible mechanism for converting fields, related to `fconv`, is present in the `:fields` clause of the `awk` macro, which can specify names for the positional fields, along with conversion functions. The `:fields` clause has different syntax, and doesn't support the `:` (colon) separator, instead assuming a fixed number of fields enumerated from the left.

Examples:

```
;; convert up to first three fields to integer:
(awk ((fconv i i i)))

;; convert all fields to floating-point
(awk ((fconv : r :)))

;; convert first and second fields to integer
;; from hexadecimal;
;; convert last field to integer from octal;
```

```
;; process pairs of fields in between
;; these by leaving the first element of
;; each pair unconverted and converting second
;; to floating-point;
(awk ((fconv x x : - r : o)))

;; convert all fields, except the first,
;; from integer, turning empty strings
;; and non-integer junk as zero;
;; leave first field unconverted:
(awk ((fconv - : iz)))
```

9.56.25 Macros `->`, `->>`, `<-`, `!>` and `<!`

Syntax:

```
(-> path form*)
(->> path form*)
(<- path form*)
(!> command form*)
(<! command form*)
```

Description:

These `awk` macros provide convenient redirection of output and input to and from files and commands.

When at least one *form* argument is present, the functions `->`, `->>` and `!>` evaluate each *form* in a dynamic environment in which the `*stdout*` variable is bound to a file output stream, for the first two functions, or output command pipe in the case of the last one.

Similarly, when at least *form* argument is present, the remaining functions `<-` and `<!` evaluate each *form* in a dynamic environment in which `*stdin*` is bound to a file input stream or input command pipe, respectively.

The *path* and *command* arguments are treated as forms, and evaluated. They should evaluate to strings.

The first evaluation of one of these macros for a given *path* or *command* being used in a particular direction (input or output) and type (file or command) creates a stream. That stream is then associated with the given *path* or *command* string, together with the direction and type. Upon a subsequent evaluation of one of these macros for the same *path* or *command* string, direction and type, a new stream is not opened; rather, the previously associated stream is used.

The `->` macro indicates that the file named *path* is to be opened for writing and overwritten, or created if it doesn't exist. The `->>` macro indicates that the file named by *path* is to be opened in append mode, created if necessary. The `<-` macro indicates that the file given by *path* is to be opened for reading.

The `!>` macro indicates that *command* is to be opened as an output command pipe. The `<!` macro indicates that *command* is to be opened as an input command pipe.

If any of these macros is invoked without any *form* arguments, then it yields the stream object associated with *path* or *command* argument, direction and type. If the association doesn't exist, the stream is first created.

If *form* arguments are present, then the value of the last one is yielded as a value, except in the case when the last form yields the `:close` keyword symbol.

If the last *form* yields the `:close` keyword symbol, the association between the *path* or *command*, direction and type and the stream is removed, and the stream is closed. In this case, the result value of the macro isn't the `:close` symbol, but rather the return value of the *close-stream* call that is implicitly applied to the stream.

Even if there is only one *form* which yields `:close`, the stream is created, if it doesn't exist prior to the macro invocation.

In each invocation of these macros, after every *form* is evaluated, the stream is implicitly flushed, if it is an output stream.

The association between the *pipe* or *command* strings, direction and type is scoped to the innermost enclosing `awk` macro. An inner `awk` macro cannot refer to the associations established in an outer `awk` macro. An outer `awk` macro can obtain an association's stream object and communicate that stream to the nested macro where it can be used.

When the surrounding `awk` macro terminates, all of the streams opened by these redirection macros are closed, without breaking those associations. If lexical closures are captured inside the macro, and then invoked after the macro has terminated, and inside those closures the redirection macros are used, those macro instances will with closed stream objects, and so attempts to perform I/O will fail.

9.56.26 Examples of `awk` Macro Usage

The following examples are `awk` macro equivalents of the examples of the POSIX `awk` utility given in IEEE Std 1003.1, 2013 Edition.

1. Print lines for which field 3 is greater than 5:

```
;; print lines with fields separated by ofs,
;; and [f 2] converted to integer:
(awk ((and [f 2] (fconv - - iz) (> [f 2] 5))))

;; print strictly original lines from orec
(awk ((and [f 2] (fconv - - iz) (> [f 2] 5))
      (prn orec)))
```

2. Print every tenth line:

```
(awk ((zerop (mod nr 10))))
```

3. Print any line with a substring matching a regex:

```
(awk (#/(G|D)(2\d[\w]*)/))
```

Note the subtle flaw here: the `[\w]*` portion of the regular expression contributes nothing to what lines are matched. The following example has a similar flaw.

4. Print any line with a substring beginning with a `G` or `D` followed by a sequence of digits and characters:

```
(awk (#/(G|D)([\d\w]*)/))
```

5. Print lines where the second field matches a regex, while the fourth one doesn't:

```
(awk (:let (r #/xyz/))
     ((and [f 3] [r [f 1]] (not [r [f 3]]))))
```

6. Print lines containing a backslash in the second field:

```
(awk ((find #\\ [f 1])))
```

7. Print lines containing a backslash using a regex constructed from a string. Note that backslash escapes are interpreted twice: once in the string literal, and once in the parsing of the regex, requiring four backslashes to encode one:

```
(awk (:let (r (regex-compile "\\\\"))
     ((and [f 1] [r [f 1]]))))
```

8. Print penultimate and ultimate field in each record, separating then by a colon:

```
;; original: {OFS=":";print $(NF-1), $NF}
;;
(awk (t (set ofs ":") (prn [f -2] [f -1])))
```

Note that the above behaves more correctly than the original Awk example because in the when there is only one field, `$(NF-1)` reduces to `$0` which refers to the entire record, not to the field. This sort of bug is why the **TXR Lisp** `awk` does not imitate the design decision to make the record the first numbered field.

9. Output the line number and number of fields separated by colon, by producing a single string first:

```
(awk (t (prn `@nr:@nf`)))
```

10. Print lines longer than 72 characters:

```
(awk ((> (len rec) 72)))
```

11. Print first two fields in reverse order, separated by `ofs`:

```
(awk (t (prn [f 1] [f 0])))
```

12. Same as 11, but with field separation consisting of a comma, or spaces and tabs, or both in sequence:

```
(awk (:set fs #/,[ \t]*|[ \t]+/)
     (t (prn [f 1] [f 0])))
```

13. Add the values in the first column, then print sum and average:

```
;; original:
;; {s += $1}
;; END {print "sum is ", s, " average is", s/NR}
;;
(awk (:let (s 0) (n 0))
     ([f 0] (fconv r) (inc s [f 0]) (inc n))
     (:end (prn `sum is @s average is @(/ s n)`)))
```

Note that the original is not robust against blank lines in the input. Blank lines are treated as if they had a first column field of zero, and are counted toward the denominator in the calculation of the average.

14. Print fields in reverse order, one per line:

```
(awk (t (tprint (reverse f))))
```

15. Print all lines between occurrences of `start` and `stop`:

```
(awk ((rng #/start/ #/stop/)))
```

16. Print lines whose first field is different from the corresponding field in the previous line:

```
(awk (:let prev)
     ((nequal [f 0] prev) (prn) (set prev [f 0])))
```

17. Simulate the `echo` utility:

```
(awk (:begin (prn `@{*args* " "}`)))
```

Note: if this is evaluated in the command line, for instance with the `-e` option, an explicit exit is required to prevent the arguments from being processed by **TXR** after `awk` completes:

```
(awk (:begin (prn `@{*args* " "}`) (exit 0)))
```

18. Print the components of the `PATH` environment variable, one per line:

```
;; Process variable as if it were a file:
(awk (:inputs (make-string-input-stream
                (getenv "PATH")))
     (:set fs ":")
     (t (tprint f)))

;; Just get, split and print; awk macro is irrelevant
(awk (:begin (tprint (split-str (getenv "PATH") ":"))))
```

19. Given a file called `input` which contains page headers of the format `"Page #"` and a **TXR Lisp** file called `prog.tl` which contains:

```
(awk (:let (n (toint n)))
     (#/Page/ (set [f 1] (pinc n)))
     (t))
```

the command line:

```
txr -Dn=5 prog.tl input
```

prints the file, filling in page numbers starting at 5.

9.57 Environment Variables and Command Line

Note that environment variable names, their values, and command-line arguments are all regarded as being externally encoded in UTF-8. **TXR** performs the encoding and decoding automatically.

9.57.1 Special variables `*args-full*`, `*args-eff*` and `*args*`

Description:

The `*args-full*` variable holds the original, complete list of arguments passed from the operating system, including the program executable name.

During command-line-option processing, **TXR** may transform the argument list. The hash-bang mechanism, and the `--args` and `--eargs` options can inject new command-line arguments, as can code which is executed during argument processing via the `-e` options and others.

The `*args-eff*` variable holds the list of *effective arguments*, which is the argument list after these transformations are applied. This variable is established and set to the same value as `*args-full*` prior to command-line processing, but is not updated with its final value until after command-line processing.

The `*args*` variable holds a list of strings representing the remaining arguments which follow any options processed by the **TXR** executable, and the script name. This list is a suffix of `*args-eff*`. Thus, the arguments before `*args*` can be calculated using the expression `(ldiff *args-eff* *args*)`.

The `*args*` variable is available to **TXR Lisp** expressions invoked from the command line via the `-p`, `-e` and other such options. During these evaluations, `*args*` holds all the remaining options, after the invoking option and its argument expression. In other words, code executed from the command line has access to the remaining arguments which follow it. Furthermore, this code may modify the value of `*args*`. Such a modification is visible to the option processing code. That is to say code executed from the command line can rewrite the remaining list of arguments, and that list takes effect.

9.57.2 Function `env`

Syntax:

```
(env)
```

Description:

The `env` function retrieves the list of environment variables. Each variable is represented by a single entry in the list: a string which contains an = (equal) character somewhere, separating the variable name from its value.

Multiple calls to `env` may return the same list, or lists which share structure.

If a list returned by `env` is modified, the behavior is unspecified.

See also: the `env-hash` function.

9.57.3 Function `env-hash`

Syntax:

```
(env-hash)
```

Description:

The `env-hash` function returns an `:equal-based` hash whose keys and values are strings. The hash table is populated with the environment variables, represented as key-value character string pairs.

The `env-hash` function allocates the hash table when it is first invoked; thereafter, it returns the same hash table.

The hash table is updated by the functions `setenv`, `unsetenv` and `getenv`.

Note: calls to the underlying C library functions `setenv` and `getenv`, and other direct

manipulations of the environment, will not update the hash table.

9.57.4 Functions `getenv`, `setenv` and `unsetenv`

Syntax:

```
(getenv name)
(setenv name value [overwrite-p])
(unsetenv name)
```

Description:

These functions provide access to, as well as manipulation of, environment variables. Of these three, `setenv` and `unsetenv` might not be available on some platforms, or `unsetenv` might be be present in a simulated form which sets the variable *name* to the empty string rather than deleting it.

The `getenv` function searches the environment for the environment variable whose name is *name*. If the variable is found, its value is returned. Otherwise `nil` is returned.

The `setenv` function creates or modifies the environment variable indicated by *name*. The *value* string argument specifies the new value for the variable. If *value* is `nil`, then `setenv` behaves like `unsetenv`, except that it observes the *overwrite-p* argument. That is to say, the meaning of a null *value* is that the variable is to be removed.

If the *overwrite-p* argument is specified, and is true, then the variable is overwritten if it already exists. If the argument is false, then the variable is not modified if it already exists. If the argument is not specified, it defaults to the value *t*, effectively giving rise to a two-argument form of `setenv` which creates or overwrites environment variables.

A variable removal is deemed to be an overwrite. Thus if both *value* and *overwrite-p* are `nil`, then `setenv` does nothing.

The `setenv` function unconditionally returns *value* regardless of whether or not it overwrites or removes an existing variable.

The `unsetenv` function removes the environment variable specified by *name*, if it exists. On some platforms, it instead sets the environment variable to the empty string.

Note: supporting removal semantics in `setenv` allows for the following simple save/modify/restore pattern:

```
(let* ((old-val (getenv "SOME-VAR")))
  (unwind-protect
    (progn (setenv "SOME-VAR" new-val)
           ...)
    (setenv "SOME-VAR" old-val)))
```

This works in the case when `SOME-VAR` exists, as well as in the case that it doesn't exist. In both cases, its previous value or, respectively, non-existence, is restored by the `unwind-protect` cleanup form.

These functions interact with the list returned by the `env` function and with the hash table returned by the `env-hash` function as follows.

A previously returned list returned by `env` is not modified. The `setenv` and `unsetenv`

functions may cause a subsequent call to `env` to return a different list. The `getenv` function has no effect on the list.

The hash table previously returned by `env-hash` is modified by `setenv` in the manner consistent with its semantics. A new entry is created in the table, if required, and an existing entry is overwritten only if the `overwrite-p` flag is specified. Likewise, if `setenv` is invoked in a way that causes the environment variable to be deleted, it is removed from the hash also. The `unsetenv` function causes the variable to be removed from the hash table also. The `getenv` function accesses the underlying environment and updates the hash table with the name-value pair which is retrieved.

9.57.5 Function `replace-env`

Syntax:

```
(replace-env env-list)
```

Description:

The `replace-env` function replaces the environment with the environment variables specified in *`env-list`*. The argument is a list of character strings, in the same format as the list returned by the `env` function: each element of the list describes an environment variable as a single character string in which the name is separated by the value by the = character. As a special concession, if this character is missing, the `replace-env` function treats that entry as being a name with an empty value.

The `replace-env` first empties the existing environment, rendering it devoid of environment variables. Then it installs the entries specified in *`env-list`*.

The return value is *`env-list`*.

Note: `replace-env` may be used to specify an exact environment to child programs executed by functions like `open-process`, `sh` or `run`.

Note: the previous environment may be saved by calling `env` and retaining the returned list. Then after modifying the environment, the original environment can be restored by passing that retained list to `replace-env`.

9.57.6 Special variable `*child-env*`

Description:

The `*child-env*` variable specifies the list of command-line variables established for programs executed via the functions `exec`, `run`, `sh`, `open-command` and `open-process`.

The initial top-level value of this variable is the symbol `t` which indicates that `*child-env*` is to be ignored, such that the executed program inherits the current set of environment variables.

If `*child-env*` has any other value, it must be a possibly empty list of environment variables, in the same format as what is returned by `env` function and accepted by `replace-env`. That value completely specifies the environment that executed programs shall receive.

Example:

```
(let ((*child-env* '("a=b")))
  ;; /usr/bin/env sees only "a" environment variable
  (get-lines (open-process "/usr/bin/env" "r")))
```

```
-> ("a=b")
```

9.58 Command-Line-Option Processing

TXR Lisp provides a support for recognizing, extracting and validating the POSIX-style options from a list of command-line arguments.

The supported options can be defined as a list of option descriptor objects each of which is constructed by a call to the `opt` function. Each option can have a long name, a short name, a type, and a description.

The `getopts` function takes a list of option descriptors, and a list of arguments, producing a parse, or else throwing an exception of type `opt-error` if an error is detected. The returned object, an instance of struct type `opts`, can then be queried for specific option values, or for the remaining non-option arguments.

The `opthelp` function takes a list of option descriptors and an output stream, and generates help text on that stream. A program supporting a `--help` option can use this to generate that portion of its help text which describes the available options. Also provided are functions `opthelp-conventions` and `opthelp-types`, which have the same interface as `opthelp` and print additional information. These may be used together with `opthelp` to provide more detailed help under a single `--help` option, or under separate options like `--extra-help`.

The `define-option-struct` macro provides a more streamlined, declarative mechanism built on the same facility. The options are declared in a more condensed way, and using symbols instead of strings. Furthermore, the parsed option values become slot values of an object, named by the same symbols.

9.58.1 Command-Line-Option Conventions

A command-line option can have a short or long name. A short name is always one character long, and treated specially in the command-line syntax. Long options have names two or more characters long. An option can have both a long and short name. Options may not begin with the − (ASCII dash) character. A long option may not contain the = character.

Short options are invoked by specifying an argument with a single leading − followed by the option character. Multiple short options which take no argument can be "clumped": combined into a single argument consisting of a single − followed by multiple short option characters.

An option can take an argument, in which case the argument is required. An option which takes no argument is Boolean, and a Boolean option never takes an argument: "takes no argument" and "Boolean" effectively mean the same thing.

Long options are invoked as an argument which begins with a −− (double dash) immediately followed by the name. When a long option takes an argument, it is mandatory. It must be specified in the same argument, separated from the name by the = character. If that is omitted, then the next command-line argument is taken as the argument. That argument is removed, and not recognized as an option, even if it looks like one.

A Boolean long option can be explicitly specified as false using the `--no-` prefix rather than the −− prefix.

Short options may be invoked using long name syntax; if `a` is a short option, then it may be referenced on the command line as `--a` and treated as a long option in all other ways, including the use of `--no-` to explicitly specify false for a Boolean option.

If a short option takes an argument, it may not clump with other short option. The following command-line argument is taken as the options argument. That argument is removed and is not recognized as an option

even if it looks like one.

If the command-line argument `--` occurs in the command line where an option would otherwise be recognized, it signifies the end of the options. The subsequent arguments are the non-option arguments, even if they resemble options.

9.58.2 Command-Line Processing Examples

The following example illustrates a complete **TXR Lisp** program which parses command-line options:

```
(defvarl options
  (list (opt "v" "verbose" :dec
             "Verbosity level. Higher values produce more chatter.")
        (opt nil "help" :bool
             "List this help text.")
        (opt "x" nil :hex
             "The X factor: a number with a mysterious\ \
              interpretation, affecting the program\ \
              behavior in strange ways.")
        (opt "z" nil) ;; undocumented option
        (opt nil "cee" :cint
             "C style integer.")
        (opt "g" "gravity" :float
             "Gravitational constant. This gives\ \
              the gravitational field\ \
              strength at the Earth's surface.")
        (opt "l" "lit" :str
             "A character string given in TXR Lisp notation.")
        (opt "c" nil 'upcase-str
             "Custom treatment: ARG is converted to uppercase.")
        (opt "b" "bool" :bool
             "A flag you can flip true.")))

(defvarl prog-name *load-path*)

(let ((o (getopts options *args*)))
  (when [o "help"]
    (put-line "Usage:\n")
    (put-line `  @{prog-name} [options] arg*`)
    (opthelp options)
    (exit 0))
  (put-line `args after opts are: @{o.out-args ", "}`))
```

The next example is equivalent to the previous, but using the `define-option-struct` macro:

```
(define-option-struct prog-opts nil
  (v   verbose :dec
               "Verbosity level. Higher values produce more chatter.")
  (nil help    :bool
               "List this help text.")
  (nil extra-help
               :bool
               "List help text with more detailed information.")
  (x   nil     :hex
               "The X factor: a number with a mysterious\ \
```

```
                 interpretation, affecting the program\ \
                 behavior in strange ways.")
  ;; undocumented Boolean:
  (z   nil)
  (nil cee     :cint
               "C style integer.")
  (g   gravity :float
               "Gravitational constant. This gives\ \
                the gravitational field\ \
                strength at the Earth's surface.")
  (l   lit     :str
               "A character string given in TXR Lisp notation.")
  (c   nil     upcase-str
               "Custom treatment: ARG is converted to uppercase.")
  (b   bool    :bool
               "A flag you can flip true."))

(defvarl prog-name *load-path*)

(let ((o (new prog-opts)))
  o.(getopts *args*)
  (when (or o.help o.extra-help)
    (put-line "Usage:\n")
    (put-line `  @{prog-name} [options] arg*`)
    o.(opthelp)
    (when o.extra-help
      o.(opthelp-types)
      o.(opthelp-conventions))
    (exit -1))
  (put-line `args after opts are: @{o.out-args ", "}`))
```

9.58.3 Structure `opt-desc`

Syntax:

```
(defstruct opt-desc
   short long helptext type
   ... unspecified slots)
```

Description:

The `opt-desc` structure describes a single command-line option.

The `short` and `long` slots are either `nil` or else hold strings. The `short` slot gives the option's short name: a one-character-long string which may not be the ASCII dash character `-`. The `long` slot gives the option's long name: a string two or more characters long which doesn't begin with a dash. An option must have at least one of these names.

The `helptext` slot provides a descriptive string. This string may be long. The `opthelp` function displays this text, formatting into multiple lines as necessary. If `helptext` is `nil`, the option is considered undocumented.

The `type` slot may be a symbol naming a global function which takes one argument, or it may be such a function object. Otherwise it must be one of the following keyword symbols:

`:bool` This indicates that the type of the option is Boolean. Such an option doesn't take any argument. Its value is `t` or `nil`.

`:dec` This indicates that the option requires an argument, which is a decimal integer with an optional positive or negative sign. This argument is converted to an integer object.

`:hex` This type indicates that the option requires an argument consisting of a hexadecimal integer with an optional positive or negative sign. This is converted to an integer object.

`:oct` This type indicates that the option requires an argument consisting of a octal integer with an optional positive or negative sign. This is converted to an integer object.

`:cint` This type indicates that the option requires an integer argument whose format conforms to one of three C language conventions in most respects, other than that this integer may have an arbitrary range. All forms may carry an optional positive or negative leading sign at the very beginning. The first convention consists of decimal digits, which must not have a superfluous leading zero. The second convention consists of octal digits which are introduced by an extra leading zero. The third convention consists of hexadecimal digits introduced by the `0x` prefix.

`:float`
This type indicates a decimal floating-point argument, which is converted to a floating-point number. Its basic form is: an optional leading plus or minus sign, followed by a sequence of one or more digits which may contain a single decimal point anywhere, including the very beginning of the sequence or at the end, optionally followed by the letter `e` or `E` followed by a decimal integer which may have a leading positive or negative sign, and include leading zeros.

`:text` This type indicates a simple textual argument. The argument is taken as verbatim UTF-8 text, converted to a string without interpreting the characters in any special way.

`:str` This type indicates that the argument consists of the interior notation of a TXR Lisp character string. It is processed by adding a double quote at the beginning or end, and parsed as a string literal. This parsing must successfully yield a string object, otherwise the argument is ill-formed.

`(list` *`type`*`)`
If the type is specified as a compound form headed by the `list` symbol, it indicates that the command-line option's argument is a list of elements. The argument appears on the command line as a single string contained within one argument. It may contain commas, and is split into pieces using the comma character as a separator. The pieces are then individually treated as of type *`type`* and converted accordingly. The option's argument is then a list object whose elements are the converted pieces. For instance `(list :dec)` will convert a list of comma-separated decimal integer tokens into a list of integer objects. The *`type`* argument must be a basic type other than `:bool`. The `list` option type does not nest.

`(cumul` *`type`*`)`
If the type is specified as a compound form headed by the `cumul` symbol, it indicates that if the option is specified multiple times, the values coming from the multiple occurrences are accumulated into a list. The *`type`* argument must be a `list` type or a basic type other than `:bool`, for example `(cumul (list :dec))` and `(cumul :str)`. This type specifier does not nest: combinations such as `(cumul (cumul ...))` and `(list (cumul ...))` are invalid. The option values are accumulated in reverse order, so that the rightmost repetition becomes the first item in the list. For instance, if the `-x` option has type `(cumul :dec)`, and the arguments presented for parsing are `("-x" "1" "-x" "2")`, then the option's value will be `(2 1)`. If a `list`-typed option is cumulative, then the option value will be a list of lists. Each repetition of the option produces a list, and the lists are accumulated.

If `type` is a function, then the option requires an argument. The argument string is passed to the function, and the value is whatever the function returns.

The `opt-desc` structure may have additional slots which are not specified.

The `opt` convenience function is provided for constructing `opt-desc` objects.

9.58.4 Function `opt`

Syntax:

```
(opt short long [type [helptext]])
```

Description:

The `opt` function provides a slightly condensed syntax for constructing an object of type `opt-desc`.

The required arguments *`short`* and *`long`* are strings, corresponding to `opt-desc` slots of the same name.

The optional parameter *`type`* corresponds to the same-named slot and defaults to `:bool`.

The optional parameter *`helptext`* corresponds to the same-named slot and defaults to `nil` (no help text provided for the option).

The `opt` function follows this equivalence:

```
(opt a b c d)  <-->  (new opt-desc short a long b
                                   type c helptext d)
```

9.58.5 Structure `opts`

Syntax:

```
(defstruct opts nil
   in-args out-args
   ... unspecified slots)
```

Description:

The `opts` structure represents a parsed command line, containing decoded information obtained from the options and an indication of where the non-option arguments start.

The `opts` structure supports direct indexing for option retrieval. That is the only documented interface for accessing the parsed options; the implementation of the information set describing the parsed options is unspecified.

The `in-args` slot holds the original argument list.

The `out-args` slot holds the tail of the argument list consisting of the non-option arguments.

The mechanism by means of which `out-args` is calculated, and by means of which the information about the options is populated, is unspecified. The only interface to that mechanism is the `getopts` function.

The `opts` object supports indexing, including indexed assignment.

If `o` is an instance of `opts` returned by `getopts`, then the expression `[o "v"]` tests whether the option `"v"` is available in `o`; that is, whether it has been specified in the command line. If so, then its associated value is returned, otherwise `nil` is returned. This `nil` is ambiguous: for a Boolean option it indicates that either the option was not specified, or that it was explicitly specified as false. For a Boolean option that was specified (positively), the value `t` is returned.

The expression `[o "v" dfl]` yields the value of option `"v"` if that option has been specified. If the option hasn't been specified, then the expression yields the value `dfl`.

Assigning to `[o "v"]` is possible. This replaces the value associated with option `"v"`. The assignment is erroneous if no such option was parsed from the command line, even if it is a valid option.

If an option is defined with both a long form and a short form, and either form of that option occurs in the command line being processed, then the option appears under both names in the index.

For instance if option `"--verbose"` has the short form `"-v"`, and either option occurs, then both the keys `"v"` and `"verbose"` will exist in the `opts` structure returned by `getopts`. Note that this behavior is different from that of the structure produced `define-option-struct` macro. Under that approach, if an option is defined with a long and short name, the structure will have only a single slot for that option, named after the long name.

9.58.6 Function `getopts`

Syntax:

```
(getopts option-desc-list arg-list)
```

Description:

The `getopts` function takes a list of `opt-desc` structures and a list of strings *`arg-list`* representing command-line arguments.

The *`arg-list`* is parsed. If the parse is unsuccessful, an exception of type `opt-error` is thrown, derived from `error`.

If there are problems in `option-desc-list` itself, then an exception of type `error` is thrown.

If the parse is successful, `getopts` returns an instance of the `opts` structure describing the parsed options and listing the non-option arguments.

9.58.7 Functions `opthelp`, `opthelp-types` and `opthelp-conventions`

Syntax:

```
(opthelp opt-desc-list [stream])
(opthelp-types opt-desc-list [stream])
(opthelp-conventions opt-desc-list [stream])
```

Description:

The `opthelp` function processes the list of `opt-desc` structures *`opt-desc-list`* and compiles a customized body of help describing all of the options which have help text. These are presented in alphabetical order. Options which do not have help text, if any, are simply listed together under a heading which indicates their undocumented status.

The text is formatted to fit within 79 columns, and begins and ends with a blank line. Its format

consists of headings which begin in the first column, and paragraphs and tables which feature a two space left margin. A blank line follows each section heading. The heading begins with a capital letter. Its remaining words are uncapitalized, and it ends with a colon.

The text is sent to *stream*, if specified. This argument defaults to `*stdout*`.

If there are problems in `option-desc-list` itself, then an exception of type `error` is thrown.

The `opthelp-types` supplementary help function processes the *opt-desc-list*, considering only those options which are documented. If any of them have typed arguments, then a legend is printed explaining the types. The legend includes only information about those option argument types which appear in *opt-desc-list*.

The `opthelp-conventions` supplementary help function processes *opt-desc-list*, considering only those options which are documented. It prints a guide to the use of options, which includes information only about the kinds of options actually present in *opt-desc-list*.

9.58.8 Macro `define-option-struct`

Syntax:

```
(define-option-struct name super opt-specifier*)
```

Description:

The `define-option-struct` macro defines a struct type, instances of which provide command-line option parsing.

The *name* and *super* parameters are subject to the same requirements and have the same semantics as the same-named parameters of `defstruct`.

The *opt-specifier* arguments are lists of between two and four elements: `(`*short-symbol long-symbol* `[`*type* `[`*help-text*`]]).` The *short-symbol* and *long-symbol* must be symbols suitable for use as slot names. One of them may be specified as `nil` indicating that the option has no long form, or no short form.

If a *opt-specifier* specifies both a *short-symbol* and a *long-symbol* then only a slot named by *long-symbol* shall exist in the structure.

The struct type defined by `define-option-struct` has four methods: `getopts`, `opthelp`, `opthelp-types` and `opthelp-conventions`. It also has two slots: `in-args` and `out-args`, which function in a manner identical to their same-named counterparts in the `opts` class.

The `getopts` method takes a single argument: the argument list to be processed. When the argument list is successfully processed.

The `opthelp`, `opthelp-types` and `opthelp-conventions` methods take an optional stream argument.

Note: to encode the option names `"t"` or `"nil"`, or option names which clash with the slot names `in-args` and `out-args` or the method names such as `getopts` or `opthelp`, symbols with these names from a package other than `usr` must be used.

9.59 System Programming

9.59.1 Accessor `errno`

Syntax:

```
(errno [new-errno])
(set (errno) new-value)
```

Description:

The `errno` function retrieves the current value of the C library error variable `errno`. If the argument *`new-errno`* is present and is not `nil`, then it specifies a value which is stored into `errno`. The value returned is the prior value.

The place form of `errno` does not take an argument.

9.59.2 Function `strerror`

Syntax:

```
(strerror errno-value)
```

Description:

The `strerror` returns a character string which provides the host platform's description of the integer *`errno-value`* obtained from the `errno` function.

If the host platform fails to provide a description, the function returns `nil`.

9.59.3 Function `exit`

Syntax:

```
(exit [status])
```

Description:

The `exit` function terminates the entire process (running **TXR** image), specifying the termination status to the operating system. Values of the optional *`status`* parameter may be `nil`, `t`, or an integer value. The value `nil` indicates an unsuccessful termination status, whereas `t` indicates a successful termination status. An absence of the *`status`* argument also specifies a successful termination status. If *`status`* is an integer value, it specifies a successful termination if it is `0`, otherwise the interpretation of the value is platform-specific.

9.59.4 Variables `e2big`, `eacces`, `eaddrinuse`, `eaddrnotavail`, `eafnosupport`, `eagain`, `ealready`, `ebadf`, `ebadmsg`, `ebusy`, `ecanceled`, `echild`, `econnaborted`, `econnrefused`, `econnreset`, `edeadlk`, `edestaddrreq`, `edom`, `edquot`, `eexist`, `efault`, `efbig`, `ehostunreach`, `eidrm`, `eilseq`, `einprogress`, `eintr`, `einval`, `eio`, `eisconn`, `eisdir`, `eloop`, `emfile`, `emlink`, `emsgsize`, `emultihop`, `enametoolong`, `enetdown`, `enetreset`, `enetunreach`, `enfile`, `enobufs`, `enodata`, `enodev`, `enoent`, `enoexec`, `enolck`, `enolink`, `enomem`, `enomsg`, `enoprotoopt`, `enospc`, `enosr`, `enostr`, `enosys`, `enotconn`, `enotdir`, `enotempty`, `enotrecoverable`, `enotsock`, `enotsup`, `enotty`, `enxio`, `eopnotsupp`, `eoverflow`, `eownerdead`, `eperm`, `epipe`, `eproto`, `eprotonosupport`, `eprototype`, `erange`, `erofs`, `espipe`, `esrch`, `estale`, `etime`, `etimedout`, `etxtbsy`, `ewouldblock` **and** `exdev`

Description:

These variables correspond to the POSIX "`errno` constants", namely `E2BIG`, `EACCES`, `EADDRINUSE` and so forth. Variables corresponding to all of the `<errno.h>` constants from the Issue 6 2004 edition of POSIX are included. The variables `eownerdead` and `enotrecoverable` from Issue 7 2018 are subject to the availability of the corresponding constants in the host platform.

9.59.5 Function `abort`

Syntax:

```
(abort)
```

Description:

The `abort` function terminates the entire process (running **TXR** image), specifying an abnormal termination status to the process.

Note: `abort` calls the C library function `abort` which works by raising the `SIG_ABRT` signal, known in **TXR** as the `sig-abrt` variable. Abnormal termination of the process is this signal's default action.

9.59.6 Functions `at-exit-call` and `at-exit-do-not-call`

Syntax:

```
(at-exit-call function)
(at-exit-do-not-call function)
```

Description:

The `at-exit-call` function registers *`function`* to be called when the process terminates normally. Multiple functions can be registered, and the same function can be registered more than once. The registered functions are called in reverse order of their registrations.

The `at-exit-do-not-call` function removes all previous `at-exit-call` registrations of *`function`*.

The `at-exit-call` function returns *`function`*.

The `at-exit-do-not-call` function returns `t` if it removed anything, `nil` if no registrations of *`function`* were found.

9.59.7 Function `usleep`

Syntax:

```
(usleep usec)
```

Description:

The `usleep` function suspends the execution of the program for at least *`usec`* microseconds.

The return value is `t` if the sleep was successfully executed. A `nil` value indicates premature wakeup or complete failure.

Note: the actual sleep resolution is not guaranteed, and depends on granularity of the system timer. Actual sleep times may be rounded up to the nearest 10 millisecond multiple on a system where timed suspensions are triggered by a 100 Hz tick.

9.59.8 Functions `mkdir` and `ensure-dir`

Syntax:

```
(mkdir path [mode])
(ensure-dir path [mode])
```

Description:

mkdir tries to create the directory named *path* using the POSIX mkdir function. An exception of type file-error is thrown if the function fails. Returns t on success.

The *mode* argument specifies the request numeric permissions for the newly created directory. If omitted, the requested permissions are #o777 (511): readable and writable to everyone. The requested permissions are subject to the system umask.

The function ensure-dir also creates a directory named *path*. Unlike mkdir, it also attempt to create all the necessary parent directories, and does not throw an error if *path* refers to an existing object, if that object is a directory or a symbolic link to a directory. Rather, in that case it returns nil instead of t.

9.59.9 Function chdir

Syntax:

```
(chdir path)
```

Description:

chdir changes the current working directory to *path*, and returns *t*, or else throws an exception of type file-error.

9.59.10 Function pwd

Syntax:

```
(pwd)
```

Description:

The pwd function retrieves the current working directory. If the underlying getcwd C library function fails with an errno other than ERANGE, an exception will be thrown.

9.59.11 Function rmdir

Syntax:

```
(rmdir path)
```

Description:

The rmdir function removes the directory named by path. If successful, it returns *t*, otherwise it throws an exception of type file-error.

Note: rmdir calls the same-named POSIX function, which requires path to be the name of an empty directory.

9.59.12 Function remove-path

Syntax:

```
(remove-path path [throw-on-error-p])
```

Description:

The remove-path function tries to remove the filesystem object named by *path*, which may be a file, directory or something else.

If successful, it returns t.

The optional Boolean parameter *throw-on-error-p*, which defaults to nil.

A failure to remove the object results in an exception of type file-error being thrown, unless the failure reason is that the object indicated by *path* doesn't exist. In this non-existence case, the behavior is controlled by the *throw-on-error* argument. If that argument is true, the exception is thrown. Otherwise, the function returns normally, producing the value nil to indicate that it didn't perform a removal.

9.59.13 Function rename-path

Syntax:

```
(rename-path from-path to-path)
```

Description:

The rename-path function tries to rename filesystem path *from-path*, which may refer to a file, directory or something else, to the path *to-path*.

If successful, it returns t.

A failure results in an exception of type file-error.

9.59.14 Functions sh and run

Syntax:

```
(sh system-command)
(run program [argument-list])
```

Description:

The sh function executes *system-command* using the system command interpreter. The run function spawns a *program*, searching for it using the system PATH. Using either method, the executed process receives environment variables from the parent.

TXR blocks until the process finishes executing. If the program terminates normally, then its integer exit status is returned. The value zero indicates successful termination.

The return value nil indicates an abnormal termination, or the inability to run the process at all.

In the case of the run function, if the child process is created successfully but the program cannot be executed, then the exit status will be an errno value from the failed exec attempt.

The standard input, output and error file descriptors of an executed command are obtained from the streams stored in the *stdin*, *stdout* and *stderr* special variables, respectively. For a detailed description of the behavior and restrictions, see the open-command function, whose description of this mechanism applies to the run and sh function also.

Note: as of **TXR** 120, the sh function is implemented using run and not by means of the system C library function, as previously. The run function is used to invoke the system interpreter by name. On Unix-like systems, the string /bin/sh is assumed to denote the system interpreter, which is expected to support a pair of arguments -c *command* to specify the command to be executed. On MS Windows, the interpreter is assumed to be the relative pathname cmd.exe and expected to support /C *command* as a way of specifying a command to execute.

9.60 Unix Filesystem Manipulation

9.60.1 Structure `stat`

Syntax:

```
(defstruct stat nil
   dev ino mod nlink uid gid
   rdev size blksize blocks
   atime atime-nsec mtime mtime-nsec
   ctime ctime-nsec path)
```

Description:

The `stat` structure defines the type of object which is returned by the `stat` and `lstat` functions. Except for `path`, `atime-nsec`, `ctime-nsec` and `mtime-nsec`, the slots are the direct counterparts of the members of POSIX C structure `struct stat`. For instance the slot `dev` corresponds to `st_dev`.

The `path` slot is set by the functions `stat` and `lstat`. Its value is `nil` when the path is not available.

The `atime-nsec`, `ctime-nsec` and `mtime-nsec` fields give the fractional parts of `atime`, `ctime` and `mtime`, respectively. They are derived from the newer style information in which the POSIX function provides the timestamps in `struct timespec` format. If that is not available from the platform, these fields take on values of zero.

9.60.2 Functions `stat`, `lstat` and `fstat`

Syntax:

```
(stat {path | stream | fd} [struct])
(lstat path)
(fstat {path | stream | fd} [struct])
```

Description:

The `stat` function retrieves information about a filesystem object whose pathname is given by the string argument *`path`*, or else about a system object associated with the open stream *`stream`*, or one associated with the integer file descriptor *`fd`*.

If a *`stream`* is specified, that stream must be of a kind from which the `fileno` function can retrieve a file descriptor, otherwise an exception of type `file-error` is thrown.

If the object is not found or cannot be accessed, an exception is thrown.

Otherwise, if the *`struct`* argument is missing, information is retrieved and returned, in the form of a new structure of type `stat`. If the *`struct`* argument is present, it must be either: an instance of the `struct` structure type, or of a type derived from that type by inheritance, or else structure type which has all the same slots as the `struct` type. The retrieved information is stored into *`struct`* and that object is returned rather than a new object.

If *`path`* refers to a symbolic link, the `stat` function retrieves information about the target of the link, if it exists, or else throws an exception of type `file-error`.

The `lstat` function behaves the same as `stat` on objects which are not symbolic links. For a symbolic link, it retrieves information about the link itself, rather than its target.

The `path` slot of the returned structure holds a copy of their *`path`* argument value. When

information is retrieved using a *stream* or *fd* argument, this slot is `nil`.

The `fstat` function is an alias for `stat`.

Note: until **TXR** 231, `stat` and `fstat` were distinct functions: `stat` accepted only *path* arguments, whereas `fstat` function accepted only *stream* or *fd* arguments.

9.60.3 Variables `s-ifmt`, `s-iflnk`, `s-ifreg`, `s-ifblk`, ..., `s-ixoth`

Description:

The following variables exist, having integer values. These are bitmasks which can be applied against the value given by the `mode` slot of the `stat` structure returned by the function `stat`: `s-ifmt`, `s-ifsock`, `s-iflnk`, `s-ifreg`, `s-ifblk`, `s-ifdir`, `s-ifchr`, `s-ififo`, `s-isuid`, `s-isgid`, `s-isvtx`, `s-irwxu`, `s-irusr`, `s-iwusr`, `s-ixusr`, `s-irwxg`, `s-irgrp`, `s-iwgrp`, `s-ixgrp`, `s-irwxo`, `s-iroth`, `s-iwoth` and `s-ixoth`.

These variables correspond to the C language constants from POSIX: `S_IFMT`, `S_IFLNK`, `S_IFREG` and so forth.

The `logtest` function can be used to test these against values of mode. For example `(logtest mode s-irgrp)` tests for the group read permission.

9.60.4 Function `umask`

Syntax:

```
(umask [mask])
```

Description:

The `umask` function provides access to the Unix C library function of the same name, which controls which permissions are denied when files are newly created.

If `umask` is called with no argument, it returns the current value of the mask.

If the *mask* argument is present, it must be an integer specifying the new mask to be installed. The previous mask is returned.

If *mask* is absent, then `umask` returns the previous mask.

Note: the value of the *mask* argument may be calculated as a bitwise or of the following constants: `s-irwxu`, `s-irusr`, `s-iwusr`, `s-ixusr`, `s-irwxg`, `s-irgrp`, `s-iwgrp`, `s-ixgrp`, `s-irwxo`, `s-iroth`, `s-iwoth` and `s-ixoth`, which correspond to the POSIX C constants `S_IRWXU`, `S_IRUSR`, `S_IWUSR`, `S_IXUSR`, `S_IRWXG`, `S_IRGRP`, `S_IWGRP`, `S_IXGRP`, `S_IRWXO`, `S_IROTH`, `S_IWOTH` and `S_IXOTH`.

Implementation note: since the `umask` C library function provides no way to retrieve the current mask without overwriting with a new one, the **TXR** `umask` function, when given no argument, simulates the pure retrieval of the mask by calling the C function with an argument of `#o777` to temporarily install the maximally safe mask. The value returned is then reinstated as the mask by another call to `umask`, and that value is also returned.

9.60.5 Functions `makedev`, `minor` and `major`

Syntax:

```
(makedev minor major)
(minor dev)
(major dev)
```

Description:

The parameters *minor*, *major* and *dev* are all integers. The `makedev` function constructs a combined device number from a minor and major pair (by calling the Unix `makedev` function). This device number is suitable as an argument to the `mknod` function (see below). Device numbers also appear as values of the `dev` slot of the `stat` structure.

The `minor` and `major` functions extract the minor and major device number from a combined device number.

9.60.6 Function `chmod`

Syntax:

```
(chmod target mode)
```

Description:

The `chmod` function changes the permissions of the filesystem object specified by *target*. It is implemented in terms of the POSIX functions `chmod` and `fchmod`. If *mode* is a character string representing a symbolic mode, then the function also makes use of `stat` or `fstat` and `umask`.

The permissions are specified by *mode*, which must be an integer or a string.

An integer *mode* is a bitwise combination of permission mode bits. The value is passed directly to the POSIX `chmod` or `fchmod` function. Note: to construct a mode value, applications may use `logior` to combine the values of the variables like `s-irusr` or `s-ixoth` or take advantage of the well-known numeric structure of POSIX permissions to express them octal in octal notation. For instance the mode `#o750` denotes that the owner has read, write and execute permissions, the group owner has read and execute, others have no permission. This value may also be calculated using `(logior s-irwxu s-irgrp s-ixgrp)`.

If the argument to *mode* is a string, it is interpreted according to the symbolic syntax of the POSIX `chmod` utility. For instance, a *mode* value of `"a+w,-s"` means to give all users (owner, group and others) write permission, and remove the setuid and setgid bits.

The full syntax and semantics of symbolic *mode* strings is given in the POSIX standard IEEE 1003.1.

The function throws a `file-error` exception if an error occurs, otherwise it returns `t`.

The *target* argument may be a character string, in which case it specifies a pathname in the filesystem. In this case, the POSIX function `chmod` is invoked.

The *target* argument may also be an integer file descriptor, or a stream. In these two cases, the POSIX `fchmod` function is invoked. For a stream *target*, the integer file descriptor is retrieved from the stream using `fileno` function.

Example:

```
;; Set permissions of foo.txt to "rw-r--r--"
;; (owner can read and write; group owner
```

```
;; and other users can only read).

;; numerically:
(chmod "foo.txt" #o644)

;; symbolically:
(chmod "foo.txt" (logior s-irusr s-iwusr
                         s-irgrp
                         s-iroth))
```

Implementation note: The implementation of the symbolic *mode* processing is based on the descriptions given in IEEE 1003.1-2018, Issue 7 and also on the `chmod` program from from GNU Coreutils 8.28: and experiments with its behavior, and its documentation.

9.60.7 Functions `chown` and `lchown`

Syntax:

```
(chown target id gid)
(lchown target id gid)
```

Description:

The `chown` and `lchown` functions change the user and group ownership of the filesystem object specified by *target*.

They implemented in terms of the POSIX functions `chown`, `fchown` and `lchown`.

The ownership attributes are specified by *uid* and *gid*, both integer arguments.

The existing ownership attributes may be obtained using the `stat` function.

These functions throw a `file-error` exception if an error occurs, otherwise they returns `t`.

The *target* argument may be a character string, in which case it specifies a pathname in the filesystem. In this case, the same-named POSIX function `chown` is invoked by `chown`, whereas `lchown` likewise invokes its respective same-named POSIX counterpart. The difference is that if *target* is a pathname denoting a symbolic link, then `lchown` operates on the symbolic link, whereas `chown` dereferences the symbolic link.

The *target* argument may also be an integer file descriptor, or a stream. In these two cases, the POSIX `fchown` function is invoked by either function. For a stream *target*, the integer file descriptor is retrieved from the stream using `fileno` function.

Note: in most POSIX systems, unprivileged processes may not change the user ownership denoted by *uid*. They may change the group ownership indicated in *gid*, if that value corresponds to the effective group ID of the calling process or one of its ancillary group IDs.

To avoid trying to change the user ownership (and therefore failing), the caller should specify a *uid* value which matches the object's existing owner.

9.60.8 Functions `utimes` and `lutimes`

Syntax:

```
(utimes target atime-s atime-ns mtime-s mtime-ns)
(lutimes target atime-s atime-ns mtime-s mtime-ns)
```

Description:

The functions `utimes` and `lutimes` change the access and modification timestamps of a file indicated by the *`target`* argument.

The difference between the two functions is that if *`target`* is the pathname of a symbolic link, then `lutimes` operates on the symbolic link itself, whereas `utimes` resolves the symbolic link.

Note: the full, complete functionality of these functions requires the platform to provide the POSIX functions `futimens` and `utimensat` functions. If these functions are not available, then other functions are relied on, with some reductions in functionality, that are documented below.

The *`target`* argument specifies the file to operate on. It may be an integer file descriptor, an open stream, or a character string representing a pathname.

The *`atime-s`* and *`mtime-s`* parameters specify the whole seconds part of the new access and modification times, expressed as seconds since the epoch.

The *`atime-ns`* and *`mtime-ns`* parameters specify the fractional part of the access and modification times, expressed in nanoseconds. If an integer argument is given to these parameters, it must lie in the range 0 to 999999999, or else the symbols `nil` or `t` may be passed as arguments.

If the symbol `nil` is passed as the nanoseconds part of the access or modification time, then the access or modification time, respectively, shall not be modified by the operation. The corresponding seconds argument is ignored.

If the symbol `t` is passed as the nanoseconds part of the access or modification time, then the access or modification time, respectively, shall be obtained from the current system time. The corresponding seconds argument is ignored.

If the `utimensat` and `futimens` functions are not available from the host system, then the above `nil` and `t` convention in the nanoseconds arguments is not supported; the function will fail by throwing an exception if an attempt is made to pass these arguments.

If the `utimensat` and `futimens` functions are not available from the host system, then operating on a symbolic link with `lutimes` is only possible if the system provides the `lutimes` C library function, otherwise the operation fails by throwing an exception (if given a path argument for *`target`*, even if that path isn't a symbolic link).

If the implementation falls back on the `utimes`, `futimes`, and `lutimes` functions, then the nanoseconds arguments are truncated to microsecond precision.

If the implementation falls back on `utime`, then the nanoseconds arguments are ignored; the times are effectively truncated to whole seconds.

9.60.9 Function `mknod`

Syntax:

```
(mknod path mode [dev])
```

Description:

The `mknod` function tries to create an entry in the filesystem: a file, FIFO, or a device special file, under the name *`path`*. If it is successful, it returns `t`, otherwise it throws an exception of type `file-error`.

The *mode* argument is a bitwise or combination of the requested permissions, and the type of object to create: one of the constants `s-ifreg`, `s-ififo`, `s-ifchr`, `s-ifblk` or `s-ifsock`. The permissions are subject to the system `umask`.

If a block or character special device (`s-ifchr` or `s-ifblk`) is being created, then the *dev* argument specifies the major and minor numbers of the device. A suitable value can be constructed from a major and minor pair using the `makedev` function.

Example:

```
;; make a character device (8, 3) called /dev/foo
;; requesting rwx------ permissions

(mknod "dev/foo" (logior #o700 s-ifchr) (makedev 8 3))
```

9.60.10 Function `mkfifo`

Syntax:

```
(mkfifo path mode)
```

Description:

The `mkfifo` function creates a POSIX FIFO object. If it is successful, it returns `t`, otherwise it throws an exception of type `file-error`.

The *mode* argument is a bitwise or combination of the requested permissions, and is subject to the system `umask`.

Note: the `mknod` function can also create FIFOs, specified via the bitwise combination of the `s-ififo` type and the permission mode bits.

9.60.11 Functions `symlink` and `link`

Syntax:

```
(symlink target path)
(link target path)
```

Description:

The `symlink` function creates a symbolic link called *path* whose contents are the absolute or relative path *target*. *target* does not actually have to exist.

The link function creates a hard link. The object at *target* is installed into the filesystem at *path* also.

If these functions succeed, they return `t`. Otherwise they throw an exception of type `file-error`.

9.60.12 Function `readlink`

Syntax:

```
(readlink path)
```

Description:

If *path* names a filesystem object which is a symbolic link, the `readlink` function reads the contents of that symbolic link and returns it as a string. Otherwise, it fails by throwing an

exception of type `file-error`.

9.60.13 Function `realpath`

Syntax:

```
(realpath path)
```

Description:

The `realpath` function provides access to the same-named POSIX function. It processes the input string *`path`* by expanding all symbolic links, removes all superfluous `".."` and `"."` path components, and extra component-separating slash characters, to produce a canonical absolute pathname.

If the underlying POSIX function indicates failure, then `nil` is returned. In that situation the `errno` value is available using the `errno` function.

9.61 Unix Filesystem Complex Operations

Functions in this category are complex functionality implemented using a combination of multiple calls into the host system's POSIX API.

9.61.1 Functions `copy-file` and `copy-files`

Syntax:

```
(copy-file from-path to-path [perms-p [times-p]])
(copy-file from-list to-dir [perms-p [times-p]])
```

Description:

The `copy-file` function creates a replica of the file `from-path` at the destination path *`to-path`*.

Both paths are opened using `open-file` in binary mode, as if using `(open-file from-path "b")` and `(open-file to-path "wb")` respectively. Then bytes are read from one stream and written to the other, in blocks which whose size is a power of two at least as large as 16834.

If the optional Boolean parameter *`perms-p`* is specified, and is true, then the permissions of *`from-path`* are propagated to *`to-path`*.

If the optional Boolean parameter *`times-p`* is specified, and is true, then the access and modification timestamps of *`from-path`* are propagated to *`to-path`*.

The `copy-file` function returns `nil` if it is successful, and throws an exception derived from `file-error` on failure.

The `copy-files` function copies multiple files, whose pathnames are given by the list argument *`from-list`* into the target directory whose path is given by *`to-dir`*.

The target directory must exist.

For each source path in *`from-list`*, the `copy-files` function forms a target path by combining the base name of the source path with *`target-dir`*. (See the `base-name` and `path-cat` functions). Then, the source path is copied to the resulting target path, as if by the `copy-file` function.

The `copy-files` function returns `nil` if it is successful, and throws an exception derived from `file-error` on failure.

Additionally, `copy-files` provides an internal catch for the `retry` and `skip` restart exceptions. If the caller, using a handler frame established by `handle`, catches an error emanating from the `copy-files` function, it can retry the failed operation by throwing the `retry` exception, or continue copying with the next file by throwing the `skip` exception.

Example:

```
;; Copy all "/mnt/cdrom/*.jpg" files into "images" directory,
;; preserving their time stamps,
;; continuing the operation in the face of
;; file-error exceptions.
(handle
  (copy-files (glob "/mnt/cdrom/*.jpg") "images" nil t)
  (file-error (throw 'skip)))
```

9.61.2 Function `cat-files`

Syntax:

```
(cat-files to-path from-path*)
```

Description:

The `cat-files` function catenates the contents of zero or more files into one file. The destination path is specified by *`to-path`*. Regardless of whether there are any *`from-path`* arguments, the file named by *`to-path`* is created, if necessary or else truncated to zero length. Then, the files named by each *`from-path`* are traversed in left-to-right order; the contents of each file is appended to the destination file.

9.61.3 Function `copy-path-rec`

Syntax:

```
(copy-path-rec from-path to-path option*)
```

Description:

The `copy-path-rec` function replicates a file system object identified by the pathname *`from-path`*, creating a similar object named *`to-path`*.

If `from-path` is a directory, it is recursively traversed and its structure and content is replicated under `to-path`.

The *`option`* arguments are keywords, which may be the following:

:perms Propagate the permissions of all objects under *`from-path`* onto their *`to-path`* counterparts. In the absence of this option, the copied objects receive the same permissions as a newly created files. On POSIX systems this means: readable and writable to the owner, group and others, by default, subject to the `umask` that is in effect.

:times Propagate the modification and access time stamps of all objects under *`from-path`* onto their *`to-path`* counterparts.

:symlinks
Copy symbolic links literally rather than dereferencing them. Symbolic links are not altered in any way; their exact content is preserved. Thus, relative symlinks which point outside of the *`from-path`* tree may turn into dangling symlinks in the *`to-path`* tree.

:owner Propagate the ownership of all objects under *from-path* to their *to-path* counterparts. Ownership refers to the owner user ID and group ID. Without this option, the ownership of the copied objects is derived from the effective user ID and group ID of the calling process. Note that it is assumed that the host system may requires superuser privileges to set both ownerships IDs of an object, and to set them to an arbitrary value. An unprivileged process may not change the user ID of a file, and may only change the group ID of a file which they own, to one of the groups of which that process is a member, either via the effective GID, or the ancillary list. The `copy-path-rec` function tests whether the application is running under superuser privileges; if not, then it only honors the `:owner` option for those objects under *from-path* which are owned by the caller, and owned by a group to which the caller belongs. Other objects are copied as if the `:owner` option were not in effect, avoiding an attempt to set their ownership that is likely to fail.

:all The `:all` keyword is a shorthand representing all of the options being applied: permissions, times, symlinks and ownership are replicated.

The `copy-path-rec` function creates all necessary pathname components required for *to-path* to come into existence, as if by using the `ensure-dir` function.

Whenever an object under *from-path* has a counterpart in *to-path* which already exists, the situation is handled as follows:

1. If a directory object is copied to an existing directory object, then that existing directory object is accepted as the copy, and the operation continues recursively within that directory. If any options are specified, then the requested attributes are propagated to that existing directory.
2. If a non-directory object is copied to a directory object, the situation throws an exception: the `copy-path-rec` function refuses to delete an entire directory or subdirectory in order to make way for a file, symbolic link, special device or any other kind of non-directory object.
3. If any object is copied to an existing non-directory object, that target object is removed first, then the copy operation proceeds.

Copying of files takes place similarly as what is described for the `copy-file` function.

Special objects such as FIFOs, character devices, block devices and sockets are copied by creating a new, similar objects at the destination path. In the case of devices, the major and minor numbers of the copy are derived from the original, so that the copy refers to the same device. However, the copy of a socket or a FIFO is effectively a new, different endpoint because these objects are identified by their pathname. Processes using the copy of a socket or a FIFO will not connect to processes which are working with the original.

The `copy-path-rec` function returns `nil` if it is successful. It throws an exception derived from `file-error` when encountering failures.

Additionally `copy-path-rec` provides an internal catch for the `retry` and `skip` restart exceptions. If the caller, using a handler frame established by `handle`, catches an error emanating from the `copy-files` function, it can retry the failed operation by throwing the `retry` exception, or continue copying with the next object by throwing the `skip` exception.

9.61.4 Function `remove-path-rec`

Syntax:

```
(remove-path-rec path)
```

Description:

The `remove-path-rec` function attempts to remove the filesystem object named by *path*. If *path* refers to a directory, that directory is recursively traversed to remove all of its contents, and is then removed.

The `remove-path-rec` function returns `nil` if it is successful. It throws an exception derived from `file-error` when encountering failures.

Additionally `remove-path-rec` provides an internal catch for the `retry` and `skip` restart exceptions. If the caller, using a handler frame established by `handle`, catches an error emanating from the `copy-files` function, it can retry the failed operation by throwing the `retry` exception, or continue removing other objects by throwing the `skip` exception. Skipping a failed remove operation may cause subsequent operations to fail. Notably, the failure to remove an item inside a directory means that removal of that directory itself will fail, and ultimately, *path* will still exist when `remove-path-rec` completes and returns.

9.61.5 Functions `chmod-rec` and `chown-rec`

Syntax:

```
(chmod-rec path mode)
(chown-rec path uid gid)
```

Description:

The `chmod-rec` and `chown-rec` functions are recursive counterparts of `chmod` and `lchown`.

The filesystem object given by *path* is recursively traversed, and each of its constituent objects is subject to a permission change in the case of `chown-rec`, or an ownership change in the case of `chown-rec`.

The `chmod-rec` function alters the permission of each object that is not a symbolic link using the `chmod` function, and *mode* is interpreted accordingly: it may be an integer or string. Each object which is a symbolic link is ignored.

The `chown-rec` function alters the permission of each object encountered, including symbolic links, using the `lchown` function.

These functions establish restart catches, similarly to `remove-path-rec` and `copy-path-rec`, allowing the caller to retry individual failed operations or skip the objects on which operations have failed.

9.61.6 Function `touch`

Syntax:

```
(touch path [ref-path])
```

Description:

The `touch` function updates the modification timestamp of the filesystem object named by *path*. If the object doesn't exist, it is created as a regular file.

If *ref-path* is specified, then the modification timestamp of the object denoted by *path* is updated to be equivalent to the modification timestamp of the object denoted by *ref-path*. Otherwise *ref-path* being absent, the modification timestamp of *path* is set to the current time.

If *path* is a symbolic link, it is dereferenced; touch operates on the target of the link.

9.61.7 Function mkdtemp

Syntax:

```
(mkdtemp prefix)
```

Description:

The mkdtemp function combines the *prefix*, which is a string, with a generated suffix to create a unique directory name. The directory is created, and the name is returned.

If the prefix argument ends in with a sequence of one or more X characters, the behavior is unspecified.

Note: this function is implemented using the same-named POSIX function. Whereas the POSIX function requires the template to end in a sequence of at least six X characters, which are replaced by the generated suffix, the **TXR Lisp** function handles this detail internally, requiring only the prefix part without those characters.

9.61.8 Function mkstemp

Syntax:

```
(mkstemp prefix [suffix])
```

Description:

The mkstemp function create a unique file name by adding a generated infix between the *prefix* and *suffix* strings. The file is created, and a stream open in "w+b" mode for the file is returned.

If either the *prefix* or *suffix* contain X characters, the behavior is unspecified.

If *suffix* is omitted, it defaults to the empty string.

The name of the file is available by interrogating the returned stream's :name property using the function stream-get-prop.

Notes: this function is implemented using the POSIX function mkstemp or, if available, using the mkstemps function which is not standardized, but appears in the GNU C Library and some other systems. If mkstemps is unavailable, then the suffix functionality is not available: the *suffix* argument must either be omitted, or must be an empty string.

Whereas the C library functions require the template to contain a sequence at least six X characters, which are replaced by the generated portion, the **TXR Lisp** function handles this detail internally, requiring no such characters in any of its inputs.

9.62 Unix Filesystem Object Existence, Type and Access Tests

Functions in this category perform various tests on the attributes of filesystem objects.

The functions all have a *path* parameter, which accepts three types of arguments. If a character string is specified, it denotes a filesystem path to be probed for properties such as ownership and permissions. The object is probed using the stat function except in the case of path-symlink-p which uses lstat. If instead a stream is specified as *path*, then the associated filesystem descriptor is probed for these properties. If an integer value is specified, it is treated as a POSIX open file descriptor that is to be probed.

Otherwise, a `stat` structure, for example one returned by the `stat` or `lstat` function may be specified, in which case no system object is probed. The properties to be tested are those given in the `stat` object.

Note: in a situation when it is necessary to use any of these functions to probe the properties of a symbolic link itself (other than the function `path-symlink-p` which does so implicitly) it is necessary to first invoke `lstat` on the symlink's path, and then pass the resulting `stat` structure to that function instead of the path.

Some of the accessibility tests (functions which determine whether the calling process has certain access rights) may not be perfectly accurate, since they are based strictly on portable information available via `stat`, together with the basic, portable POSIX APIs for inquiring about security credentials, such as `getuid`. They ignoring any special permissions which may exist such as operating system and file system specific extended attributes (for example, file immutability connected to a "secure level" and such) and special process capabilities not reflected in the basic credentials.

With the exception of two functions, the accessibility tests use the real credentials of the caller, rather than the effective credentials. Thus, in a setuid process, where the real and effective privileges are different, the access tests inquire about whether the real user has the given access, not the effective user. In this aspect, the functions are similar to the POSIX `access` function which also uses real credentials. The functions `path-private-to-me-p` and `path-strictly-private-to-me-p` use effective credentials, because they answer a different question: can the given filesystem object be trusted? The trust has to be determined from the point of view of the effective user, because security-sensitive actions are being performed in their context; and the effective user does not trust the real user.

9.62.1 Function `path-exists-p`

Syntax:

```
(path-exists-p path)
```

Description:

The `path-exists-p` function returns `t` if *`path`* is a string which resolves to a filesystem object. Otherwise it returns `nil`. If the *`path`* names a dangling symbolic link, it is considered nonexistent.

If *`path`* is an object returned by `stat` or `lstat`, `path-exists-p` unconditionally returns `t`.

9.62.2 Functions `path-file-p`, `path-dir-p`, `path-symlink-p`, `path-blkdev-p`, `path-chrdev-p`, `path-sock-p` and `path-pipe-p`

Syntax:

```
(path-file-p path)
(path-dir-p path)
(path-symlink-p path)
(path-blkdev-p path)
(path-chrdev-p path)
(path-sock-p path)
(path-pipe-p path)
```

Description:

`path-file-p` tests whether *`path`* exists and is a regular file.

`path-dir-p` tests whether *`path`* exists and is a directory.

`path-symlink-p` tests whether *`path`* exists and is a symbolic link.

Similarly, `path-blkdev-p` tests for a block device, `path-chrdev-p` for a character device, `path-sock-p` for a socket and `path-pipe-p` for a named pipe.

9.62.3 Function `path-dir-empty`

Syntax:

```
(path-dir-empty path)
```

Description:

The `path-dir-empty` function returns `t` if *`path`* is an empty directory.

Implementation note: this function performs a test similar to `path-dir-p`; then, if it is confirmed that *`path`* is a directory, a directory stream is opened and entries are read. If an entry is seen which has a name other than `"."` or `".."` then it is concluded that the directory is not empty and `nil` is returned. If no such entry is seen, then the directory is deemed empty and `t` is returned.

9.62.4 Functions `path-setgid-p`, `path-setuid-p` and `path-sticky-p`

Syntax:

```
(path-setgid-p path)
(path-setuid-p path)
(path-sticky-p path)
```

Description:

`path-setgid-p` tests whether *`path`* exists and has the set-group-ID permission set.

`path-setuid-p` tests whether *`path`* exists and has the set-user-ID permission set.

`path-sticky-p` tests whether *`path`* exists and has the "sticky" permission bit set.

9.62.5 Functions `path-mine-p` and `path-my-group-p`

Syntax:

```
(path-mine-p path)
(path-my-group-p path)
```

Description:

`path-mine-p` tests whether *`path`* exists, and is effectively owned by the calling process; that is, it has a user ID equal to the real user ID of the process.

`path-my-group-p` tests whether *`path`* exists, and is effectively owned by a group to which the calling process belongs. This means that the group owner is either the same as the real group ID of the calling process, or else is among the supplementary group IDs of the calling process.

9.62.6 Function `path-readable-to-me-p`

Syntax:

```
(path-readable-to-me-p path)
```

Description:

`path-readable-to-me-p` tests whether the calling process can read the object named by *`path`*. If necessary, this test examines the real user ID of the calling process, the real group ID,

and the list of supplementary groups.

9.62.7 Function `path-writable-to-me-p`

Syntax:

```
(path-writable-to-me-p path)
```

Description:

`path-writable-to-me-p` tests whether the calling process can write the object named by *path*. If necessary, this test examines the real user ID of the calling process, the real group ID, and the list of supplementary groups.

9.62.8 Function `path-read-writable-to-me-p`

Syntax:

```
(path-read-writable-to-me-p path)
```

Description:

`path-readable-to-me-p` tests whether the calling process can both read and write the object named by *path*. If necessary, this test examines the real user ID of the calling process, the real group ID, and the list of supplementary groups.

9.62.9 Function `path-executable-to-me-p`

Syntax:

```
(path-executable-to-me-p path)
```

Description:

`path-executable-to-me-p` tests whether the calling process can execute the object named by *path*, or perform a search (name lookup, not implying sequential readability) on it, if it is a directory. If necessary, this test examines the real user ID of the calling process, the real group ID, and the list of supplementary groups.

9.62.10 Functions `path-private-to-me-p` and `path-strictly-private-to-me-p`

Syntax:

```
(path-private-to-me-p path)
(path-strictly-private-to-me-p path)
```

Description:

The `path-private-to-me-p` and `path-strictly-private-to-me-p` functions report whether the calling process can rely on the object indicated by `path` to be, respectively, private or strictly private to the security context implied by its effective user ID.

"Private" means that beside the effective user ID of the calling process and the superuser, no other user ID has write access to the object, and thus its contents may be trusted to be be free from tampering by any other user.

"Strictly private" means that not only is the object private, as above, but users other than the effective user ID of the calling process and superuser also not not have read access.

The rules which the function applies are as follows:

A file to be examined is initially assumed to be strictly private.

If the file is not owned by the effective user ID of the caller, or else by the superuser, then it is not private.

If the file grants write permission to "others", then it is not private.

If the file grants read permission to "others", then it is not strictly private.

If the file grants write permission to the group owner, then it is not private if the group contains names other than that of the file owner or the superuser.

If the file grants read permission to the group owner, then it is not strictly private if the group contains names other than that of the file owner or the superuser.

Note that this interpretation of "private" and "strictly private" is vulnerable to the following time-of-check to time-of-use race condition with regard to the group check. At the time of the check, the group might be empty or contain only the caller as a member. But by the time the file is subsequently accessed, the group might have been innocently extended by the system administrator to include additional users, who can maliciously modify the file.

Another issue is that if any components of *path* can be subverted by another user, test may not be trusted. It becomes vulnerable to a time-of-check to time-of-use race condition.

The function `path-components-safe` function is provided to perform a security check on an entire path.

9.62.11 Function `path-components-safe`

Syntax:

```
(path-components-safe path)
```

Description:

On Unix platforms, the `path-components-safe` performs a security check on an entire relative or absolute *path*, returning `t` if the entire path is examined without encountering an error, and the check passes, otherwise `nil`. On native Microsoft Windows, the function unconditionally returns true.

An exception may be thrown if an an inaccessible or nonexistent path component is encountered, too many symbolic links have to be resolved or there is some other problem preventing the traversal of *path*.

The objective of this function is to determine that every portion of `path` is writable only to the effective user: that if the path is used for filesystem access, its meaning cannot be altered by an adversarial user who is able to control a symbolic link or a directory component.

The function expands symbolic links on its own, one level at a time, and walks the components coming from a link target.

Note: directories which are owned by root, and have the sticky bit, as is the usual configuration of `tmp` are considered safe, even though multiple users have write permissions.

9.62.12 Functions `path-newer` and `path-older`

Syntax:

```
(path-newer left-path right-path)
(path-older left-path right-path)
```

Description:

The `path-newer` function compares two paths or stat results by modification time. It returns `t` if *`left-path`* exists, and either *`right-path`* does not exist, or has a modification time stamp in the past relative to *`left-path`*.

The `path-older` function is equivalent to `path-newer` with the arguments reversed.

Note: `path-newer` takes advantage of subsecond timestamp resolution information, if available. The implementation is based on using the `mtime-nsec` field of the `stat` structure, if it isn't `nil`.

9.62.13 Function `path-same-object`

Syntax:

```
(path-same-object left-path right-path)
```

Description:

The `path-same-object` function returns `t` if *`left-path`* and *`right-path`* resolve to the same filesystem object: the same inode number on the same device.

9.62.14 Function `path-search`

Syntax:

```
(path-search name [search-path])
```

Description:

The `path-search` function searches for the existence of a filesystem object named by *`name`* in the directories specified *`search-path`*.

If *`name`* is the empty string or one of the two strings `"."` (dot) or `".."` (dotdot), then `nil` is returned. If *`name`* contains any path separator characters (any of the set of characters found in the `path-sep-chars` string) then the function returns *`name`* without performing any search. In all these trivial cases, the *`search-path`* argument is ignored.

The *`search-path`* argument, if present, may be a string or a list of strings. If omitted, then it takes on the value of the `PATH` environment variable if that variable exists, or else takes on the value `nil` indicating an empty search path.

If *`search-path`* is a string, it is converted to a list of directories by splitting on the separator character, which may be `:` (colon) or `;` (semicolon) depending on the system. Then, for each directory in the list, `path-search` affixes the *`name`* to that component, as if using the `path-cat` function, and tests whether the resulting path refers to an existing filesystem object. If so, then the search terminates and that resulting path is returned. If the entire list is traversed without finding a filesystem object, then `nil` is returned. If any error whatsoever occurs while determining whether the resulting path exists, the situation is treated as nonexistence, and the search continues.

Note: subtle discrepancies may exist between `path-search` and the host platform's

mechanisms for searching for an executable program. For instance, since `path-search` is interested in existence only, it may return a path which exists, but is not executable. Whereas a path searching implementation which tests for executability will in that case continue searching, and not return that path.

9.63 Unix Credentials

9.63.1 Functions `getuid`, `geteuid`, `getgid` and `getegid`

Syntax:

```
(getuid)
(geteuid)
(getgid)
(getegid)
```

Description:

These functions directly correspond to the POSIX C library functions of the same name. They retrieve the real user ID, effective user ID, real group ID and effective group ID, respectively, of the calling process.

9.63.2 Functions `setuid`, `seteuid`, `setgid` and `setegid`

Syntax:

```
(setuid uid)
(seteuid uid)
(setgid gid)
(setegid gid)
```

Description:

These functions directly correspond to the POSIX C library functions of the same name. They set the real user ID, effective user ID, real group ID and effective group ID, respectively, of the calling process. On success, they return `t`. On failure, they throw an exception of type `system-error`.

9.63.3 Function `getgroups`

Syntax:

```
(getgroups)
```

Description:

The `getgroups` function retrieves the list of supplementary group IDs of the calling process by calling the same-named POSIX C library function.

Whether or not the effective group ID retrieved by `getegid` is included in this list is system-dependent. Programs should not depend on its presence or absence.

9.63.4 Function `setgroups`

Syntax:

```
(setgroups gid-list)
```

Description:

The `setgroups` function corresponds to a C library function found in some Unix operating systems, complementary to the `getgroups` function. The argument to *`gid-list`* must be a list of numeric group IDs. If the function is successful, this list is installed as the list of supplementary

group IDs of the calling process, and the value `t` is returned. On failure, it throws an exception of type `system-error`.

9.63.5 Functions `getresuid` and `getresgid`

Syntax:

```
(getresuid)
(getresgid)
```

Description:

These functions directly correspond to the POSIX C library functions of the same names available in some Unix operating systems. Each function retrieves a three element list of numeric IDs. The `getresuid` function retrieves the real, effective and saved user ID of the calling process. The `getresgid` function retrieves the real, effective and saved group ID of the calling process.

9.63.6 Functions `setresuid` and `setresgid`

Syntax:

```
(setresuid real-uid effective-uid saved-uid)
(setresgid real-gid effective-gid saved-gid)
```

Description:

These functions directly correspond to the POSIX C library functions of the same names available in some Unix operating systems. They change the real, effective and saved user ID or group ID, respectively, of the calling process.

A value of -1 for any of the IDs specifies that the ID is not to be changed.

Only privileged processes may arbitrarily change IDs to different values.

Unprivileged processes are restricted in the following way: each of the new IDs that is replaced must have a new value which is equal to one of the existing three IDs.

9.64 Unix Password Database

9.64.1 Structure `passwd`

Syntax:

```
(defstruct passwd nil
   name passwd uid gid
   gecos dir shell)
```

Description:

The `passwd` structure corresponds to the C type `struct passwd`. Objects of this struct are produced by the password database query functions `getpwent`, `getpwuid`, and `getpwnam`.

9.64.2 Functions `getpwent`, `setpwent` and `endpwent`

Syntax:

```
(getpwent)
(setpwent)
(endpwent)
```

Description:

The first time `getpwent` function is called, it returns the first password database entry. On subsequent calls it returns successive entries. Entries are returned as instances of the `passwd` structure. If the function cannot retrieve an entry for any reason, it returns `nil`.

The `setpwent` function rewinds the database scan.

The `endpwent` function releases the resources associated with the scan.

9.64.3 Function `getpwuid`

Syntax:

```
(getpwuid uid)
```

Description:

The `getpwuid` searches the password database for an entry whose user ID field is equal to the numeric *`uid`*. If the search is successful, then a `passwd` structure representing the database entry is returned. If the search fails, `nil` is returned.

9.64.4 Function `getpwnam`

Syntax:

```
(getpwnam name)
```

Description:

The `getpwnam` searches the password database for an entry whose user name is equal to *`name`*. If the search is successful, then a `passwd` structure representing the database entry is returned. If the search fails, `nil` is returned.

9.65 Unix Group Database

9.65.1 Structure `group`

Syntax:

```
(defstruct group nil
   name passwd gid mem)
```

Description:

The `group` structure corresponds to the C type `struct group`. Objects of this struct are produced by the password database query functions `getgrent`, `getgrgid`, and `getgrnam`.

9.65.2 Functions `getgrent`, `setgrent` and `endgrent`

Syntax:

```
(getgrent)
(setgrent)
(endgrent)
```

Description:

The first time `getgrent` function is called, it returns the first group database entry. On subsequent calls it returns successive entries. Entries are returned as instances of the `passwd` structure. If the function cannot retrieve an entry for any reason, it returns `nil`.

The `setgrent` function rewinds the database scan.

The `endgrent` function releases the resources associated with the scan.

9.65.3 Function `getgrgid`

Syntax:

```
(getgrgid gid)
```

Description:

The `getgrgid` searches the group database for an entry whose group ID field is equal to the numeric *gid*. If the search is successful, then a `group` structure representing the database entry is returned. If the search fails, `nil` is returned.

9.65.4 Function `getgrnam`

Syntax:

```
(getgrnam name)
```

Description:

The `getgrnam` searches the group database for an entry whose group name is equal to *name*. If the search is successful, then a `group` structure representing the database entry is returned. If the search fails, `nil` is returned.

9.66 Unix Password Hashing

9.66.1 Function `crypt`

Syntax:

```
(crypt key salt)
```

Description:

The `crypt` function is a wrapper for the Unix C library function of the same name. It calculates a hash over the *key* and *salt* arguments, which are strings. The hash is returned as a string.

The *key* and *salt* arguments are converted into UTF-8 prior to being passed into the underlying platform function. The hash value is assumed to be UTF-8 and converted to Unicode characters, though it is not expected to contain anything but 7 bit ASCII characters.

Note: if C library function `crypt` uses a static buffer for its return value. If that function is used, the Lisp string returned by the **TXR Lisp** function carries its own copy of that buffer. Where available, the `crypt_r` function is used which avoids static storage.

9.67 Unix Signal Handling

On platforms where certain advanced features of POSIX signal handling are available at the C API level, **TXR** exposes signal-handling functionality.

A **TXR** program can install a **TXR Lisp** function (such as an anonymous `lambda`, or the function object associated with a named function) as the handler for a signal.

When that signal is delivered, **TXR** will intercept it with its own safe, internal handler, mark the signal as deferred (in a **TXR** sense) and then dispatch the registered function at a convenient time.

Handlers currently are not permitted to interrupt the execution of most **TXR** internal code. Immediate, asynchronous execution of handlers is currently enabled only while **TXR** is blocked on I/O operations or sleeping. Additionally, the `sig-check` function can be used to dispatch and clear deferred signals. These handlers are then safely called if they were subroutines of `sig-check`, and not asynchronous interrupts.

9.67.1 Variables `sig-hup`, `sig-int`, `sig-quit`, `sig-ill`, `sig-trap`, `sig-abrt`, `sig-bus`, `sig-fpe`, `sig-kill`, `sig-usr1`, `sig-segv`, `sig-usr2`, `sig-pipe`, `sig-alrm`, `sig-term`, `sig-chld`, `sig-cont`, `sig-stop`, `sig-tstp`, `sig-ttin`, `sig-ttou`, `sig-urg`, `sig-xcpu`, `sig-xfsz`, `sig-vtalrm`, `sig-prof`, `sig-poll`, `sig-sys`, `sig-winch`, `sig-iot`, `sig-stk-flt`, `sig-io`, `sig-lost` and `sig-pwr`

Description:

These variables correspond to the C signal constants `SIGHUP`, `SIGINT` and so forth. The variables `sig-winch`, `sig-iot`, `sig-stkflt`, `sig-io`, `sig-lost` and `sig-pwr` may not be available since a system may lack the corresponding signal constants. See notes for the function `log-authpriv`.

The highest signal number is 31.

9.67.2 Functions `set-sig-handler` and `get-sig-handler`

Syntax:

```
(set-sig-handler signal-number handling-spec)
(get-sig-handler signal-number)
```

Description:

The `set-sig-handler` function is used to specify the handling for a signal, such as the installation of a handler function. It updates the signal handling for a signal whose number is *signal-number* (usually one of the constants like `sig-hup`, `sig-int` and so forth), and returns the previous value. The `get-sig-handler` function returns the current value.

The *signal-number* must be an integer the range 1 to 31.

Initially, all 31 signal handling specifications are set to the value `t`.

The *handling-spec* parameter may be a function. If a function is specified, then the signal is enabled and connected to that function until another call to `set-sig-handler` changes the handling for that signal.

If *handling-spec* is the symbol `nil`, then the function previously associated with the signal, if any, is removed, and the signal is disabled. For a signal to be disabled means that the signal is set to the `SIG_IGN` disposition (refer to the C API).

If *handling-spec* is the symbol `t`, then the function previously associated with the signal, if any, is removed, and the signal is set to its default disposition. This means that it is set to `SIG_DFL` (refer to the C API). Some signals terminate the process if they are generated while the handling is configured to the default disposition.

Note that the certain signals like `sig-quit` and `sig-kill` cannot be ignored or handled. Please observe the signal documentation in the IEEE POSIX standard, and your platform.

A signal handling function must take two arguments. It is of the form:

```
(lambda (signal async-p) ...)
```

The *signal* argument is an integer indicating the signal number for which the handler is being invoked. The *asyncp-p* argument is a Boolean value. If it is `t`, it indicates that the handler is being invoked asynchronously—directly in a signal handling context. If it is `nil`, then it is a deferred call. Handlers may do more things in a deferred call, such as terminate by throwing exceptions, and perform I/O.

The return value of a handler is normally ignored. However if it invoked asynchronously (the *async-p* argument is true), then if the handler returns a non-`nil` value, it is understood that the handler requesting that it be deferred. This means that the signal will be marked as deferred, and the handler will be called again at some later time in a deferred context, whereby *async-p* is `nil`. This is not guaranteed, however; it's possible that another signal will arrive before that happens, possibly resulting in another async call, so the handler must be prepared to deal with an async call at any time.

If a handler is invoked synchronously, then its return value is ignored.

In the current implementation, signals do not queue. If a signal is delivered to the process again, while it is marked as deferred, it simply stays deferred; there is no counter associated with a signal, only a Boolean flag.

9.67.3 Function `sig-check`

Syntax:

```
(sig-check)
```

Description:

The `sig-check` function tests whether any signals are deferred, and for each deferred signal in turn, it executes the corresponding handler. For a signal to be deferred means that the signal was caught by an internal handler in **TXR** and the event was recorded by a flag. If a handler function is removed while a signal is deferred, the deferred flag is cleared for that signal.

Calls to the `sig-check` function may be inserted into CPU-intensive code that has no opportunity to be interrupted by signals, because it doesn't invoke any I/O functions.

9.67.4 Function `raise`

Syntax:

```
(raise signal)
```

Description:

The `raise` function sends *signal* to the process. It is a wrapper for the C function of the same name.

The return value is `t` if the function succeeds, otherwise `nil`.

9.67.5 Function `kill`

Syntax:

```
(kill process-id [signal])
```

Description:

The `kill` function is used for sending a signal to a process group or process. It is a wrapper for the POSIX `kill` function.

If the *signal* argument is omitted, it defaults to the same value as `sig-term`.

The return value is `t` if the function succeeds, otherwise `nil`.

9.67.6 Function `strsignal`

Syntax:

```
(strsignal signal)
```

Description:

The `strsignal` function returns a character string describing the specified signal number. It is based on the same-named POSIX C library function.

9.68 Unix Processes

9.68.1 Functions `fork` and `wait`

Syntax:

```
(fork)
(wait [pid [flags]])
```

Description:

The `fork` and `wait` functions are interfaces to the Unix functions `fork` and `waitpid`.

The `fork` function creates a child process which is a replica of the parent. Both processes return from the function. In the child process, the return value is zero. In the parent, it is an integer representing the process ID of the child. If the function fails to create a child, it returns `nil` rather than an integer. In this case, the `errno` function can be used to inquire about the cause.

The `wait` function, if successful, returns a cons cell consisting of a pair of integers. The `car` of the cons is the process ID of the process or group which was successfully waited on, and the `cdr` is the status. If `wait` fails, it returns `nil`. The `errno` function can be used to inquire about the cause.

The *process-id* argument, if not supplied, defaults to -1, which means that `wait` waits for any process, rather than a specific process. Certain other values have special meaning, as documented in the POSIX standard for the `waitpid` function.

The *flags* argument defaults to zero. If it is specified as nonzero, it should be a bitwise combination (via the `logior` function) of the variables `w-nohang`, `w-untraced` and `w-continued`. If `w-nohang` is used, then `wait` returns a cons cell whose `car` specifies a process ID value of zero in the situation that at least one of the processes designated by `process-id` exist and are children of the calling process, but have not changed state. In this case, the status value in the `cdr` is unspecified.

Status values may be inspected with the functions `w-ifexited`, `w-exitstatus`, `w-ifsignaled`, `w-termsig`, `w-coredump`, `w-ifstopped`, `w-stopsig` and `w-ifcontinued`.

9.68.2 Functions `w-ifexited`, `w-exitstatus`, `w-ifsignaled`, `w-termsig`, `w-coredump`, `w-ifstopped` and `w-stopsig`

Syntax:

```
(w-ifexited status)
(w-exitstatus status)
```

```
(w-ifsignaled status)
(w-termsig status)
(w-coredump status)
(w-ifstopped status)
(w-stopsig status)
(w-ifcontinued status)
```

Description:

These functions analyze process exit values produced by the `wait` function.

They are closely based on the POSIX macros `WIFEXITED`, `WEXITSTATUS`, and so on.

The *`status`* value is either an integer, or a cons cell. In this case, the cons cell is expected to have an integer in its `cdr` which is used as the status.

The `w-ifexited`, `w-ifsignaled`, `w-coredump`, `w-ifstopped` and `w-ifcontinued` functions have Lisp Boolean return semantics, unlike their C language counterparts: they return `t` or `nil`, rather than zero or nonzero. The others return integer values.

9.68.3 Function `exec`

Syntax:

```
(exec file [args])
```

Description:

The exec function replaces the process image with the executable specified by string argument *`file`*. The executable is found by searching the system path.

The *`file`* argument becomes the first argument of the executable, argument zero.

If *`args`* is specified, it is a list of strings. These are passed as the additional arguments of the executable.

If `exec` fails, an exception of type `file-error` is thrown.

9.68.4 Function `exit*`

Syntax:

```
(exit* status)
```

Description:

The `exit*` function terminates the entire process (running **TXR** image), specifying the termination status to the operating system. The *`status`* argument is treated exactly like that of the `exit` function. Unlike that function, this one exits the process immediately, cleaning up only low-level operating system resources such as closing file descriptors and releasing memory mappings, without performing userspace cleanup.

`exit*` is implemented using a call to the POSIX function `_exit`.

9.68.5 Functions `getpid` and `getppid`

Syntax:

```
(getpid)
(getppid)
```

Description:

These functions retrieve the current process ID and the parent process ID respectively. They are wrappers for the POSIX functions `getpid` and `getppid`.

9.68.6 Function `daemon`

Syntax:

```
(daemon nochdir noclose)
```

Description:

This is a wrapper for the function `daemon` which originated in BSD Unix.

It returns `t` if successful, `nil` otherwise, and the `errno` variable is set in that case.

Unlike in the underlying same-named platform function, the *`nochdir`* and *`noclose`* arguments are Boolean, rather than integer values.

9.69 Unix File Descriptors

9.69.1 Function `open-fileno`

Syntax:

```
(open-fileno file-descriptor [mode-string [pid]])
```

Description:

The `open-fileno` function creates and returns a **TXR** stream over a file descriptor. The *`file-descriptor`* argument must be an integer denoting a valid file descriptor.

For a description of *`mode-string`*, see the `open-file` function.

If the *`pid`* argument is present, it must be a positive integer corresponding to a process ID. The `open-fileno` function will associate the process ID with the returned stream. When the stream is closed with `close-stream`, special handling takes place, as documented for that function.

9.69.2 Function `fileno`

Syntax:

```
(fileno stream)
```

Description:

The `fileno` function returns the underlying file descriptor of *`stream`*, if it has one. Otherwise, it returns `nil`.

This is equivalent to querying the stream using `stream-get-prop` for the `:fd` property.

9.69.3 Function `dupfd`

Syntax:

```
(dupfd old-fileno [new-fileno])
```

Description:

The `dupfd` function provides an interface to the POSIX functions `dup` or `dup2`, when called with one or two arguments, respectively.

9.69.4 Function `pipe`

Syntax:

```
(pipe)
```

Description:

The `pipe` function, if successful, returns a pair of integer file descriptors as a cons-cell pair. The descriptor in the `car` field of the pair is the read end of the pipe. The `cdr` holds the write end.

If the function fails, it throws an exception of type `file-error`.

9.69.5 Function `close`

Syntax:

```
(close fileno [throw-on-error-p])
```

Description:

The `close` function passes the integer descriptor *`fileno`* to the POSIX `close` function. If the operation is successful, then `t` is returned. Otherwise an exception of type `file-error` is thrown, unless the *`throw-on-error-p`* argument is present, with a true value. In that case, `close` indicates failure by returning `nil`.

9.69.6 Function `poll`

Syntax:

```
(poll poll-list [timeout])
```

Description:

The `poll` function suspends execution while monitoring one or more file descriptors for specified events. It is a wrapper for the same-named POSIX function.

The *`poll-list`* argument is a sequence of `cons` pairs. The `car` of each pair is either an integer file descriptor, or else a stream object which has a file descriptor (the `fileno` function can be applied to that stream to retrieve a descriptor). The `cdr` of each pair is an integer bit mask specifying the events, whose occurrence the file descriptor is to be monitored for. The variables `poll-in`, `poll-out`, `poll-err` and several others are available which hold bitmask values corresponding to the constants `POLLIN`, `POLLOUT`, `POLLERR` used with the C language `poll` function.

The *`timeout`* argument, if absent, defaults to the value -1, which specifies an indefinite wait. A nonnegative value specifies a wait with a timeout, measured in milliseconds.

The function returns a list of pairs representing the descriptors or streams which were successfully polled. If the function times out, it returns an empty list. If an error occurs, an exception is thrown.

The returned list is similar in structure to the input list. However, it holds only entries which polled positive. The `cdr` of every pair now holds a bitmask of the events which were to have occurred.

9.69.7 Function `isatty`

Syntax:

```
(isatty stream)
(isatty fileno)
```

Description:

The `isatty` function provides access to the underlying POSIX function of the same name.

If the argument is a *`stream`* object which has a `:fd` property, then the file descriptor number is retrieved. The behavior is then as if that descriptor number were passed as the *`fileno`* argument.

If the argument is not a *`stream`*, it must be a *`fileno`*: an integer in the representation range of the C type `int`.

The POSIX `isatty` is invoked on this integer. If it that returns 1, then `t` is returned, otherwise `nil`.

9.70 Unix File Control

9.70.1 Variables `o-accmode`, `o-rdonly`, `o-wronly`, `o-rdwr`, `o-creat`, `o-noctty`, `o-trunc`, `o-append`, `o-nonblock`, `o-sync`, `o-async`, `o-directory`, `o-nofollow`, `o-cloexec`, `o-direct`, `o-noatime` and `o-path`

Description:

These variables correspond to the POSIX file mode constants `O_ACCMODE`, `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT`, `O_NOCTTY`, and so forth.

The availability of the variables `o-async`, `o-directory`, `o-nofollow`, `o-cloexec`, `o-direct`, `o-noatime` and `o-path` depends on the host platform.

Some of these flags may be set or cleared on an existing file descriptor using the `f-setfl` command of the `fcntl` function, in accordance with POSIX and the host platform documentation.

9.70.2 Variables `seek-set`, `seek-cur` and `seek-end`

Description:

These variables correspond to the ISO C constants `SEEK_SET`, `SEEK_CUR` and `SEEK_END`. These values, usually associated with the ISO C `fseek` function, are also used in the `fcntl` file locking interface as values of the `whence` member of the `flock` structure.

9.70.3 Variables `f-dupfd`, `f-dupfd-cloexec`, `f-getfd`, `f-setfd`, `f-getfl`, `f-setfl`, `f-getlk`, `f-setlk` and `f-setlkw`

Description:

These variables correspond to the POSIX `fcntl` command constants `F_DUPFD`, `F_GETFD`, `F_SETFD`, and so forth. Availability of the `f-dupfd-cloexec` depends on the host platform.

9.70.4 Variable `fd-cloexec`

Description:

The `fd-cloexec` variable corresponds to the POSIX `FD_CLOEXEC` constant. It denotes the flag which may be set by the `fd-setfd` command of the `fcntl` function.

9.70.5 Variables `f-rdlck`, `f-wrlck` and `f-unlck`

Description:

These variables correspond to the POSIX lock type constants `F_RDLCK`, `F_WRLCK` and

F_UNLCK. They specify the possible values of the type field of the flock structure.

9.70.6 Structure flock

Syntax:

```
(defstruct flock nil
   type whence
   start len
   pid)
```

Description:

The flock structure corresponds to the POSIX structure of the same name. An instance of this structure must be specified as the third argument of the fcntl function when the *command* argument is one of the values f-getlk, f-setlk or f-setlkw.

All slots must be initialized with appropriate values before calling fcntl with the exception that the f-getlk command does not access the existing value of the pid slot.

9.70.7 Function fcntl

Syntax:

```
(fcntl fileno command [arg])
```

Description:

The fcntl function corresponds to the same-named POSIX function. The *fileno* and *command* arguments must be integers. The **TXR Lisp** fileno restricts the *command* argument to the supported values for which symbolic variable names are provided. Other integer *command* values are rejected by returning -1 and setting the errno variable to EINVAL. Whether the third argument is required, and what type it must be, depends on the *command* value. Commands not requiring the third argument ignore it if it is passed.

fcntl commands for which POSIX requires an argument of type long require *arg* to be an integer.

The file locking commands f-getlk, f-setlk and f-setlkw require *arg* to be a flock structure.

The fcntl function doesn't throw an error if the underlying POSIX function indicates failure; the underlying function's return value is converted to a Lisp integer and returned.

9.71 Unix Itimers

Itimers ("interval timers") can be used in combination with signal handling to execute asynchronous actions. Itimers deliver delayed, one-time signals, and also periodically recurring signals. For more information, consult the POSIX specification.

9.71.1 Variables itimer-real, itimer-virtual and itimer-prof

Description:

These variables correspond to the POSIX constants ITIMER_REAL, ITIMER_VIRTUAL and ITIMER_PROF. Their values are suitable as the *timer* argument of the getitimer and setitimer functions.

9.71.2 Functions `getitimer` and `setitimer`

Syntax:

```
(getitimer timer)
(setitimer timer interval value)
```

Description:

The `getitimer` function returns the current value of the specified timer, which must be `itimer-real`, `itimer-virtual` or `itimer-prof`.

The current value consists of a list of two integer values, which represents microseconds. The first value is the timer interval, and the second value is the timer's current value.

Like `getitimer`, the `setitimer` function also retrieves the specified timer. In addition, it stores a new value in the timer, which is given by the two arguments, expressed in microseconds.

9.72 Unix Syslog

On platforms where a Unix-like syslog API is available, **TXR** exports this interface. **TXR** programs can configure logging via the `openlog` function, control the logging mask via `setlogmask` and generate logs via `syslog`, or using special syslog streams.

9.72.1 Variables `log-pid`, `log-cons`, `log-ndelay`, `log-odelay`, `log-nowait` and `log-perror`

Description:

These variables take on the values of the corresponding C preprocessor constants from the `<syslog.h>` header: `LOG_PID`, `LOG_CONS`, etc. These integer values represent logging options used in the *`options`* argument to the `openlog` function.

Note: `LOG_PERROR` is not in POSIX, and so `log-perror` might not be available. See notes about `LOG_AUTHPRIV` in the documentation for `log-authpriv`.

9.72.2 Special variables `log-user`, `log-daemon`, `log-auth` and `log-authpriv`

Description:

These variables take on the values of the corresponding C preprocessor constants from the `<syslog.h>` header: `LOG_USER`, `LOG_DAEMON`, `LOG_AUTH` and `LOG_AUTHPRIV`. These are the integer facility codes specified in the `openlog` function.

Note: `LOG_AUTHPRIV` is not in POSIX, and so `log-authpriv` might not be available. For portability use code like `(or (symbol-value 'log-authpriv) 0)` to evaluate to 0 if `log-authpriv` doesn't exist, or else check for its existence using `(boundp 'log-authpriv)`.

9.72.3 Variables `log-emerg`, `log-alert`, `log-crit`, `log-err`, `log-warning`, `log-notice`, `log-info` and `log-debug`

Description:

These variables take on the values of the corresponding C preprocessor constants from the `<syslog.h>` header: `LOG_EMERG`, `LOG_ALERT`, etc. These are the integer priority codes specified in the `syslog` function.

9.72.4 Special variable `*stdlog*`

Description:

The `*stdlog*` variable holds a special kind of stream: a syslog stream. Each newline-terminated line of text sent to this stream becomes a log message.

The stream internally maintains a priority value that is applied when it generates messages. By default, this value is that of `log-info`. The stream holds the priority as the value of the `:prio` stream property, which may be changed with the `stream-set-prop` function.

The latest priority value which has been configured on the stream is used at the time the newline character is processed and the log message is generated, not necessarily the value which was in effect at the time the accumulation of a line began to take place.

Messages sent to `*stdlog*` are delimited by newline characters. That is to say, each line of text written to the stream is a new log.

9.72.5 Function `openlog`

Syntax:

```
(openlog id-string [options [facility]])
```

Description:

The `openlog` function is a wrapper for the `openlog` C function, and the arguments have the same semantics. It is not necessary to use `openlog` in order to call the `syslog` function or to write data to `*stdlog*`. The call is necessary in order to override the default identifying string, to set options, such as having the PID (process ID) recorded in log messages, and to specify the facility.

The *`id-string`* argument is mandatory.

The *`options`* argument is a bitwise mask (see the logior function) of option values such as `log-pid` and `log-cons`. If it is missing, then a value of 0 is used, specifying the absence of any options.

The *`facility`* argument is one of the values `log-user`, `log-daemon` or `log-auth`. If it is missing, then `log-user` is assumed.

9.72.6 Function `closelog`

Syntax:

```
(closelog)
```

Description:

The `closelog` function is a wrapper for the C function `closelog`.

9.72.7 Function `setlogmask`

Syntax:

```
(setlogmask bitmask-integer)
```

Description:

The `setlogmask` function interfaces to the corresponding C function, and has the same argument and return value semantics. The *`bitmask-integer`* argument is a mask of priority values

to enable. The return value is the prior value. Note that if the argument is zero, then the function doesn't set the mask to zero; it only returns the current value of the mask.

Note that the priority values like `log-emerg` and `log-debug` are integer enumerations, not bitmasks. These values cannot be combined directly to create a bitmask. Rather, the `mask` function should be used on these values.

Example:

```
;; Enable LOG_EMERG and LOG_ALERT messages,
;; suppressing all others
(setlogmask (mask log-emerg log-alert))
```

9.72.8 Function `syslog`

Syntax:

(syslog *priority format format-arg**)

Description:

The `syslog` function is the interface to the `syslog` C function. The `printf` formatting capabilities of the function are not used; the *`format`* argument follows the conventions of the **TXR Lisp** `format` function instead. Note in particular that the `%m` convention for interpolating the value of strerror(errno) which is available in some versions of the `syslog` C function is currently not supported.

Note that syslog messages are not newline-terminated.

9.73 Unix Path Globbing

On platforms where the POSIX `glob` function is available **TXR** provides this functionality in the form of a like-named function, and some numeric constants. **TXR** also provides access the `fnmatch` function, where available.

9.73.1 Variables `glob-err`, `glob-mark`, `glob-nosort`, `glob-nocheck`, `glob-noescape`, `glob-period`, `glob-altdirfunc`, `glob-brace`, `glob-nomagic`, `glob-tilde`, `glob-tilde-check` and `glob-onlydir`

Description:

These variables take on the values of the corresponding C preprocessor constants from the `<glob.h>` header: `GLOB_ERR`, `GLOB_MARK`, `GLOB_NOSORT`, etc.

These values are passed as the optional second argument of the `glob` function. They are bitmasks and so multiple values can be combined using the `logior` function.

Note that the `glob-period`, `glob-altdirfunc`, `glob-brace`, `glob-nomagic`, `glob-tilde`, `glob-tilde-check` and `glob-onlydir` variables may not be available. They are extensions in the GNU C library implementation of `glob`.

The standard `GLOB_APPEND` flag is not represented as a **TXR** variable. The `glob` function uses it internally when calling the C library function multiple times, due to having been given multiple patterns.

9.73.2 Function `glob`

Syntax:

```
(glob {pattern | patterns} [flags [errfun]])
```

Description:

The `glob` function is a interface to the Unix function of the same name. The first argument must either be a single *`pattern`*, which is a string, or else sequence of strings specifying multiple *`patterns`*, which are strings. Each string is a glob pattern: a pattern which matches zero or more pathnames, similar to a regular expression. The function tries to expand the patterns and return a list of strings representing the matching pathnames in the file system.

If there are no matches, then an empty list is returned.

The optional *`flags`* argument defaults to zero. If given, it may be a bitwise combination of the values of the variables `glob-err`, `glob-mark`, `glob-nosort` and others. The `glob-append`

If the *`errfun`* argument is specified, it gives a callback function which is invoked when `glob` encounters errors accessing paths. The function takes two arguments: the pathname and the `errno` value which occurred for that pathname. The function's return value is Boolean. If the function returns true, then `glob` will terminate.

The *`errfun`* may terminate the traversal by a nonlocal exit, such as by throwing an exception or performing a block return.

The *`errfun`* may not reenter the `glob` function. This situation is detected and diagnosed by an exception.

The *`errfun`* may not capture a continuation across the error boundary. That is to say, code invoked from the error may not capture a continuation up to a prompt which surrounds the `glob` call. Such an attempt is detected and diagnosed by an exception.

If a sequence of *`patterns`* is specified instead of a single pattern, `glob` makes multiple calls to the underlying C library function. The second and subsequent calls specify the `GLOB_APPEND` flag to add the matches to the result. The following equivalence applies:

```
(glob (list p0 p1 ...) f e)  <-->  (append (glob p0 f e)
                                           (glob p1 f e)
                                           ...)
```

Details of the semantics of the `glob` function, and the meaning of all the *`flags`* arguments are given in the documentation for the C function.

9.73.3 Variables `fnm-pathname`, `fnm-noescape`, `fnm-period`, `fnm-leading-dir`, `fnm-casefold` and `fnm-extmatch`

Description:

These variables take on the values of the corresponding C preprocessor constants from the `<fnmatch.h>` header: `FNM_PATHNAME`, `FNM_NOESCAPE`, `FNM_PERIOD`, etc.

These values are bit masks which may be combined with the `logior` function to form the optional third *`flags`* argument of the `fnmatch` function.

Note that the `fnm-leading-dir`, `fnm-casefold` and `fnm-extmatch` functions may not be available. They are GNU extensions, found in the GNU C library.

9.73.4 Function `fnmatch`

Syntax:

```
(fnmatch pattern string [flags]])
```

Description:

The `fnmatch` function, if available, provides access to the like-named POSIX C library function. The *`pattern`* argument specifies a POSIX-shell-style filename-pattern-matching expression. Its exact features and dialect are controlled by *`flags`*. If *`string`* matches *`pattern`* then `t` is returned. If there is no match, then `nil` is returned. If the C function indicates that an error has occurred, an exception is thrown.

9.74 Unix Filesystem Traversal

On platforms where the POSIX `nftw` function is available **TXR** provides this functionality in the form of the analogous Lisp function `ftw`, accompanied by some numeric constants.

Likewise, on platforms where the POSIX functions `opendir` and `readdir` are available, **TXR** provides the functionality in the form of same-named Lisp functions, a structure type named `dirent` and some accompanying numeric constants.

9.74.1 Variables `ftw-phys`, `ftw-mount`, `ftw-chdir`, `ftw-depth` and `ftw-actionretval`

Description:

These variables hold numeric values that may be combined into a single bitmask bitmask value using the `logior` function. This value is suitable as the *`flags`* argument of the `ftw` function.

These variables correspond to the C constants `FTW_PHYS`, `FTW_MOUNT`, etc.

Note that `ftw-actionretval` is a GNU extension that is not available on all platforms. If the platform's `nftw` function doesn't have this feature, then this variable is not defined.

9.74.2 Variables `ftw-f`, `ftw-d`, `ftw-dnr`, `ftw-ns`, `ftw-sl`, `ftw-dp` and `ftw-sln`

Description:

These variables provide symbolic names for the integer values that are passed as the `type` argument of the callback function called by `ftw`. This argument classifies the kind of file system node visited, or error condition encountered.

These variables correspond to the C constants `FTW_F`, `FTW_D`, etc.

Not all of them are present. If the underlying platform doesn't have a given constant, then the corresponding variable doesn't exist in **TXR**.

9.74.3 Variables `ftw-continue`, `ftw-stop`, `ftw-skip-subtree` and `ftw-skip-siblings`

Description:

These variables are defined if the variable `ftw-actionretval` is defined.

If the value of `ftw-actionretval` is included in the *`flags`* argument of `ftw`, then the

callback function can use the values of these variables as return codes. Ordinarily, the callback returns zero to continue the search and nonzero to stop.

These variables correspond to the C constants `FTW_CONTINUE`, `FTW_STOP`, etc.

9.74.4 Function `ftw`

Syntax:

```
(ftw path-or-list callbackfun [flags [nopenfd]])
[callbackfun path type stat-struct level base]
```

Description:

The `ftw` function provides access to the `nftw` POSIX C library function.

Note that the *`flags`* and *`nopenfd`* arguments are reversed with respect to the C language interface. They are both optional; *`flags`* defaults to the value of `ftw-phys` and *`nopenfd`* defaults to 20. If an argument is given to *`flags`*, then the presence of the `ftw-phys` is no longer implied; the flag must be explicitly included in the argument in order to be present.

Compatibility Note: the *`flags`* parameter defaults to an argument value of zero in **TXR** versions 283 or lower.

The *`path-or-list`* argument may be a string specifying the top-level pathname that `ftw` shall visit. Or else, *`path-or-list`* may be a list. If it is a list, then `ftw` recursively invokes itself over each of the elements, taking that element as the *`path-or-name`* argument of the recursive call, passing down all other argument values as-is. The traversal stops when any recursive invocation of `ftw` returns a value other than `t` or `nil`, and that value is returned. If `t` or `nil` is returned, the traversal continues with the application of `ftw` to the next list element, if any. If the list is completely traversed, and some recursive invocations of `ftw` return `t`, then the return value is `t`. If all recursive invocations return `nil` then `nil` is returned. If the list is empty, `t` is returned.

The `ftw` function walks the filesystem, as directed by the *`path-or-list`* argument and *`flags`* bitmask arguments.

For each visited entry, it calls the supplied *`callbackfun`* function, which receives five arguments. If this function returns normally, it must return either `nil`, `t`, or an integer value in the range of the C type `int`.

The `ftw` function can continue the traversal by returning any non-integer value, or the integer value zero. If `ftw-actionretval` is included in the *`flags`* bitmask, then the only integer code which continues the traversal without any special semantics is `ftw-continue` and only `ftw-stop` stops the traversal. (Non-integer return values behave like `ftw-continue`).

The *`path`* argument of *`callbackfun`* gives the path of the visited filesystem object.

The *`type`* argument is an integer code which indicates the kind of object that is visited, or an error situation in visiting that filesystem entry. See the documentation for `ftw-f` and `ftw-d` for possible values.

The *`stat-struct`* argument provides information about the filesystem object as a `stat` structure, the same kind of object as what is returned by the `stat` function.

The *`level`* argument is an integer value representing the directory level depth. This value is obtained from the C structure `FTW` in the `nftw` C API.

The *base* argument indicates the length of the directory part of the path argument. Characters in excess of this length are thus the base name of the visited object, and the expression [*path base*..:] calculates the base name.

The ftw function returns either t upon successful completion, or an integer value returned by *callbackfun*, as described below. On failure it throws an exception derived from file-error, whose specific type is based on analyzing the POSIX errno value.

The *callbackfun* may return a value of any type. If it returns a value that is not of integer type, then zero is returned to the nftw function and traversal continues. Similarly, traversal continues if the function returns an integer zero.

If *callbackfun* returns an integer value, that value must be in the range of the C type int. That int value is returned to nftw. If the value is not zero, and is not -1, then nftw will terminate, and return that value, which ftw then returns. If the value is -1, then nftw is deemed to have failed, and ftw will thrown an exception of type file-error, whose specific type is based on analyzing the POSIX errno value. If the value is zero, then the traversal continues.

The *callbackfun* may also terminate the traversal by a nonlocal exit, such as by throwing an exception or performing a block return.

The *callbackfun* may not reenter the ftw function. This situation is detected and diagnosed by an exception.

The *callbackfun* may not capture a continuation across the callback boundary. That is to say, code invoked from the callback may not capture a continuation up to a prompt which surrounds the ftw call. Such an attempt is detected and diagnosed by an exception.

9.74.5 Structure dirent

Syntax:

```
(defstruct dirent nil
   name ino type)
```

Description:

Objects of the dirent structure type are returned by the readdir function.

The name slot is a character string giving the name of the directory entry. If the opendir function's *prefix-p* argument is specified as true, then readdir operations produce dirent structures whose name slot is a path formed by combining the directory path with the directory entry name.

The ino slot is an integer giving the inode number of the object named by the directory entry.

The type slot indicates the type of the object, which is an integer code. Support for this member is platform-dependent. If the directory traversal doesn't provide the information, then this slot takes on the nil value. In this situation, the dirstat function may be used to backfill the missing information.

9.74.6 Variables dt-blk, dt-chr, dt-dir, dt-fifo, dt-lnk, dt-reg, dt-sock and dt-unknown

Description:

These variables give the possible type code values exhibited by the `type` slot of the `dirent` structure.

If the underlying host platform does not feature a `d_type` field in the `dirent` C structure, then almost all these variables are defined anyway using the values that they have on GNU/Linux. These definitions are useful in conjunction with the `dirstat` function below.

If the host platform does does not feature a `d_type` field in the `dirent` structure, then the variable `dt-unknown` is not defined. Note: the application can take advantage of this this to detect the situation, in order to conditionally define code in such a way that some run-time checking is avoided.

9.74.7 Function `opendir`

Syntax:

```
(opendir dir-path [prefix-p])
```

Description:

The `opendir` function initiates a traversal of the directory object named by the string argument *`dir-path`*, which must be the name of a directory. If `opendir` is not able to open the directory traversal, it throws an exception of type `system-error`. Otherwise an object of type `dir` is returned, which is a directory traversal handle suitable as an argument for the `readdir` function.

If the *`prefix-p`* argument is specified and has a true value, then it indicates that the subsequent `readdir` operations should produce the value of the `name` slot of the `dirent` structure by combining *`dir-path`* with the directory entry name using the `path-cat` function.

9.74.8 Function `readdir`

Syntax:

```
(readdir dir-handle [dirent-struct])
```

Description:

The `readdir` function returns the next available directory entry from the directory traversal controlled by *`dir-handle`*, which must be a `dir` object returned by `opendir`.

If no more directory entries remain, then `readdir` returns `nil`. In this situation, the *`dir-handle`* is also closed, as if by a call to `closedir`.

Otherwise, the next available directory entry is returned as a structure object of type `dirent`.

The `readdir` function internally skips and does not report the `"."` (dot) and `".."` (dotdot) directory entries.

If the *`dirent-struct`* argument is specified, then it must be a `dirent` structure, or one which has all of the required slots. In this case, `readdir` stores values in that structure and returns it. If *`dirent-struct`* is absent, then `readdir` allocates a fresh `dirent` structure.

9.74.9 Function `closedir`

Syntax:

```
(opendir dir-handle)
```

Description:

The `closedir` function terminates the directory traversal managed by *`dir-handle`*, releasing its resources.

If this has already been done before, `closedir` returns `nil`, otherwise it returns `t`.

Further `readdir` calls on the same *`dir-handle`* return `nil`.

Note: the `readdir` function implicitly closes *`dir-handle`* when the handle indicates that no more directory entries remain to be traversed.

9.74.10 Function `dirstat`

Syntax:

```
(dirstat dirent-struct [dir-path [struct]])
```

Description:

The `dirstat` function invokes `lstat` on the object represented by the `dirent` structure *`dirent-struct`*, sets the `type` slot of the *`dirent-struct`* accordingly, and then returns the value that `lstat` returned.

If the *`struct`* argument is specified, it is passed to `lstat`.

The *`dir-path`* parameter must be specified, if the `name` slot of *`dirent-struct`* is a simple directory entry name, rather than the full path to the object. In that case, the slot's value gives the effective path. If the `name` slot is already a path (due to, for instance, a true value of *`prefix-p`* having been passed to `opendir`) then *`dir-path`* must not be specified. If *`dir-path`* is specified, then its value is combined with the *`name`* slot of *`dirent-struct`* using `path-cat` to form the effective path.

The `lstat` function is invoked on the effective path, and if it succeeds, then type information is obtained from the resulting structure to set the value of the `type` slot of *`dirent-struct`*. The same structure that was returned by `lstat` is then returned.

9.75 Unix Sockets

On platforms where the underlying system interface is available, **TXR** provides a sockets library for communicating over Internet networks, or over Unix sockets.

Stream as well as datagram sockets are supported.

The classic Version 4 of the Internet protocol is supported, as well as IP Version 6.

Sockets are mapped to **TXR** streams. The `open-socket` function creates a socket of a specified type, in a particular address family. This socket is actually a stream (always, even if it isn't used for data transfer, but only as a passive contact point).

The functions `sock-connect`, `sock-bind`, `sock-listen`, `sock-accept` and `sock-shutdown` are used for enacting socket communication scenarios.

Stream sockets use ordinary streams, reusing the same underlying framework that is used for file I/O and process types.

Datagram socket streams are implemented using special datagram socket streams. Datagram socket

streams eliminate the need for operations analogous to the `sendto` and `recvfrom` socket API functions, even in server programs which handle multiple clients. An overview of datagrams is treated in the following section, Datagram Socket Streams.

The `getaddrinfo` function is provided for resolving host names and services to IPv4 and IPv6 addresses.

Several structure types are provided for representing socket addresses, and options for `getaddrinfo`.

Various numeric constants are also provided: `af-unix`, `af-inet`, `af-inet6`, `sock-stream`, `sock-dgram` and others.

9.75.1 Datagram Socket Streams

Datagram socket streams are a new paradigm unique to **TXR** which attempts to unify the programming model of stream and datagram sockets.

A datagram socket stream is created by the `open-socket` function, when the `sock-dgram` socket type is specified. Another way in which a datagram socket is created is when `sock-accept` is invoked on a datagram socket, and returns a new socket.

I/O is performed on datagram sockets using the regular I/O functions. None of the functions take or return peer addresses. There are no I/O functions which are analogous to the C library `recvfrom` and `sendto` functions which are usually used for datagram programming. Datagram I/O assumes that the datagram datagram socket is connected to a specific remote peer, and that peer is implicitly used for all I/O.

Datagram streams solve the message framing problem by considering a single datagram to be an entire stream. On input, a datagram stream holds an entire datagram in a buffer. The stream ends (experiences the EOF condition) after the last byte of this buffer is removed by an input operation. Another datagram will be received and buffered if the EOF condition is first explicitly cleared with the `clear-error` function, and then another input operation is attempted. On output, a datagram stream gathers data into an ever-growing output buffer which isn't subject to any automatic flushing. An explicit `flush-stream` operation sends the buffer contents to the connected peer as a new datagram, and empties the buffer. Subsequent output operations prepare data for a new datagram. The `close-stream` function implicitly flushes the stream in the same way, and thus also potentially generates a datagram.

A client-style datagram stream can be explicitly connected to a peer with the `sock-connect` function. This is equivalent to connecting a datagram socket using the C library `connect` function. Writes on the stream will be transmitted using the C library function `send`. A client-style datagram stream can also be "soft-connected" to a peer using the `sock-set-peer` function. Writes on the stream will transmit data using the C library function `sendto` to the peer address.

A datagram server program which needs to communicate using multiple peers is implemented by means of the `sock-accept` function which, unlike the C library `accept` function, works with datagram sockets as well as stream sockets. The server creates a datagram socket, and uses `sock-bind` to bind it to a local address. Optionally, it may also call `sock-listen` which is a no-op on datagram sockets. Supporting this function on datagram sockets allows program code to be more easily changed between datagram and stream operation. The server then uses `sock-accept` to accept new clients. Note that this is not possible with the C library function `accept`, which only works with stream sockets.

The `sock-accept` function receives a datagram from a client, and creates a new datagram socket stream which is connected to that client, and whose input buffer contains the received datagram. Input operations on this stream consume the datagram. Note that clearing the EOF condition and trying to receive another datagram is not recommended on datagram streams returned by `sock-accept`, since they share the same underlying operating system socket, which is not actually connected to a specific peer. The receive

operation could receive a datagram from any peer, without any indication which peer that is. Datagram servers should issue a new `sock-accept` call for each client datagram, treating it as a new stream.

Datagram sockets ignore almost all aspects of the *`mode-string`* passed in `open-socket` and `sock-accept`. The only attribute not ignored is the buffer size specified with a decimal digit character; however, it cannot be the only item in the mode string. The string must be syntactically valid, as described under the `open-file` function. The buffer size attribute controls the size used by the datagram socket for receiving a datagram: the capture size. A datagram socket has obtains a default capture size if one isn't specified by the *`mode-string`*. The default capture size is 65536 bytes for a datagram socket created by `open-socket`. If a size is not passed to `sock-accept` via its *`mode-string`* argument when it is invoked on a datagram socket, that socket's size is used as the capture size of the newly created datagram socket which is returned.

9.75.2 Structure `sockaddr`

Syntax:

```
(defstruct sockaddr nil
  canonname
  (:static family nil))
```

Description:

The `sockaddr` structure represents the abstract base class for socket addresses, from which several other types are derived: `sockaddr-in`, `sockaddr-in6` and `sockaddr-un`.

It has a single static slot named `family` and a single instance slot `canonname`, both initialized to `nil`.

Note: the `canonname` slot is optionally set by the `getaddrinfo` function on address structures that it returns, if requested via the `ai-canonname` flag. The slot only provides information to the application, playing no semantic role in addressing.

9.75.3 Structure `sockaddr-in`

Syntax:

```
(defstruct sockaddr-in sockaddr
  (addr 0) (port 0) (prefix 32)
  (:static family af-inet))
```

Description:

The `sockaddr-in` address represents a socket address used in the context of networking over IP Version 4. It may be used with sockets in the `af-inet` address family.

The `addr` slot holds an integer denoting an abstract IPv4 address. For instance the hexadecimal integer literal constant `#x7F000001` or its decimal equivalent `2130706433` represents the loopback address, whose familiar "dot notation" is `127.0.0.1`. Conversion of the abstract IP address to four bytes in network order, as required, is handled internally.

The `port` slot holds the TCP or UDP port number, whose value ranges from 0 to 65535. Zero isn't a valid port; the value is used for requesting an ephemeral port number in active connections. Zero also appears in situations when the port number isn't required: for instance, when the `getaddrinfo` function is used with the aim of looking up the address of a host, without caring about the port number.

The `prefix` field is set by the function `inaddr-str`, when it recognizes and parses a prefix

field in the textual representation.

The `family` static slot holds the value `af-inet`.

9.75.4 Structure `sockaddr-in6`

Syntax:

```
(defstruct sockaddr-in6 sockaddr
  (addr 0) (port 0) (flow-info 0) (scope-id 0)
  (prefix 128)
  (:static family af-inet6))
```

Description:

The `sockaddr-in6` address represents a socket address used in the context of networking over IP Version 6. It may be used with sockets in the `af-inet6` address family.

The `addr` slot holds an integer denoting an abstract IPv6 address. IPv6 addresses are pure binary integers up to 128 bits wide.

The `port` slot holds the TCP or UDP port number, whose value ranges from 0 to 65535. In IPv6, the port number functions similarly to IPv6; see `sockaddr-in`.

The `flow-info` and `scope-id` are special IPv6 parameters corresponding to the `sin6_flowinfo` and `sin6_scope_id` slots of the `sockaddr_in6` C language structure. Their meaning and use are beyond the scope of this document.

The `prefix` field is set by the function `in6addr-str`, when it recognizes and parses a prefix field in the textual representation.

The `family` static slot holds the value `af-inet6`.

9.75.5 Structure `sockaddr-un`

Syntax:

```
(defstruct sockaddr-un sockaddr
  path
  (:static family af-unix))
```

Description:

The `sockaddr-un` address represents a socket address used for interprocess communication within a single operating system node, using the "Unix domain" sockets of the `af-unix` address family.

This structure has only one slot, `path` which holds the rendezvous name for connecting pairs of socket endpoints. This name appears in the filesystem.

When the `sockaddr-un` structure is converted to the C structure `struct sockaddr_un`, the `path` slot undergoes conversion to UTF-8. The resulting bytes are stored in the `sun_path` member of the C structure. If the resulting UTF-8 byte string is larger than the `sun_path` array, it is silently truncated.

Note: Linux systems have support for "abstract" names which do not appear in the filesystem. These abstract names are distinguished by starting with a null byte. For more information, consult Linux documentation. This convention is supported in the `path` slot of the `sockaddr-un`

structure. If `path` contains occurrences of the pseudo-null character U+DC00, these translate to null bytes in the `sun_path` member of the corresponding C structure `struct sockaddr_un`. For example, the path `"\xDC00;foo"` is valid and represents an abstract address consisting of the three bytes `"foo"` followed by null padding bytes.

The `family` static slot holds the value `af-unix`.

9.75.6 Structure `addrinfo`

Syntax:

```
(defstruct addrinfo nil
  (flags 0) (family 0) (socktype 0))
```

Description:

The `addrinfo` structure is used in conjunction with the `getaddrinfo` function. If that function's *`hints`* argument is specified, it is of this type. The purpose of the argument is to narrow down or possibly alter the selection of addresses which are returned.

The `flags` slot holds a bitwise or combination (see the `logior` function) of `getaddrinfo` flags: values given by the variables `ai-passive`, `ai-numerichost`, `ai-v4mapped`, `ai-canonname`, `ai-all`, `ai-addrconfig` and `ai-numericserv`. These correspond to the C constants `AI_PASSIVE`, `AI_NUMERICHOST` and so forth.

If `ai-canonname` is specified, then every returned address structure will have its `canonname` member set to a string value rather than `nil`. This string is a copy of the canonical name reported by the underlying C library function, which that function places only into the first returned address structure.

The `family` slot holds an address family, which may be the value of `af-unspec`, `af-unix`, `af-inet` or `af-inet6`.

The `socktype` slot holds, a socket type. Socket types are given by the variables `sock-dgram` and `sock-stream`.

9.75.7 Function `getaddrinfo`

Syntax:

```
(getaddrinfo [node [service [hints]]])
```

Description:

The `getaddrinfo` returns a list of socket addresses based on search criteria expressed in its arguments. That is to say, the returned list, unless empty, contains objects of type `sockaddr-in` and `sockaddr-in6`.

The function is implemented directly in terms of the like-named C library function. All parameters are optional. Omitting any argument causes a null pointer to be passed for the corresponding parameter of the C library function.

The *`node`* and *`service`* parameters may be character strings which specify a host name, and service. The contents of these strings may be symbolic, like `"www.example.com"` and `"ssh"` or numeric, like `"10.13.1.5"` and `"80"`.

If an argument is given for the `hints` parameter, it must be of type `addrinfo`.

The *node* and *service* parameters may also be given integer arguments. An integer argument value in either of these parameters is converted to a null pointer when calling the C `getaddrinfo` function. The integer values are then simply installed into every returned address as the IP address or port number, respectively. However, if both arguments are numeric, then no addresses are returned, since the C library function is then called with a null node and service.

9.75.8 Variables `af-unix`, `af-inet` and `af-inet6`

Description:

These variables hold integers which give the values of address families. They correspond to the C constants `AF_UNIX`, `AF_INET` and `AF_INET6`. Address family values are used in the *hints* argument of the `getaddrinfo` function, and in the `open-socket` function. Note that unlike the C language socket addressing structures, the **TXR** socket addresses do not contain an address family slot. That is because they indicate their family via their type. That is to say, an object of type `sockaddr-in` is an address which is implicitly associated with the `af-inet` family via its type.

9.75.9 Variables `sock-stream` and `sock-dgram`

Description:

These variables hold integers which give the values of address families. They correspond to the C constants `SOCK_STREAM` and `SOCK_DGRAM`.

9.75.10 Variables `ai-passive`, `ai-numerichost`, `ai-v4mapped`, `ai-all`, `ai-addrconfig` and `ai-numericserv`

Description:

These variables hold integers which are bitmasks that combine together via bitwise or, to express the `flags` slot of the `addrinfo` structure. They correspond to the C constants `AI_PASSIVE`, `AI_NUMERICHOST`, `AI_V4MAPPED` and so forth. They influence the behavior of the `getaddrinfo` function.

9.75.11 Variables `inaddr-any`, `inaddr-loopback`, `in6addr-any` and `in6addr-loopback`

Description:

These integer-valued variables provide constants for commonly used IPv4 and IPv6 address values.

The value of `inaddr-any` and `in6addr-any` is zero. This address is used in binding a passive socket to all of the external interfaces of a host, so that it can accept connections or datagrams from all attached networks.

The `inaddr-loopback` variable is IPv4 loopback address, the same integer as the hexadecimal constant `#x7F000001`.

The `in6addr-loopback` is the IPv6 loopback address. Its value is 1.

Example:

```
;; Construct an IPv6 socket address suitable for binding
;; a socket to the loopback network, port 1234:
(new sockaddr-in6 addr in6addr-loopback port 1234)
```

```
;; Mistake: IPv4 address used with IPv6 sockaddr.
(new sockaddr-in6 addr inaddr-loopback)
```

9.75.12 Function `open-socket`

Syntax:

```
(open-socket family type [mode-string])
```

Description:

The `open-socket` function creates a socket, which is a kind of stream.

The *`family`* parameter specifies the address family of the socket. One of the values `af-unix`, `af-inet` or `af-inet6` should be used to create a Unix domain, Internet IPv4 or Internet IPv6 socket, respectively.

The *`type`* parameter specifies the socket type, either `sock-stream` (stream socket) or `sock-dgram` (datagram socket).

The *`mode-string`* specifies several properties of the stream; for a description of *`mode-string`* parameters, refer to the `open-file` function. Note that the defaulting behavior for an omitted *`mode-string`* argument is different under `open-socket` from other functions. Because sockets are almost always used for bidirectional data flow, the default mode string is `"r+b"` rather than the usual `"r"`.

The rationale for including the `"b"` flag in the default mode string is that network protocols are usually defined in a way that is independent of machine and operating system, down to the byte level, even when they are textual. It doesn't make sense for the same **TXR** program to see a network stream differently based on what platform it is running on. Line-ending conversion has to do with how a platform locally stores text files, whereas network streams are almost always external formats.

Like other stream types, stream sockets are buffered and marked as non-real-time streams. Specifying the `"i"` mode in *`mode-string`* marks a socket as a real-time stream, and, if it is opened for writing or reading and writing, changes it to use line buffering.

9.75.13 Function `open-socket-pair`

Syntax:

```
(open-socket-pair family type [mode-string])
```

Description:

The `open-socket-pair` function provides an interface to the functionality of the `socket-pair` C library function.

If successful, it creates and returns a list of two stream objects, which are sockets that are connected together.

Note: the Internet address families `af-inet` and `af-inet6` are not supported.

The `mode-string` is applied to each stream. For a description, see `open-socket` and `open-file`.

9.75.14 Functions `sock-family` and `sock-type`

Syntax:

```
(sock-family socket)
(sock-type socket)
```

Description:

These functions retrieve the integer values representing the address family and type of a socket. The argument to the *socket* parameter must be a socket stream or a file or process stream. For a file stream, both functions return `nil`. An exception of type `type-error` is thrown for other stream types.

9.75.15 Accessor `sock-peer`

Syntax:

```
(sock-peer socket)
(set (sock-peer socket) address)
```

Description:

The `sock-peer` function retrieves the peer address has most recently been assigned to *socket*.

Sockets which are not connected initially have a peer address value of `nil`. A socket which is connected to a remote peer receives that peer's address as its `sock-peer`.

If a socket is connected to a remote peer via a successful use of the `sock-connect` function, then its `sock-peer` address is set to match that of the peer.

Sockets returned by the `sock-accept` function are connected, and have the remote endpoint address as their `sock-peer` address.

Assigning an address to a `sock-peer` form is equivalent to using `sock-set-peer` to set the address.

Implementation note: the `sock-peer` function does not use the `getpeername` C library function; the association between a stream and `sockaddr` struct is maintained by **TXR**.

9.75.16 Function `sock-set-peer`

Syntax:

```
(sock-set-peer socket address)
```

Description:

The `sock-set-peer` function stores *address* into *socket* as that socket's peer.

Subsequently, the `sock-peer` function will retrieve that address.

If *address* is not an appropriate address object in the address family of *socket*, the behavior is unspecified.

9.75.17 Function `sock-connect`

Syntax:

```
(sock-connect socket address [timeout-usec])
```

Description:

The `sock-connect` function connects a socket stream to a peer address.

The *address* argument must be a `sockaddr` object of type matching the address family of the socket.

If the operation fails, an exception of type `socket-error` is thrown. Otherwise, the function returns *socket*.

If the *timeout-usec* argument is specified, it must be a fixnum integer. It denotes a connection timeout period in microseconds. If the connection doesn't succeed within the specified timeout, an exception of type `timeout-error` is thrown.

9.75.18 Function `sock-bind`

Syntax:

```
(sock-bind socket address)
```

Description:

The `sock-bind` function binds a socket stream to a local address after enabling the socket stream's `so-reuseaddr` option.

The *address* argument must be a `sockaddr` object of type matching the address family of the socket.

If the operation fails, an exception of type `socket-error` is thrown. Otherwise, the function returns `t`.

9.75.19 Function `sock-listen`

Syntax:

```
(sock-listen socket [backlog])
```

Description:

The `sock-listen` function prepares *socket* for listening for connections. The *backlog* parameter, if specified, requires an integer argument. The default value is 16.

9.75.20 Function `sock-accept`

Syntax:

```
(sock-accept socket [mode-string [timeout-usec]])
```

Description:

The `sock-accept` function waits for a client connection on *socket*, which must have been prepared for listening for connections using `sock-bind` and `sock-listen`.

If the operation fails, an exception of type `socket-error` is thrown. Otherwise, the function returns a new socket which is connected to the remote peer.

The peer's address may be retrieved from this socket using `sock-peer`.

The `mode-string` parameter is applied to the new socket just like the similar argument in `open-socket`. It defaults to `"r+b"`.

If the *timeout-usec* argument is specified, it must be a fixnum integer. It denotes a timeout period in microseconds. If no peer connects for the specified timeout, `sock-accept` throws an exception of type `timeout-error`.

9.75.21 Variables `shut-rd`, `shut-wr` and `shut-rdwr`

Description:

The values of these variables are useful as the second argument to the `sock-shutdown` function.

9.75.22 Function `sock-shutdown`

Syntax:

```
(sock-shutdown sock [direction])
```

Description:

The `sock-shutdown` function indicates that no further communication is to take place on *socket* in the specified direction(s).

If the operation fails, an exception of type `socket-error` is thrown. Otherwise, the function returns `t`.

The `direction` parameter is one of the values given by the variables `shut-rd`, `shut-wr` or `shut-rdwr`. These values shut down communication in the read direction, write direction, or both directions, respectively.

If the argument is omitted, `sock-shutdown` defaults to closing the write direction.

Notes: shutting down is most commonly requested in the write direction, to perform a "half close". The communicating process thereby indicates that it has written all the data which it intends to write. When the shutdown action is processed on the remote end, that end is unblocked from waiting on any further data, and effectively experiences an "end of stream" condition on its own socket or socket-like endpoint, while continuing to be able to transmit data. Shutting down in the reading direction is potentially abrupt. If it is executed before an "end of stream" indication is received from a peer, it results in an abortive close.

9.75.23 Functions `sock-recv-timeout` and `sock-send-timeout`

Syntax:

```
(sock-recv-timeout sock usec)
(sock-send-timeout sock usec)
```

Description:

The `sock-recv-timeout` and `sock-send-timeout` functions configure, respectively, receive and send timeouts on socket *sock*.

The *usec* parameter specifies the value, in microseconds. It must be a `fixnum` integer.

When a receive timeout is configured on a socket, then an exception of type `timeout-error` is thrown when an input operation waits for at least `usec` microseconds without receiving input.

Similarly, when a send timeout is configured, then an exception of type `timeout-error` is thrown when an output operation waits for at least `usec` microseconds for the availability of

buffer space in the socket.

9.75.24 Variables `sol-socket`, `ipproto-ip`, `ipproto-ipv6`, `ipproto-tcp` and `ipproto-udp`

Description:

These variables represent the protocol levels of socket options and are suitable for use as the *`level`* argument of the `sock-opt` and `sock-set-opt` functions. The variables correspond to the POSIX C constants `SOL_SOCKET`, `IPPROTO_IP`, `IPPROTO_IPV6`, `IPPROTO_TCP` and `IPPROTO_UDP`.

9.75.25 Variables `so-acceptconn`, `so-broadcast`, `so-debug`, `so-dontroute`, `so-error`, `so-keepalive`, `so-linger`, `so-oobinline`, `so-rcvbuf`, `so-rcvlowat`, `so-rcvtimeo`, `so-reuseaddr`, `so-sndbuf`, `so-sndlowat`, `so-sndtimeo` and `so-type`

Description:

These variables represent socket options at the `sol-socket` protocol level and are suitable for use as the *`option`* argument of the `sock-opt` and `sock-set-opt` functions. The variables correspond to the POSIX C constants `SO_ACCEPTCONN`, `SO_BROADCAST`, `SO_DEBUG`, etc.

Note that the `sock-recv-timeout` and `sock-send-timeout` are a more convenient interface for setting the value of the `so-rcvtimeo` and `so-sndtimeo` socket options.

9.75.26 Variables `ipv6-join-group`, `ipv6-leave-group`, `ipv6-multicast-hops`, `ipv6-multicast-if`, `ipv6-multicast-loop`, `ipv6-unicast-hops` and `ipv6-v6only`

Description:

These variables represent socket options at the `ipproto-ipv6` protocol level and are suitable for use as the *`option`* argument of the `sock-opt` and `sock-set-opt` functions. The variables correspond to the POSIX C constants `IPV6_JOIN_GROUP`, `IPV6_LEAVE_GROUP`, `IPV6_MULTICAST_HOPS`, etc.

9.75.27 Variable `tcp-nodelay`

Description:

This variable represents a socket option at the `ipproto-tcp` protocol level and is suitable for use as the *`option`* argument of the `sock-opt` and `sock-set-opt` functions. The variable corresponds to the POSIX C constant `TCP_NODELAY`.

9.75.28 Accessor `sock-opt`

Syntax:

```
(sock-opt socket level option [ffi-type])
(set (sock-opt socket level option [ffi-type]) value)
```

Description:

The `sock-opt` function retrieves the value of the specified socket option, at the specified protocol level, associated with `socket`, which must be a socket stream.

The `level` argument should be one of the protocol levels `sol-socket`, `ipproto-ip`, `ipproto-ipv6`, `ipproto-tcp` and `ipproto-udp`.

The `option` argument should be one of the socket options `so-acceptconn`, `so-`

broadcast, so-debug, ..., ipv6-join-group, ..., ipv6-v6only and tcp-nodelay.

The *ffi-type* argument, which must be a compiled FFI type, specifies the type of the socket option's value. The type is most commonly int or uint, but it can be any other fixed-size type, including structs. (Variable-size types, such as C char arrays, are unsupported.) The *ffi-type* argument defaults to (ffi int).

Assigning a value to a sock-opt place is equivalent to calling sock-set-opt with that value.

Note: the sock-opt and sock-set-opt functions call the POSIX C getsockopt and setsockopt functions, respectively. Consult the POSIX specification for more information about these functions and in particular the various socket options (and the types they require).

9.75.29 Function sock-set-opt

Syntax:

```
(sock-set-opt socket level option value [ffi-type])
```

Description:

The sock-set-opt function sets the value of the specified socket option, at the specified protocol level, associated with socket, which must be a socket stream.

See the documentation of the sock-opt function for a description of the *level*, *option* and *ffi-type* arguments. Like the sock-opt function, sock-set-opt's *ffi-type* argument defaults to (ffi int).

9.75.30 Functions str-inaddr and str-in6addr

Syntax:

```
(str-inaddr address [port])
(str-in6addr address [port])
```

Description:

The str-inaddr and str-in6addr functions convert an IPv4 and IPv6 address, respectively, to textual notation which is returned as a character string. The conversion is done in conformance with RFC 5952, section 4.

IPv6 addresses representing IPv6-mapped IPv4 addresses are printed in the hybrid notation exemplified by ::ffff:192.168.1.1.

The *address* parameter must be a nonnegative integer in the appropriate range for the address type.

If the *port* number argument is supplied, it is included in the returned character string, according to the requirements in section 6 of RFC 5952 pertaining to IPv6 addresses (including IPv6-mapped IPv6 addresses) and section 3.2.3 of RFC 3986 for IPv4 addresses. In brief, IPv6 addresses with ports are expressed as [address]:port and IPv6 addresses follow the traditional address:port pattern.

9.75.31 Functions str-inaddr-net and str-in6addr-net

Syntax:

```
(str-inaddr-net address [width])
```

```
(str-in6addr-net address [width])
```

Description:

The functions `str-inaddr-net` and `str-in6addr-net` convert, respectively, IPv4 and IPv6 network prefix addresses to the "slash notation". For IPv6 addresses, the requirements of section 2.3 of RFC 4291 are implemented. For IPv4, section 3.1 of RFC 4632 is followed.

The condensed portion of the IP address is always determined by measuring the contiguous extent of all-zero bits in the least significant position of the address. For instance an IPv4 address which has at least 24 zero bits in the least significant position, so that the only nonzero bits are in the highest octet, is always condensed to a single decimal number: the value of the first octet.

If the *width* parameter is specified, then its value is incorporated into the returned textual notation as the width. No check is made whether this width large enough to span all of the nonzero bits in the address.

If *width* is omitted, then it is calculated as the number of bits in the address, excluding the contiguous all-zero bits in the least significant position: how many times the address can be shifted to the right before a 1 appears in the least significant bit.

9.75.32 Functions `inaddr-str` and `in6addr-str`

Syntax:

```
(inaddr-str string)
(in6addr-str string)
```

Description:

The `inaddr-str` and `in6addr-str` functions recover an IPv4 or IPv6 address from a textual representation. If the parse is successful, the address is returned as, respectively, a `sockaddr-in` or `sockaddr-in6` structure.

If *string* is a malformed address, due to any issue such as invalid syntax or a numeric value being out of range, an exception is thrown.

The `inaddr-str` function recognizes the dot notation consisting of four decimal numbers separated by period characters. The numbers must be in the range 0 to 255. Note: superfluous leading zeros are permitted, though this is a nonstandard extension; not all implementations of this notations support this.

A prefix may be specified in the notation as a slash followed by a decimal number, in the range 0 to 32. In this case, the integer value of the prefix appears as the `prefix` member of the returned `sockaddr-in` structure. Furthermore, the address is masked, so that any bits not included in the prefix are zero. For instance, the address `"255.255.255.255/1"` is equivalent to `"128.0.0.0"`, except that the `prefix` if the returned structure is 1 rather than 32. When a prefix is not specified, the `prefix` member of the structure retains its default value of 32. When the prefix is specified, the address part need not contain all four octets; it may contain between one and four octets. Thus, `"192.168/16"` is a valid address, equivalent to `"192.168.0.0/16"`.

A port number may be specified in the notation as a colon, followed by a decimal number in the range 0 to 65535. The integer value of this port number appears as the `port` member of the returned structure. An example of this notation is `"127.0.0.1:23"`.

A prefix and port number may both be specified; in this case the prefix must appear first, followed by the port number. For example, `"127/8:23"`.

The `in6addr-str` function recognizes the IPv6 notation consisting of 16-bit hexadecimal pieces separated by colons. If the operation is successful, it returns a `sockaddr-in6` structure. Each piece must be a value in the range 0 to FFFF. The hexadecimal digits may be any mixture of uppercase and lowercase. Leading zeros are permitted. Up to eight such pieces must be specified. If fewer pieces are specified, then the token `::` (double colon) must appear in the address exactly once. That token denotes the condensation of a sufficient number of zero-valued pieces to make eight pieces. The token must be in one of three positions: it may be the leftmost element of the address, immediately followed by a hexadecimal piece; it may be the rightmost element of the address, preceded by a hexadecimal piece; or else, it may be in the middle of the address, flanked on both sides by hexadecimal pieces.

The `in6addr-str` also recognizes the special notation for IPv6-mapped IPv4 addresses. This notation consists of the address string `"::FFFF"` which may appear in any uppercase/lowercase mixture, possibly with leading zeros, followed by an IPv4 address given in the four-octet dot notation. For example, `"::FFFF:127.0.0.1"`.

A prefix may be specified using a slash, followed by a decimal number in the range 0 to 128. The handling of the prefix is similar to that of `inaddr-str` except that pieces of the address may not be omitted. Condensing the pieces of the IPv6 address is always done by means of the `::` token, whether or not a prefix is present. Furthermore, the octets specified in the IPv6-mapped IPv4 notation must all be present, regardless of the prefix.

A port number may be specified in the notation as follows: the entire address, including any slash-separated prefix, must appear surrounded in square brackets. The closing square bracket must be followed by a colon and one or more digits denoting a decimal number in the range 0 to 65535. For instance `"[1:2:3::4/64]:1234"`.

9.75.33 Function `sockaddr-str`

Syntax:

```
(sockaddr-str string)
```

Description:

The function `sockaddr-str` analyzes the *`string`* argument to determine whether it represents a valid IPv4, IPv6 or Unix domain address. If so, it constructs an object, representing that address, of type `sockaddr-in`, `sockaddr-in` or `sockaddr-un`.

The slash prefix notation, and port numbers are handled, and represented in the returned structures accordingly.

The `sockaddr-str` function works by applying simple tests to the input, and then invoking the functions `inaddr-str` or `in6addr-str`, or constructing a `sockaddr-un` structure whose `path` slot is *`string`*.

The precise procedure followed is:

1. If *`string`* starts with `"["` then it is handled via `in6addr-str`.
2. If *`string`* starts with `"/"` then it is assumed to be the path of a Unix socket. A `sockaddr-un` structure is constructed whose `path` slot is *`string`*. This is the only case in which a structure of type `sockaddr-un` is returned.
3. If *`string`* contains `"::"` as a substring, it is handled via `in6addr-str`.
4. If *`string`* contains `"."` then it is handled via `inaddr-str`.

5. If the above tests fail, *string* is passed to `in6addr-str`, and if that call returns normally, `sockaddr-str` returns that value.
6. Otherwise, `string` is passed to `inaddr-str`.

9.75.34 Method `str-addr`

Syntax:

```
sockaddr.(str-addr)
```

Description:

A method named `str-addr` is defined for the struct types `sockaddr-in`, `sockaddr-in6` and `sockaddr-un`. It returns a text representation of the address as a string. If the `port` slot of `sockaddr-in` or `sockaddr-in6` is a nonzero integer, then it is incorporated into the text representation. Likewise if the `prefix` slot has a non-default value specifying fewer bits than the width of the address, the prefix notation is produced.

The intent is that the representations produced are suitable as input to the `sockaddr-str` function which will reproducing an address object of the same type, featuring the same `addr`, `port` and `prefix`. In the case of `sockaddr-un`, the `sockaddr-str` function will reproduce the same address only if the `path` slot is a string starting with `"/"`.

9.76 Unix Terminal Control

TXR provides access to the terminal control "termios" interfaces defined by POSIX, and some of the extensions to it in Linux. By using termios, programs can control serial devices, consoles and virtual terminals. Terminal control in POSIX revolves around a C language structure called `struct termios`. This is mirrored in a **TXR Lisp** structure also called `termios`.

Like-named **TXR Lisp** functions are provided which correspond to the C functions `tcgetattr`, `tcsetattr`, `tcsendbreak`, `tcdrain`, `tcflush` and `tcflow`.

These have somewhat different argument conventions. The TTY device is specified last, so that it can conveniently default to the `*stdin*` stream. A TTY device can be specified as either a stream object or a numeric file descriptor.

The functions `cfgetispeed`, `cfgetospeed`, `cfsetispeed` and `cfsetospeed` are not provided, because they are unnecessary. Device speed (informally, "baud rate") is specified directly as a integer value in the `termios` structure. The **TXR Lisp** termios functions automatically convert between integer values and the speed constants (like `B38400`) used by the C API.

All of the various termios-related constants are provided, including some nonstandard ones. They appear in lowercase. For instance `IGNBRK` and `PARENB` are simply known as the predefined **TXR Lisp** variables `ignbrk` and `parenb`.

9.76.1 Structure `termios`

Syntax:

```
(defstruct termios nil
  iflag oflag cflag lflag
  cc ispeed ospeed)
```

Description:

The `termios` structure represents the kernel level terminal device configuration. It holds hardware related setting such as serial line speed, parity and handshaking. It also holds software

settings like translations, and settings affecting input behaviors. The structure closely corresponds to the C language `termios` structure which exists in the POSIX API.

The `iflag`, `oflag`, `cflag` and `lflag` slots correspond to the `c_iflag`, `c_oflag`, `c_cflag` and `c_lflag` members of the C structure. They hold integer values representing bit-fields.

The `cc` slot corresponds to the `c_cc` member of the C structure. Whereas the C structure's `c_cc` member is an array of the C type `cc_t`, the `cc` slot is a vector of integers, whose values must have the same range as the `cc_t` type.

9.76.2 Variables `ignbrk`, `brkint`, `ignpar`, `parmrk`, `inpck`, `istrip`, `inlcr`, `igncr`, `icrnl`, `iuclc`, `ixon`, `ixany`, `ixoff`, `imaxbel` **and** `iutf8`

Description:

These variables specify bitmask values for the `iflag` slot of the `termios` structure. They correspond to the C language preprocessor symbols `IGNBRK`, `BRKINT`, `IGNPAR` and so forth.

The `imaxbel` and `iutf8` variables are specific to Linux and may not be present. Portable code should test for their presence with `boundp`.

The `iuclc` variable is a legacy feature not found on all systems.

Note: the `termios` methods `set-iflags` and `clear-iflags` provide a convenient means for setting and clearing combinations of these flags.

9.76.3 Variables `opost`, `olcuc`, `onlcr`, `ocrnl`, `onocr`, `onlret`, `ofill`, `ofdel`, `vtdly`, `vt0`, `vt1`, `nldly`, `nl0`, `nl1`, `crdly`, `cr0`, `cr1`, `cr2`, `cr3`, `tabdly`, `tab0`, `tab1`, `tab2`, `tab3`, `bsdly`, `bs0`, `bs1`, `ffdly`, `ff0` **and** `ff1`

Description:

These variables specify bitmask values for the `oflag` slot of the `termios` structure. They correspond to the C language preprocessor symbols `OPOST`, `OLCUC`, `ONLCR` and so forth.

The variable `ofdel` is Linux-specific. Portable programs should test for its presence using `boundp`.

The `olcuc` variable is a legacy feature not found on all systems.

Likewise, whether the following groups of symbols are present is platform-specific: `nldly`, `nl0` and `nl1`; `crdly`, `cr0`, `cr1`, `cr2` and `cr3`; `tabdly`, `tab0`, `tab1`, `tab2` and `tab3`; `bsdly`, `bs0` and `bs1`; and `ffdly`, `ff0` and `ff1`.

Note: the `termios` methods `set-oflags` and `clear-oflags` provide a convenient means for setting and clearing combinations of these flags.

9.76.4 Variables `csize`, `cs5`, `cs6`, `cs7`, `cs8`, `cstopb`, `cread`, `parenb`, `parodd`, `hupcl`, `clocal`, `cbaud`, `cbaudex`, `cmspar` **and** `crtscts`

Description:

These variables specify bitmask values for the `cflag` slot of the `termios` structure. They correspond to the C language preprocessor symbols `CSIZE`, `CS5`, `CS6` and so forth.

The following are present on Linux, and may not be available on other platforms. Portable code should test for them using `boundp`: `cbaud`, `cbaudex`, `cmspar` and `crtscts`.

Note: the `termios` methods `set-cflags` and `clear-cflags` provide a convenient means for setting and clearing combinations of these flags.

9.76.5 Variables `isig`, `icanon`, `echo`, `echoe`, `echok`, `echonl`, `noflsh`, `tostop`, `iexten`, `xcase`, `echoctl`, `echoprt`, `echoke`, `flusho`, `pendin` and `extproc`

Description:

These variables specify bitmask values for the `lflag` slot of the `termios` structure. They correspond to the C language preprocessor symbols `ISIG`, `ICANON`, `ECHO` and so forth.

The following are present on Linux, and may not be available on other platforms. Portable code should test for them using `boundp`: `iexten`, `xcase`, `echoctl`, `echoprt`, `echoke`, `flusho`, `pendin` and `extproc`.

Note: the `termios` methods `set-lflags` and `clear-lflags` provide a convenient means for setting and clearing combinations of these flags.

9.76.6 Variables `vintr`, `vquit`, `verase`, `vkill`, `veof`, `vtime`, `vmin`, `vswtc`, `vstart`, `vstop`, `vsusp`, `veol`, `vreprint`, `vdiscard`, `vwerase`, `vlnext` and `veol2`

Description:

These variables specify integer offsets into the vector stored in the `cc` slot of the `termios` structure. They correspond to the C language preprocessor symbols `VINTR`, `VQUIT`, `VERASE` and so forth.

The following are present on Linux, and may not be available on other platforms. Portable code should test for them using `boundp`: `vswtc`, `vreprint`, `vdiscard`, `vlnext` and `veol2`.

9.76.7 Variables `tcooff`, `tcoon`, `tcioff` and `tcion`

Description:

These variables hold integer values suitable as the *`action`* argument of the `tcflow` function. They correspond to the C language preprocessor symbols `TCOOFF`, `TCOON`, `TCIOFF` and `TCION`.

9.76.8 Variables `tciflush`, `tcoflush` and `tcioflush`

Description:

These variables hold integer values suitable as the *`queue`* argument of the `tcflush` function. They correspond to the C language preprocessor symbols `TCIFLUSH`, `TCOFLUSH` and `TCIOFLUSH`.

9.76.9 Variables `tcsanow`, `tcsadrain` and `tcsaflush`

Description:

These variables hold integer values suitable as the *`actions`* argument of the `tcsetattr` function. They correspond to the C language preprocessor symbols `TCSANOW`, `TCSADRAIN` and `TCSAFLUSH`.

9.76.10 Functions `tcgetattr` and `tcsetattr`

Syntax:

```
(tcgetattr [device])
(tcsetattr termios [actions [device]])
```

Description:

The `tcgetattr` and `tcsetattr` functions, respectively, retrieve and install the configuration of the terminal driver associated with the specified device.

These functions are wrappers for the like-named POSIX C library functions, but with different argument conventions, and operating using a **TXR Lisp** structure.

The `tcgetattr` function, if successful, returns a new instance of the `termios` structure.

The `tcsetattr` function requires an instance of a `termios` structure as an argument to its *`termios`* parameter.

A program may alter the settings of a terminal device by retrieving them using `tcgetattr`, manipulating the structure returned by this function, and then using `tcsetattr` to install the modified structure into the device.

The *`actions`* argument of `tcsetattr` may be given as the value of one of the variables `tcsanow`, `tcsadrain` or `tcsaflush`. If it is omitted, the default is `tcsadrain`.

If an argument is given for *`device`* it must be either a stream, or an integer file descriptor. In either case, it is expected to be associated with a terminal (TTY) device.

If the argument is omitted, it defaults to the stream currently stored in the `*stdin*` stream special variable, expected to be associated with a terminal device.

Notes:

The C `termios` structure usually does not have members for representing the input and output speed. **TXR Lisp** does not use such members, in any case, even if they are present. The speeds are encoded in the `cc_iflag` and `cc_lflag` bitmasks. When retrieving the settings, the `tcgetattr` function uses the POSIX functions `cfgetispeed` and `cfgetospeed` to retrieve the speed values from the C structure. These values are installed as the `ispeed` and `ospeed` slots of the Lisp structure. A reverse conversion takes place when setting are installed using `tcsetattr`: the speed values are taken from the slots, and installed into the C structure using `cfsetispeed` and `cfsetospeed` before the structure is passed to the C `tcsetattr` function.

On Linux, TTY devices do not have a separate input and output speed. The C `termios` structure stores only one speed which is taken as both the input and output speed, with a special exception. The input speed may be programmed as zero. In that case, it is independently represented. speed may be programmed as zero.

9.76.11 Function `tcsendbreak`

Syntax:

```
(tcsendbreak [duration [device]])
```

Description:

The `tcsendbreak` function generates a break signal on serial devices. The *duration* parameter specifies the length of the break signal in milliseconds. If the argument is omitted, the value 500 is used.

The *device* parameter is exactly the same as that of the `tcsetattr` function.

9.76.12 Function `tcdrain`

Syntax:

```
(tcdrain [device])
```

Description:

The `tcdrain` function waits until all queued output on a terminal device has been transmitted. It is a direct wrapper for the like-named POSIX C function.

The *device* parameter is exactly the same as that of the `tcsetattr` function.

9.76.13 Function `tcflush`

Syntax:

```
(tcflush queue [device])
```

Description:

The `tcflush` function discards either untransmitted output data, or received and yet unread input data, depending on the value of the *queue* argument. It is a direct wrapper for the like-named POSIX C function.

The *queue* argument should be the value of one of the variables `tciflush`, `tcoflush` and `tcioflush`, which specify the flushing of input data, output data or both.

The *device* parameter is exactly the same as that of the `tcsetattr` function.

9.76.14 Function `tcflow`

Syntax:

```
(tcflow action [device])
```

Description:

The `tcflow` function provides bidirectional flow control on the specified terminal device. It is a direct wrapper for the like-named POSIX C function.

The *action* argument should be the value of one of the variables `tcooff`, `tcoon`, `tcioff` and `tcion`.

The *device* parameter is exactly the same as that of the `tcsetattr` function.

9.76.15 Methods `set-iflags`, `set-oflags`, `set-cflags`, `set-lflags`, `clear-iflags`, `clear-oflags`, `clear-cflags` and `clear-lflags`

Syntax:

```
termios.(set-iflags flags*)
termios.(set-oflags flags*)
```

```
termios.(set-cflags flags*)
termios.(set-lflags flags*)
termios.(clear-iflags flags*)
termios.(clear-oflags flags*)
termios.(clear-cflags flags*)
termios.(clear-lflags flags*)
```

Description:

These methods of the `termios` structure set or clear multiple flags of the four bitmask flag fields.

The *flags* arguments specify zero or more integer values. These values are combined together bitwise, as if by the `logior` function to form a single effective mask. If there are no *flags* arguments, then the effective mask is zero.

The `set-iflags` method sets, in the `iflag` slot of the *termios* structure, all of the bits which are set in the effective mask. That is to say, the effective mask is combined with the value in `iflag` by a `logior` operation, and the result is stored back into `iflag`. Similarly, the `set-oflags`, `set-cflags` and `set-lflags` methods operate on the `oflag`, `cflag` and `lflag` slots of the structure.

The `clear-iflags` method clears, in the `iflag` slot of the *termios* structure, all of the bits which are set in the effective mask. That is to say, the effective mask is bitwise inverted as if by the `lognot` function, and then combined with the existing value of the `iflag` slot using `logand`. The resulting value is stored back into the `iflag` slot. Similarly, the `clear-oflags`, `clear-cflags` and `clear-lflags` methods operate on the `oflag`, `cflag` and `lflag` slots of the structure.

Note: the methods `go-raw`, `go-cbreak` and `go-canon` are provided for changing the settings to raw, "cbreak" and canonical mode. These methods should be preferred to directly manipulating the flag and `cc` slots.

Example

In this example, `tio` is assumed to be a variable holding an instance of a `termios` struct:

```
;; clear the ignbrk, brkint, and various other flags:
tio.(clear-iflags ignbrk brkint parmrk istrip
                  inlcr igncr icrnl ixon)

;; set the csize and parenb flags:
tio.(set-cflags csize parenb)
```

9.76.16 Methods `go-raw` and `go-cbreak`

Syntax:

```
termios.(go-raw)
termios.(go-cbreak)
```

Description:

The `go-raw` and `go-cbreak` methods of the `termios` structure manipulate the flag slots, as well as certain elements of the `cc` slot, in order to prepare the terminal settings for, respectively, "raw" and "cbreak" mode, described below.

Note that manipulating the `termios` structure doesn't actually put these settings into effect in the

terminal device; the settings represented by the structure must be installed into the device using `tcsetattr`. There is no way to reverse the effect of these methods. To precisely restore the previous terminal settings, the program should retain a copy of the original `termios` structure.

"Raw" mode refers to a configuration of the terminal device driver in which input and output is passed transparently and without accumulation, conversion or interpretation. Input isn't buffered into lines; as soon as a single byte is received, it is available to the program. No special input characters such as commands for generating an interrupt or process suspend request are processed by the terminal driver; all characters are treated as input data. Input isn't echoed; the only output which takes place is that generated by program output requests to the device.

"Cbreak" mode is named after a concept and function in the "curses" terminal control library. It refers to a configuration of the terminal device driver which is less transparent than "raw" mode. Input isn't buffered into lines, and line editing commands are ordinary input characters, allowing character-at-a-time input. However, most input translations are preserved, except that the conversion of CR characters to NL is disabled. The signal-generating characters are processed in this mode. This latter feature of the configuration is the likely inspiration for the word "cbreak". Unless otherwise configured, the interrupt character corresponds to the Ctrl-C key, and "break" is another term for an interactive interruption.

9.76.17 Methods `string-encode` and `string-decode`

Syntax:

```
termios.(string-encode)
termios.(string-decode string)
```

Description:

The `string-encode` method converts the terminal state stored in a `termios` structure into a textual format, returning that representation as a character string.

The `string-decode` method parses the character representation produced by `string-encode` and populates the *`termios`* structure with the settings are encoded in that string.

If a string is passed to `string-decode` which wasn't produced by `string-encode`, the behavior is unspecified. An exception may or may not be thrown, and the contents of *`termios`* may or may not be affected.

Note: the textual representation produced by `string-encode` is intended to be identical to that produced by the `-g` option of the GNU Coreutils version of the `stty` utility, on the same platform. That is to say, the output of `stty -g` may be used as input into `string-decode`, and the output of `string-encode` may be used as an argument to `stty`.

9.77 Unix System Identification

9.77.1 Structure `utsname`

Syntax:

```
(defstruct utsname nil
  sysname nodename release
  version machine domainname)
```

Description:

The `utsname` structure corresponds to the POSIX structure of the same name. An instance of this structure is returned by the `uname` function.

9.77.2 Function `uname`

Syntax:

```
(uname)
```

Description:

The `uname` function corresponds to the POSIX function of the same name. It returns an instance of the `utsname` structure. Each slot of the returned structure is initialized with a character string that identifies the corresponding attribute of the host system.

The host system might not support the reporting of the NIS domain name. In this case, the `domainname` slot of the returned `utsname` structure will have the value `nil`.

9.78 Unix Resource Limits

9.78.1 Structure `rlim`

Syntax:

```
(defstruct rlim nil
  cur max)
```

Description:

The `rlim` structure is required by the functions `getrlimit` and `setrlimit`. It is analogous to the C structure by the same name described in POSIX.

9.78.2 Variables `rlim-saved-max`, `rlim-saved-cur` and `rlim-infinity`

Description:

These variables correspond to the POSIX constants `RLIM_SAVED_MAX`, `RLIM_SAVED_CUR` and `RLIM_INFINITY`. They have the same values, and are suitable as slot values of the `rlim` structure.

Variables @, rlimit-core @, rlimit-cpu @, rlimit-data @, rlimit-fsize @, rlimit-nofile @ rlimit-stack and @ rlimit-as

Description:

These variables correspond to the POSIX constants `RLIMIT_CORE`, `RLIMIT_CPU`, `RLIMIT_DATA` and so forth.

9.78.3 Functions `getrlimit` and `setrlimit`

Syntax:

```
(getrlimit resource [rlim])
(setrlimit resource rlim)
```

Description:

The `getrlimit` function retrieves information about the limits imposed for a particular parameter indicated by the *`resource`* integer.

The `setrlimit` function changes the limit information for a resource parameter.

The *`resource`* parameter is the value of one of the variables `rlimit-core`, `rlimit-cpu`, `rlimit-data` and so forth.

The *rlim* argument is a structure of type `rlim`. If this argument is given to the `getrlimit` function, then it fills in that structure with the retrieved parameters. Otherwise it allocates a new structure and fills that one. In either situation, the filled structure is returned, if the underlying call to the host operating system is successful.

In the case of `setrlimit`, the `rlim` object must have non-negative integer values which are in the range of the platform's `rlim_t` type.

If the underlying system call fails, then these functions throw an exception. In the successful case, the `getrlimit` function returns the `rlim` structure, and `setrlimit` returns `t`.

Further information about resource limits is available in the POSIX standard and platform documentation.

9.79 Unix Memory Mapping

The **TXR Lisp** interface to the POSIX `mmap` family of functions is based around the `carray` type. The `mmap` function returns a special variant of a `carray` object which keeps track of the memory mapping. When such an object becomes unreachable and is reclaimed by garbage collection, the mapping is automatically unmapped.

In addition to `mmap`, the functions `munmap`, `mprotect`, `madvise` and `msync` are provided, all taking a `carray` as their leftmost argument.

The **TXR Lisp** functions do not strictly follow the argument conventions of the same-named, corresponding POSIX functions. Adjustments which are likely to be defaulted are moved to the right. For instance, the `msync` operation is often applied to the entire memory mapping. Therefore, the first argument is the `carray` object which keeps track of the mapping. The second argument specifies the flags to be applied, which constitute the last argument of the underlying POSIX function. The remaining two arguments are the size and offset. If these are omitted, then `msync` applies to the entire region, whose address and size are known to the `carray` object.

Cautionary note: misuse of `mmap` and related functions can easily cause the **TXR** image to receive a fatal signal due to a bad memory access. Care must be taken to prevent such a situation, or else to catch such signals and recover.

9.79.1 Function `mmap`

Syntax:

```
(mmap ffi-type length prot flags
      [source [offset [addr]]])
```

Description:

The `mmap` function provides access to the same-named POSIX platform function for creating memory mappings. The POSIX function can be used for creating virtual memory views of files and special devices. Views can be read-only, and they can be mutable. They can be in such a way that changes appear only in the mapping itself, or in such a way that the changes are actually propagated to the mapped object itself. Mappings can be shared among processes, providing a shared memory mechanism: for instance, if `fork` is called, any `map-shared` mappings created by the parent are shared with the child: the child process does not get a copy of a shared mapping, but a reference to it. The function can also be used simply for allocating memory: on some platforms, the POSIX `mmap` function is used as the basis for the `malloc` function. It behaves as a pure allocator when asked to create a mapping which is private, and anonymous (not backed by an object).

The **TXR Lisp** mmap function is integrated with the carray type and the FFI type system. A mapping returned by mmap is represented by a carray object.

The required *ffi-type* argument specifies the element type of the array; it must be a compiled FFI type. Note: this may be produced by the ffi macro. For instance, the type int may be specified using the expression (ffi int). The type must be a complete type suitable as the element type of an array; a type with a zero size such as void is invalid.

The *length* argument specifies the requested length of the mapping in bytes. Note that mmap allocates or configures virtual memory pages, not bytes. Internally to the system, the *length* argument is converted to a number of pages. If it specifies a fractional number of pages, it is rounded up. For instance, if the page size is 4096 bytes, and *length* is specified as 5000, it will be internally rounded up to 8192. The returned **TXR Lisp** carray object is oblivious to this padding: it works with the given 5000-byte size. Note: the page-size variable holds the system's page size. However, by the use of mmap extensions, it is possible for individual mappings to have their own page size. Mixed page size virtual memory systems exist.

The mmap function determines the number of elements in the array by dividing the *length* by the size of *type*, using a division that truncates toward zero. The returned carray shall have that many elements. If the division is inexact, it means that some bytes from the underlying memory mapping are unused, even if length is a multiple of the page size.

The required *prot* argument must be some bitwise combination of the portable values prot-read, prot-write and prot-exec. Additional system-specific prot- values may be available also for specifying additional properties. If *prot* is specified as zero, then the mapping, if successfully created, may be inaccessible: prot-read must be present to ensure read access, and prot-write to ensure write access.

The *flags* argument is a bitwise combination of values given by various map- variables. At the very least, it must contain exactly one of map-shared or map-private, to request a shared or private mapping, respectively. If a mapping is requested which is neither shared nor private, the underlying POSIX function will likely fail. If a *source* is specified, indicating a filesystem object to be mapped, the map-anon flag must be omitted. Vice versa, if *source* is not specified, this means that the mapping will be anonymous. In this situation, the map-anon flag must be present.

Note: in the context of mmap, "anonymous" means "not associated with a filesystem object referenced by a descriptor". It does not mean "without a name", but refers to a pure memory allocation from virtual memory. Memory maps do not have a name, whether anonymous or not. Moreover, the filesystem object associated with a memory map itself does not necessarily have a name. An open file that has been deleted from the directory structure is anonymous, yet a memory mapping can be created using its descriptor, and that mapping is not "anonymous".

The *offset* argument is used with a non-anonymous mapping. It specifies that the mapping doesn't begin at the start of the file or file-like object, but rather at the specified offset. The offset may not be an arbitrary integer; it must be a multiple of the page size. Unless certain nonportable *flags* are used to specify an alternative page size, the value of the page-size variable may be relied upon to indicate the page size. If an *offset* is specified for an anonymous mapping, with a nonzero value, the underlying POSIX function may indicate failure.

If the *length* and *offset* values cause one or more pages to be mapped which are beyond the end of the file, then accessing those pages may produce a signal which is fatal if not handled.

The *addr* argument is used for specifying the address in conjunction with the map-fixed flag.

Possibly, certain nonportable values in the *flags* argument may similarly require *addr*. If no bit is present in *flags* which requires *addr*, then *addr* should either not be specified, or specified as zero. A nonzero value of *addr* must be a multiple of the page size.

The mmap function returns a carray object if successful. Upon failure, an exception derived from error is thrown.

Note: when a carray object returned by mmap is identified by the garbage collector as unreachable, and reclaimed, the memory mapping is unmapped. The munmap function can be invoked on the carray to release the mapping before the object becomes garbage. The carray-free function cannot be used on a mapped carray.

9.79.2 Function munmap

Syntax:

```
(munmap carray)
```

Description:

The munmap function releases the memory mapping tracked by *carray*, which must be an object previously returned by mmap. An exception is thrown if the object is any other kind of carray.

Note: the memory mapping is released by means of the same-named POSIX function. No provision is made for selectively unmapping the pages of a mapping; the entire mapping associated with a *carray* is removed.

When the memory mapping is released, munmap returns t. Thereafter, the *carray*'s contents may no longer be accessed, subject to error exceptions being thrown.

If munmap is called again on a carray on which it had previously been successfully called, the additional calls return nil.

9.79.3 Functions mprotect, madvise and msync

Syntax:

```
(mprotect carray prot [offset [size]])
(madvise carray advice [offset [size]])
(msync carray flags [offset [size]])
```

Description:

The functions mprotect, madvise and msync perform various operations and adjustments on a memory mapping, using the same-named, corresponding POSIX functions.

All functions follow the same argument conventions with regard to the *carray* argument and the optional *offset* and *size* arguments. The respective second arguments *prot*, *advice* and *flags* are all integers. Of these, *prot* and *flags* are bitmapped flags, whereas *advice* specifies an enumerated command.

The *prot* argument is a bitwise combination of prot- values such as prot-read, prot-write and prot-exec. The mprotect function adjusts the protection bits of the mapping accordingly.

The *advice* argument of madvise should specify one of the following portable values, or else some system-specific nonportable madv- value: madv-normal, madv-random, madv-

sequential, madv-willneed or madv-dontneed.

The *flags* argument of msync should specify exactly one of the values ms-async and ms-sync. Additional ms- values such as ms-invalidate may be combined in.

If *offset* and *size* are omitted, they default to zero and the size of the entire mapping, respectively, so the operation applies to the entire mapping.

If only *size* is specified, it must not exceed the mapping size, or an error exception is thrown. The *offset* argument defaults to zero.

If only *offset* is specified, it must not exceed the length of the mapping, or else an error exception is thrown. The size is calculated as the difference between the offset and the length. It may be zero.

If both *offset* and *size* are specified, they must not specify a region any portion of which lies outside the mapping. If *size* is zero, *offset* may be equal to the length of the mapping.

The *offset* must be a multiple of the page size, or else the operation will fail, since these functions work with virtual memory pages, and not individual bytes. The *length* is adjusted by the system to a multiple of the applicable page size, as noted in the description of mmap.

When any of these three functions succeeds, it returns t. Otherwise, it throws an exception.

9.79.4 Variables map-shared, map-private, map-anon and map-fixed

Description:

The integer values of these variables are bitmasks, intended to be combined with logior to prepare a value for the *flags* argument of mmap.

Additional nonportable, system-dependent map- variables may be available. Their names are derived by taking the MAP_-prefixed symbol from the platform header file, converting it to lowercase and replacing underscores by hyphen characters. Any such variable which exists, but has a value of zero, is present only for compatibility with other systems. For instance map-huge-shift may be present in non-Linux ports of **TXR**, but with a zero value; it has a nonzero value on Linux systems to which it is specific. Applications critically relying on certain flags should test the corresponding variables for nonzero to make sure they are actually available.

9.79.5 Variables prot-none, prot-read, prot-write and prot-exec

Description:

The integer values of these variable are bitmasks, intended to be combined with logior to prepare a value for the *prot* argument of mmap and mprotect.

Additional nonportable, system-dependent prot- variables may be available. Their names are derived by taking the PROT_-prefixed symbol from the platform header file, converting it to lowercase and replacing underscores by hyphen characters. Any such variable which exists, but has a value of zero, is present only for compatibility with other systems.

9.79.6 Variables madv-normal, madv-random, madv-sequential, madv-willneed and madv-dontneed

Description:

The integer values of these variable are bitmasks, intended to be combined with `logior` to prepare a value for the *`advice`* argument of the `madvise` function.

Additional nonportable, system-dependent `madv-` variables may be available. Their names are derived by taking the `MADV_`-prefixed symbol from the platform header file, converting it to lower case and replacing underscores by hyphen characters. Any such variable which exists, but has a value of zero, is present only for compatibility with another system.

9.79.7 Variables `ms-async`, `ms-sync` and `ms-invalidate`

Description:

The integer values of these variable are bitmasks, intended to be combined with `logior` to prepare a value for the *`advice`* argument of the `msync` function.

As described under `msync`, exactly one of `ms-async` and `ms-sync` should be present; `ms-invalidate` is optional.

9.80 Web Programming Support

9.80.1 Functions `url-encode` and `url-decode`

Syntax:

```
(url-encode string [space-plus-p])
(url-decode string [space-plus-p])
```

Description:

These functions convert character strings to and from a form which is suitable for embedding into the request portions of URL syntax.

Encoding a string for URL use means identifying in it certain characters that might have a special meaning in the URL syntax and representing it using "percent encoding": the percent character, followed by the ASCII value of the character. Spaces and control characters are also encoded, as are all byte values greater than or equal to 127 (7F hex). The printable ASCII characters which are percent-encoded consist of this set:

```
:/?#[]@!$&'()*+,;=%
```

More generally, strings can consists of Unicode characters, but the URL encoding consists only of printable ASCII characters. Unicode characters in the original string are encoded by expanding into UTF-8, and applying percent-encoding the UTF-8 bytes, which are all in the range `\x80-\xFF`.

Decoding is the reverse process: reconstituting the UTF-8 byte sequence specified by the URL-encoding, and then decoding the UTF-8 sequence into the string of Unicode characters.

There is an additional complication: whether or not to encode spaces as plus, and to decode plus characters to spaces. In encoding, if spaces are not encoded to the plus character, then they are encoded as `%20`, since spaces are reserved characters that must be encoded. In decoding, if plus characters are not decoded to spaces, then they are left alone: they become plus characters in the decoded string.

The `url-encode` function performs the encoding process. If the `space-plus-p` argument is omitted or specified as `nil`, then spaces are encoded as `%20`. If the argument is a value other than

nil, then spaces are encoded as the character + (plus).

The url-decode function performs the decoding process. If the space-plus-p argument is omitted or specified as nil, then + (plus) characters in the encoded data are retained as + characters in the decoded strings. Otherwise, plus characters are converted to spaces.

9.80.2 Functions html-encode, html-encode* and html-decode

Syntax:

```
(html-encode text-string)
(html-decode html-string)
```

Description:

The html-encode and html-decode functions convert between an HTML and raw representation of text.

The html-encode function returns a string which is based on the content of *text-string*, but in which all characters which have special meaning in HTML have been replaced by HTML codes for representing those characters literally. The returned string is the HTML-encoded verbatim representation of *text-string*.

The html-decode function converts *html-string*, which may contain HTML character encodings, into a string which contains the actual characters represented by those encodings.

The function composition (html-decode (html-encode text)) returns a string which is equal to text.

The reverse composition (html-encode (html-decode html)) does not necessarily return a string equal to html.

For instance if html is the string " Hello, world! ", then html-decode produces " Hello, world! ". From this, html-encode produces "<p>Hello, world!</p>".

The html-encode* function is similar to html-encode except that it does not encode the single and double quote characters (ASCII 39 and 34, respectively). Text prepared by this function may not be suitable for insertion into a HTML template, depending on the context of its insertion. It is suitable as text placed between tags but not necessarily as tag attribute material.

9.80.3 Functions base64-encode, base64-decode and base64-decode-buf

Syntax:

```
(base64-encode [string | buf] [column-width])
(base64-decode string)
(base64-decode-buf string)
```

Description:

The base64-encode function converts the UTF-8 representation of *string*, or the contents of *buf*, to Base64 and returns that representation as a string. The Base64 encoding is described in RFC 4648, section 5.

The second argument must either be a character string, or a buffer object.

The base64-decode functions performs the opposite conversion; it extracts the bytes encoded

in a Base64 string, and decodes them as UTF-8 to return a character string.

The `base64-decode-buf` extracts the bytes encoded in a Base64 string, and returns a new buffer object containing these bytes.

The Base64 encoding divides the UTF-8 representation of *string* or the bytes contained in *buf* into groups of six bits, each representing the values 0 to 63. Each value is then mapped to the characters `A` to `Z`, `a` to `z`, the digits `0` to `9` and the characters + and `/`. One or two consecutive occurrences of the character = are added as padding so that the number of non-whitespace characters is divisible by four. These characters map to the code 0, but are understood not to contribute to the length of the encoded message. The `base64-encode` function enforces this convention, but `base64-decode` doesn't require these padding characters.

Base64-encoding an empty string or zero-length buffer results in an empty string.

If the *column-width* argument is passed to `base64-encode`, then the Base64 encoded string, unless empty, contains newline characters, which divide it into lines which are *column-width* long, except possibly for the last line.

9.80.4 Functions `base64-stream-enc` and `base64-stream-dec`

Syntax:

```
(base64-stream-enc out in [nbytes [column-width]])
(base64-stream-dec out in)
```

Description:

The `base64-stream-enc` and `base64-stream-dec` perform, respectively, bulk Base64 encoding and decoding between streams. This format is described in RFC 4648, section 5.

The *in* and *out* arguments must be stream objects. The *out* stream must support output. In the decode operation, it must support byte output. The *in* stream must support input. In the encode operation it must support byte input.

The `base64-stream-enc` function reads a sequence of bytes from the *in* stream and writes characters to the *out* stream comprising the Base64 encoding of that sequence. If the *nbytes* argument is specified, it must be a nonnegative integer. At most *nbytes* bytes will be read from the *in* stream. If *nbytes* is omitted, then the operation will read from the *in* stream without limit, until that stream indicates that no more bytes are available.

The optional *column-with* argument influences the formatting of Base64 output, in the same manner as documented for the `base64-encode` function.

The `base64-stream-dec` function reads the characters of a Base64 encoding from the *in* stream and writes the corresponding byte sequence to the *out* stream. It keeps reading and decoding until it encounters the end of the stream, or a character not used in Base64: a character that is not whitespace according to `chr-isspace`, isn't any of the Base64 coding characters (not an alphanumeric character, and not one of the characters +, `/` or =. If the function stops due to a non-Base64 character, that character is pushed back into the *in* stream.

The `base64-stream-enc` function returns the number of bytes encoded; the `base64-stream-dec` function returns the number of bytes decoded.

9.80.5 Functions `base64url-encode`, `base64url-decode` and `base64url-decode-buf`

Syntax:

```
(base64url-encode [string | buf] [column-width])
(base64url-decode string)
(base64url-decode-buf string)
```

Description:

The `base64url-encode`, `base64url-decode` and `base64url-decode-buf` functions conform, in nearly every respect, to the descriptions of, respectively, `base64-encode`, `base64-decode` and `base64-decode-buf`. The difference is that these functions use the encoding described in section 6 of RFC 4648, rather than section 5. This means that, in the encoding alphabet, instead of the symbols + (plus) and / (slash) the symbols − (minus) and _ (underline) are used.

9.80.6 Functions `base64url-stream-enc` and `base64url-stream-dec`

Syntax:

```
(base64url-stream-enc out in [nbytes [column-width]])
(base64url-stream-dec out in)
```

Description:

The `base64url-stream-enc` and `base64url-stream-dec` functions conform, in nearly every respect, to the descriptions of, respectively, `base64-stream-enc` and `base64-stream-dec`. The difference is that these functions use the encoding described in section 6 of RFC 4648, rather than section 5. This means that, in the encoding alphabet, instead of the symbols + (plus) and / (slash) the symbols − (minus) and _ (underline) are used.

9.81 Filter Module

The filter module provides a trie (pronounced "try") data structure, which is suitable for representing dictionaries for efficient filtering. Dictionaries are unordered collections of keys, which are strings, which have associated values, which are also strings. A trie can be used to filter text, such that keys appearing in the text are replaced by the corresponding values. A trie supports this filtering operation by providing an efficient prefix-based lookup method which only looks at each input character once, and which does not require knowledge of the length of the key in advance.

9.81.1 Function `make-trie`

Syntax:

```
(make-trie)
```

Description:

The `make-trie` function creates an empty trie. There is no special data type for a trie; a trie is some existing type such as a hash table.

9.81.2 Function `trie-add`

Syntax:

```
(trie-add trie key value)
```

Description:

The `trie-add` function adds the string *key* to the trie, associating it with *value*. If *key* already exists in *trie*, then the value is updated with *value*.

The *trie* must not have been compressed with *trie-compress*.

A trie can contain keys which are prefixes of other keys. For instance it can contain "dog" and "dogma". When a trie is used for matching and substitution, the longest match is used. If the input presents the text "doggy", then the match is "dog". If the input is "dogmatic", then "dogma" matches.

9.81.3 Function trie-compress

Syntax:

```
(trie-compress trie)
```

Description:

The trie-compress function changes the representation of *trie* to a representation which occupies less space and supports faster lookups. The new representation is returned.

The compressed representation of a trie does not support the trie-add function.

The trie-compress function destructively manipulates *trie*, and may return an object that is the same object as trie, or it may return a different object, while at the same time still modifying the internals of *trie*. Consequently, the program should not retain the input object trie, but use the returned object in its place.

9.81.4 Function trie-lookup-begin

Syntax:

```
(trie-lookup-begin trie)
```

Description:

The trie-lookup-begin function returns a context object for performing an open-coded lookup traversal of a trie. The *tri* argument is expected to be a trie that was created by the make-trie function.

9.81.5 Function trie-lookup-feed-char

Syntax:

```
(trie-lookup-feed-char trie-context char)
```

Description:

The trie-lookup-feed-char function performs a one character step in a trie lookup. The *trie-context* argument must be a trie context returned by *trie-lookup-begin*, or by some previous call to trie-lookup-feed-char. The *char* argument is the next character to match.

If the lookup is successful (the match through the trie can continue with the given character) then a new trie context object is returned. The old trie context remains valid.

If the lookup is unsuccessful, nil is returned.

Note: determining whether a given string is stored in a trie can be performed looking up every character of the string successively with trie-lookup-feed-char, using the newly returned context for each successive operation. If every character is found, it means that either that exact string is found in the trie, or a prefix. The ambiguity can be resolved by testing whether the trie has a value at the last node using trie-value-at. For instance, if "catalog" is inserted

into an empty trie with value `"foo"`, then `"cat"` will look up successfully, being a prefix of `"catalog"`; however, the value at `"cat"` is `nil`, indicating that `"cat"` is only a prefix of one or more entries in the trie.

9.81.6 Function `trie-value-at`

Syntax:

```
(trie-value-at trie-context)
```

Description:

The `trie-value-at` function returns the value stored at the node in in the trie given by *`trie-context`*. Nodes which have not been given a value hold the value `nil`.

9.81.7 Function `filter-string-tree`

Syntax:

```
(filter-string-tree filter obj)
```

Description:

The `filter-string-tree` function returns a tree structure similar to *`obj`* in which all of the string atoms have been filtered through *`filter`*.

The *`obj`* argument is a string-tree structure: either the symbol `nil`, denoting an empty structure; a string; or a list of tree structures. If *`obj`* is `nil`, then `filter-string-tree` returns `nil`.

The *`filter`* argument is a filter: it is either a trie, a function, or nil. If *`filter`* is `nil`, then `filter-string-trie` just returns *`obj`*.

If *`filter`* is a function, it must be a function that can be called with one argument. The strings of the string tree are filtered by passing each one into the function and substituting the return value into the corresponding place in the returned structure.

Otherwise if *`filter`* is a trie, then this trie is used for filtering, the string elements similarly to a function. For each string, a new string is returned in which occurrences of the keys in the trie are replaced by the values in the trie.

9.81.8 Function `filter-equal`

Syntax:

```
(filter-equal filter-1 filter-2 obj-1 obj-2)
```

Description:

The `filter-equal` function tests whether two string trees are equal under the given filters.

The precise semantics can be given by this expression:

```
(equal (filter-string-tree filter-1 obj-1)
       (filter-string-tree filter-2 obj-2))
```

The string tree *`obj-1`* is filtered through *`filter-1`*, as if by the `filter-string-tree` function, and similarly, *`obj-2`* is filtered through *`filter-2`*. The resulting structures are compared using `equal`, and the result of that is returned.

9.81.9 Function `regex-from-trie`

Syntax:

```
(regex-from-trie trie)
```

Description:

The `regex-from-trie` function returns a representation of *`trie`* as regular-expression abstract syntax, suitable for processing by the `regex-compile` function.

The values stored in the trie nodes are not represented in the regular expression.

The *`trie`* may be one that has been compressed via `trie-compress`; in fact, a compressed *`trie`* results in more compact syntax.

Note: this function is useful for creating a compact, prefix-compressed regular expression which matches a list of strings.

9.81.10 Special variable `*filters*`

Description:

The `*filters*` special variable holds a hash table which associates symbols with filters. This hash table defines the named filters used in the **TXR** pattern language. The names are the hash-table keys, and filter objects are the values. Filter objects are one of three representations. The value `nil` represents a null filter, which performs no filtering, passing the input string through. A filter object may be a raw or compressed trie. It may also be a Lisp function, which must be callable with one argument of string type, and must return a string.

The application may define new filters by associating symbolic keys in `*filters*` with values which conform to the above representation of filters.

The behavior is unspecified if any of the predefined filters are removed or redefined, and are subsequently used, or if the `*filters*` variable is replaced or rebound with a hash-table value which omits those keys, or associates them with different values.

Note that functions `html-encode`, `html-encode*` and `html-decode` use, respectively, the HTML-related `:tohtml`, `:tohtml*` and `:fromhtml`.

9.82 Access To TXR Pattern Language From Lisp

It is useful to be able to invoke the abilities of the **TXR** pattern Language from **TXR Lisp**. An interface for doing this provided in the form of the `match-fun` function, which is used for invoking a **TXR** pattern function.

The `match-fun` function has a cumbersome interface which requires the **TXR Lisp** program to explicitly deal with the variable bindings emerging from the pattern match in the form of an association list.

To make it the interface easier to use, **TXR** provides the macros `txr-if`, `txr-when` and `txr-case`.

9.82.1 Function `match-fun`

Syntax:

```
(match-fun name args [input [files]])
```

Description:

The `match-fun` function invokes a **TXR** pattern function whose name is given by *`name`*, which must be a symbol.

The *`args`* argument is a list of expressions. The expressions may be symbols which will be interpreted as pattern variables, and may be bound or unbound. If they are not symbols, then they are treated as expressions (of the pattern language, not **TXR Lisp**) and evaluated accordingly.

The optional *`input`* argument is an object of one of several types. It may be a stream, character string or list of strings. If it is a string, then it is converted to a list containing that string. A list of strings represents zero or more lines of text to be processed. If the *`input`* argument is omitted, then it defaults to `nil`, interpreted as an empty list of lines.

The *`files`* argument is a list of filename specifications, which follow the same conventions as files given on the **TXR** command line. If the pattern function uses the `@(next)` directive, it can process these additional files. If this argument is omitted, it defaults to `nil`.

The `match-fun` function's return value falls into three cases. If there is a match failure, it returns `nil`. Otherwise it returns a cons cell. The `car` field of the cons cell holds the list of captured bindings. The `cdr` of the cons cell is one of two values. If the entire input was processed, the cdr field holds the symbol `t`. Otherwise it holds another cons cell whose `car` is the remainder of the list of lines which were not matched, and whose `cdr` is the line number.

Example:

```
@(define foo (x y))
@x:@y
@line
@(end)
@(do
   (format t "~s\n"
             (match-fun 'foo '(a b)
                        '("alpha:beta" "gamma" "omega") nil)))

Output:
(((a . "alpha") (b . "beta")) ("omega") . 3)
```

In the above example, the pattern function `foo` is called with arguments `(a b)`. These are unbound variables, so they correspond to parameters `x` and `y` of the function. If `x` and `y` get bound, those values propagate to `a` and `b`. The data being matched consists of the lines `"alpha:beta"`, `"gamma"` and `"omega"`. Inside `foo`, `x` and `y` bind to `"alpha"` and `"beta"`, and then the line variable binds to `"gamma"`. The input stream is left with `"omega"`.

Hence, the return value consists of the bindings of `x` and `y` transferred to `a` and `b`, and the second cons cell which gives information about the rest of the stream: it is the part starting at `"omega"`, which is line 3. Note that the binding for the `line` variable does not propagate out of the pattern function `foo`; it is local inside it.

9.82.2 Function `match-fboundp`

Syntax:

```
(match-fboundp symbol)
```

Description:

The `match-fboundp` function returns `t` or `nil` if, respectively, *`symbol`* is the name of an existing pattern function.

9.82.3 Macro `txr-if`

Syntax:

```
(txr-if name (argument*) input
        then-expr [else-expr])
```

Description:

The `txr-if` macro invokes the **TXR** pattern-matching function *`name`* on some input given by the *`input`* parameter, whose semantics are the same as the *`input`* argument of the `match-fun` function.

If *`name`* succeeds, then *`then-expr`* is evaluated, and if it fails, *`else-expr`* is evaluated instead.

In the successful case, *`then-expr`* is evaluated in a scope in which the bindings emerging from the *`name`* function are turned into **TXR Lisp** variables. The result of `txr-if` is that of *`then-expr`*.

In the failed case, *`else-expr`* is evaluated in a scope which does not have any new bindings. The result of `txr-if` is that of *`else-expr`*. If *`else-expr`* is missing, the result is `nil`.

The *`argument`* forms supply arguments to the pattern function *`name`*. There must be as many of these arguments as the function has parameters.

Any argument which is a symbol is treated, for the purposes of calling the pattern function, as an unbound pattern variable. The function may or may not produce a binding for that variable. Also, every argument which is a symbol also denotes a local variable that is established around *`then-expr`* if the function succeeds. For any such pattern variable for which the function produces a binding, the corresponding local variable will be initialized with the value of that pattern variable. For any such pattern variable which is left unbound by the function, the corresponding local variable will be set to `nil`.

Any *`argument`* can be a form other than a symbol. In this situation, the argument is evaluated, and will be passed to the pattern function as the value of the binding for the corresponding argument.

Example:

```
@(define date (year month day))
@{year /\d\d\d\d/}-@{month /\d\d/}-@{day /\d\d/}
@(end)
@(do
   (each ((date '("09-10-20" "2009-10-20"
                  "July-15-2014" "foo")))
     (txr-if date (y m d) date
       (put-line `match: year @y, month @m, day @d`)
       (put-line `no match for @date`))))

Output:
```

```
no match for 09-10-20
match: year 2009, month 10, day 20
no match for July-15-2014
no match for foo
```

9.82.4 Macro `txr-when`

Syntax:

```
(txr-when name (argument*) input form*)
```

Description:

The `txr-when` macro is based on `txr-if`. It is equivalent to

```
(txr-if name (argument*) input (progn form*))
```

If the pattern function *`name`* produces a match, then each *`form`* is evaluated in the scope of the variables established by the *`argument`* expressions. The result of the `txr-when` form is that of the last *`form`*.

If the pattern function fails then the forms are not evaluated, and the result value is `nil`.

9.82.5 Macro `txr-case`

Syntax:

```
(txr-case input-form
   {(name (argument*) form*)}*
   [(t form*)])
```

Description:

The `txr-case` macro evaluates *`input-form`* and then uses the value as an input to zero or more test clauses. Each test clause invokes the pattern function named by that clause's *`name`* argument.

If the function succeeds, then each *`form`* is evaluated, and the value of the last *`form`* is taken to be the result value of `txr-case`, which terminates. If there are no forms, then `txr-case` terminates with a `nil` result.

The forms are evaluated in an environment in which variables are bound based on the *`argument`* forms, with values depending on the result of the invocation of the *`name`* pattern function, in the same manner as documented in detail for the `txr-if` macro.

If the function fails, then the forms are not evaluated, and control passes to the next clause.

A clause which begins with the symbol `t` executes unconditionally and causes `txr-case` to terminate. If it has no forms, then `txr-case` yields `nil`, otherwise the forms are evaluated in order and the value of the last one specifies the result of `txr-case`.

The value of the input *`input-form`* is expected to be one of the same kinds of objects as given by the requirements for the *`input`* argument of the `match-fun` functions.

If *`input-form`* evaluates to a stream object according to the `streamp` function, then the stream is converted to a lazy list of lines, as if by invoking the `get-lines` function on that stream; that list then serves as input to the clauses.

9.82.6 Function `txr-parse`

Syntax:

```
(txr-parse [source [error-stream
            [error-retval [name]]]])
```

Description:

The `txr-parse` function converts textual **TXR** query syntax into a Lisp data structure representation.

The *source* argument may be either a character string, or a stream. If it is omitted, then `*stdin*` is used as the stream.

The *source* must provide the text representation of one complete **TXR** query.

The optional *error-stream* argument can be used to specify a stream to which diagnostics of parse errors are sent. If absent, the diagnostics are suppressed.

The optional *name* argument can be used to specify the file name which is used for reporting errors. If this argument is missing, the name is taken from the name property of the *source* argument if it is a stream, or else the word `string` is used as the name if *source* is a string.

If there are no parse errors, the function returns the parsed data structure. If there are parse errors, and the *error-retval* parameter is present, its value is returned. If the *error-retval* parameter is not present, then an exception of type `syntax-error` is thrown.

9.83 Debugging Functions

9.83.1 Functions `source-loc` and `source-loc-str`

Syntax:

```
(source-loc form)
(source-loc-str form [alternative])
```

Description:

These functions map an expression in a **TXR** program to the file name and line number of the source code where that form came from.

The `source-loc` function returns the raw information as a cons cell whose `car`/`cdr` consist of the line number, and file name.

The `source-loc-str` function formats the information as a string.

Forms which were parsed from a file have source location info tracking to their origin in that file. Forms which are the result of macro-expansion are traced to the form whose evaluation produced them. That is to say, they inherit that form's source location info.

More precisely, when a form is produced by macro-expansion, it usually consists of material which was passed to the macro as arguments, plus some original material allocated by the macro, and possibly literal structure material which is part of the macro code. After the expansion is produced, any of its constituent material which already has source location info keeps that info. Those nodes which are newly allocated by the macro-expansion process inherit their source location info from the form which yields the expansion.

If *form* is not a piece of the program source code that was constructed by the **TXR** parser or by a

macro, and thus it was neither attributed with source location info, nor has it inherited such info, then `source-loc` returns `nil`.

In the same situation, and if its *`alternative`* argument is missing, the `source-loc-str` returns a string whose text conveys that the source location is not available. If the *`alternative`* argument is present, it is returned.

9.83.2 Functions `rlcp` and `rlcp-tree`

Syntax:

```
(rlcp dest-form source-form)
(rlcp dest-tree source-form)
```

Description:

The `rlcp` function copies the source code location info ("rl" means "read location") from the *`source-form`* object to the *`dest-form`* object. These objects are pieces of list-based syntax. If *`dest-form`* already has source code location info, then no copying takes place.

The `rlcp-tree` function copies the source code location info from `rlcp` into every cons cell in the *`dest-tree`* tree structure which doesn't already have location info. It may be regarded as a recursive application of `rlcp` via `car`/`cdr` recursion on the tree structure. However, the traversal performed by `rlcp-tree` gracefully handles circular structures.

Note: these functions are intended to be used in certain kinds of macros. If a macro transforms *`source-form`* to *`dest-form`*, this function can be used to propagate the source code location info also, so that when the **TXR Lisp** evaluator encounters errors in transformed code, it can give diagnostics which refer to the original untransformed source code.

The macro expander already performs this transfer. If a macro call form has location info, the expander propagates that info to that form's expansion. In some situations, it is useful for a macro or other code transformer to perform this action explicitly.

9.83.3 Special variable `*rec-source-loc*`

Description:

The Boolean special variable `*rec-source-loc*` controls whether the `read` and `iread` functions record source location info. The variable is `nil` by default, so that these functions do not record source location info. If it is true, then these functions record source location info.

Regardless of the value of this variable, source location info is recorded for Lisp forms which are read from files or streams under the `load` function or specified on the **TXR** command line. Source location info is also always recorded when reading the **TXR** pattern language syntax.

Note: recording and propagating location info incurs a memory and performance penalty. The individual cons cells and certain other literal objects in the structure which emerges from the parser are associated with source location info via a global weak hash table.

9.83.4 Function `macro-ancestor`

Syntax:

```
(macro-ancestor form)
```

Description:

The `macro-ancestor` function returns information about the macro-expansion ancestor of *form*. The ancestor is the original form whose expansion produced *form*.

If *form* is not the result of macro-expansion, or the ancestor information is unavailable, the function returns `nil`.

9.84 Profiling

9.84.1 Operator `prof`

Syntax:

```
(prof form*)
```

Description:

The `prof` operator evaluates the enclosed forms from left to right similarly to `progn`, while determining the memory allocation requests and time consumed by the evaluation of the forms.

If there are no forms, the prof operator measures the smallest measurable operation of evaluating nothing and producing `nil`.

If the evaluation terminates normally (not abruptly by a nonlocal control transfer), then `prof` yields a list consisting of:

```
(value malloc-bytes gc-bytes milliseconds)
```

where *value* is the value returned by the rightmost *form*, or `nil` if there are no forms, *malloc-bytes* is the total number of bytes of all memory allocation requests (or at least those known to the **TXR** runtime, such as those of all internal objects), *gc-bytes* is the total number of bytes drawn from the garbage-collected heaps, and *milliseconds* is the total processor time consumed over the execution of those forms.

Notes:

The bytes allocated by the garbage collector from the C function `malloc` to create heap areas are not counted as *malloc-bytes*. *malloc-bytes* includes storage such as the space used for dynamic strings, vectors and bignums (in addition to their gc-heap-allocated nodes), and the various structures used by the `cobj` type objects such as streams and hashes. Objects in external libraries that use uninstrumented allocators are not counted: for instance the C `FILE *` streams.

9.84.2 Macro `pprof`

Syntax:

```
(pprof form*)
```

Description:

The `pprof` (pretty-printing `prof`) macro is similar to `progn`. It evaluates *form*s, and returns the rightmost one, or `nil` if there are no forms.

Over the evaluation of *form*s, it counts memory allocations, and measures CPU time. If *form*s terminate normally, then just prior to returning, `pprof` prints these statistics in a concise report on the `*stdout*`.

The `pprof` macro relies on the `prof` operator.

9.85 Garbage Collection

9.85.1 Function `sys:gc`

Syntax:

```
(sys:gc [full])
```

Description:

The `gc` function triggers garbage collection. Garbage collection means that unreachable objects are identified and reclaimed, so that their storage can be reused.

The function returns `nil` if garbage collection is disabled (and consequently nothing is done), otherwise `t`.

The Boolean *full* argument, defaulting to `nil`, indicates whether a full garbage collection should be requested.

Even if this argument is `nil`, a full garbage collection may occur due to having been scheduled.

9.85.2 Function `sys:gc-set-delta`

Syntax:

```
(sys:gc-set-delta bytes)
```

Description:

The `gc-set-delta` function sets the GC delta parameter.

Note: This function may disappear in a future release of **TXR** or suffer a backward-incompatible change in its syntax or behavior.

When the amount of new dynamic memory allocated since the last garbage collection equals or exceeds the GC delta, a garbage collection pass is triggered. From that point, a new delta begins to be accumulated.

Dynamic memory is used for allocating heaps of small garbage-collected objects such as cons cells, as well as the satellite data attached to some objects: like the storage arrays of vectors, strings or bignum integers. Most garbage collector behaviors are based on counting objects in the heaps.

Sometimes a program works with a small number of objects which are very large, frequently allocating new, large objects and turning old ones into garbage. For instance a single large integer could be many megabytes long. In such a situation, a small number of heap objects therefore control a large amount of memory. This requires garbage collection to be triggered much more often than when working with small objects, such as conses, to prevent runaway allocation of memory. It is for this reason that the garbage collector uses the GC delta.

There is a default GC delta of 64 megabytes. This may be overridden in special builds of **TXR** for small systems.

9.85.3 Function `finalize`

Syntax:

```
(finalize object function [reverse-order-p])
```

Description:

The `finalize` function registers *`function`* to be invoked in the situation when *`object`* is identified by the garbage collector as unreachable. A function registered in this way is called a finalizer.

If and when this situation occurs, the finalizer *`function`* will be called with *`object`* as its only argument.

Multiple finalizer functions can be registered for the same object, up to an internal limit which is not required to be greater than 255. If the limit is exceeded, `finalize` throws an error exception.

All registered finalizers are called when the object becomes unreachable. Finalizers registered against an object may also be invoked and removed using the `call-finalizers` function.

If the *`reverse-order-p`* argument isn't specified, or is `nil`, then finalizer is registered at the end of the list.

If *`reverse-order-p`* is true, then the finalizer is registered at the front of the list.

Finalizers which are activated in the same finalization processing phase are called in the order in which they appear in the registration list.

After a finalization call takes place, its registration is removed. However, neither *`object`* nor *`function`* are reclaimed immediately; they are treated as if they were reachable objects until at least the next garbage collection pass. It is therefore safe for *`function`* to store somewhere a persistent reference to *`object`* or to itself, thereby reinstating these objects as reachable.

A finalizer is itself permitted to call `finalize` to register the original `object` or any other object for finalization. Finalization processing can be understood as taking place in one or more rounds. At the start of each round, finalizers are identified that are to be called, arranged in order, and removed from the registration list. If this identification stage produces no finalizers, then finalization ends. Otherwise, those finalizers are processed, and then another round is initiated, to look for eligible finalizers that may have been registered during the previous round.

Note: it is possible for the application to create an infinite finalization loop, if one or more objects have finalizers that register new finalizers, which register new finalizers and so on.

Note: if a finalizer is invoked by the garbage collector rather than explicit finalization via `call-finalizers`, and that finalizer calls `finalize` to make a registration, that registration will not be eligible for processing in the same phase, because the criteria for finalization is unreachability.

9.85.4 Function `call-finalizers`

Syntax:

```
(call-finalizers object)
```

Description:

The `call-finalizers` function invokes and removes the finalizers, if any, registered against *`object`*. If any finalizers are called, it returns `t`, otherwise `nil`.

Finalization performed by `call-finalizers` works in the manner described under the specification of the `finalize` function.

It is permissible for a finalizer function itself to call `call-finalizers`. Such a call can happen in two possible contexts: finalization initiated by by garbage collection, or under the scope of a `call-finalizers` invocation from application code. Doing so is safe, since the finalization logic may be reentered recursively. When finalizers are being called during a round of processing, those finalizers have already been removed from the registration list, and will not be redundantly invoked by a recursive invocation of finalization.

Under the scope of garbage-collection-driven reclamation, the order of finalizer calls may not be what the application logic expects. For instance even though a finalizer registered for some object `A` itself invokes `(call-finalizers B)`, it may be the case during GC reclamation that both `A` and `B` are identified as unreachable objects at the same time, and some or all finalizers registered against `B` have already been called before the given `A` finalizer performs the explicit `call-finalizers` invocation against `B`. Thus the call either has no effect at all, or only calls some remaining `B` finalizers that have not yet been processed, rather than all of them, as the application expects.

The application must avoid creating a dependency on the order of finalization calls, to prevent the situation that the finalization actions are only correct under an explicit `call-finalizers` but incorrect under spontaneous reclamation driven by garbage collection.

9.86 Stack-Overflow Protection

TXR features a rudimentary mechanism for guarding against stack overflows, which cause the **TXR** process to crash. This capability is separate from and exists in addition to the possibility of catching a `sig-segv` (segmentation violation) signal upon stack overflow using `set-sig-handler`.

The stack-overflow guard mechanism is based on **TXR**, at certain key places in the execution, checking the current position of the stack relative to a predetermined limit. If the position exceeds the limit, then an exception of type `stack-overflow`, derived from `error`, is thrown.

The stack-overflow guard mechanism is configured on startup. On platforms where it is possible to inquire the system's actual stack limit, and where the stack limit is at least 512 kilobytes, **TXR** sets the limit to within a certain percentage of the actual value. If it is not possible to determine the system's stack limit, or if the system indicates that the stack size is unlimited, then a default limit is imposed. If the system's limit is configured below a certain small value, then that small value is used as the stack limit.

The `get-stack-limit` and `set-stack-limit` functions are provided to manipulate the stack limit.

The mechanism cannot contain absolutely all sources of stack-overflow threat under all conditions. External functions are not protected, and not all internal functions are monitored. If **TXR** is close to the limit, but a function is called whose stack growth is not monitored, such as an external function or unmonitored internal function, it is possible that the stack may overflow anyway.

9.86.1 Functions `get-stack-limit` and `set-stack-limit`

Syntax:

```
(get-stack-limit)
(set-stack-limit value)
```

Description:

The `get-stack-limit` returns the current value of the stack limit. If the guard mechanism is not enabled, it returns `nil`, otherwise it returns a positive integer, which is measured in bytes.

The `set-stack-limit` configures the stack limit according to *value*, possibly enabling or

disabling the guard mechanism, and returns the previous stack limit in exactly the same manner as `get-stack-limit`.

The *`value`* must be a non-negative integer or else the symbol `nil`.

The values zero or `nil` disable the guard mechanism. Positive integer values set the limit. The value may be truncated to a multiple of some denomination or otherwise adjusted, so that a subsequent call to `get-stack-limit` need not retrieve that exact value.

If *`value`* is too close to the system's stack limit or beyond, the effectiveness of the stack-overflow detection mechanism is compromised. Likewise, if *`value`* is too low, the operation of **TXR** shall become unreliable. Values smaller than 32767 bytes are strongly discouraged.

9.87 Modularization

9.87.1 Variable `self-path`

Description:

This variable holds the invocation pathname of a **TXR** program that was specified on the command line.

The value of `self-path` when **TXR Lisp** expressions are being evaluated in command-line arguments is the string `"cmdline-expr"`. The value of `self-path` when a **TXR** query is supplied on the command line via the `-c` command-line option is the string `"cmdline"`.

When a file is being compiled using the `--compile` option, the value of `self-path` is the source file path.

When the interactive listener is entered, `self-path` is set to the value `"listener"`, even if prior to that, a file was compiled or executed, for which `self-path` had been set to the name of that file.

Note that for programs read from a file, `self-path` holds the resolved name, and not the invocation name. For instance if `foo.tl` is invoked using the name `foo`, whereby **TXR** infers the suffix, then `self-path` holds the suffixed name.

Note that the functions `load`, `compile-file` and `compile-update-file` have no effect on the value of `self-path`. The variable is set strictly by command line processing.

9.87.2 Variable `stdlib`

Description:

The `stdlib` variable expands to the directory where the **TXR** standard library is installed. It includes the trailing slash.

Note: there is no need to use the value of this variable to load library modules. Library modules are keyed to specific symbols, and lazily loaded. When a **TXR Lisp** library function, macro or variable is referenced for the first time, the library module which defines it is loaded. This includes references which occur during the code expansion phase, at "macro time", so it works for macros. In the middle of processing a syntax tree, the expander may encounter a symbol that is registered for autoloading, and trigger the load. When the load completes, the symbol might now be defined as a macro, which the expander can immediately use to expand the given form that is being traversed.

9.87.3 Function `load`

Syntax:

```
(load target)
```

Description:

The `load` function causes a file containing **TXR Lisp** or **TXR** code to be read and processed. The *`target`* argument is a string. The function can load **TXR Lisp** source files as well as compiled files.

Firstly, the value in *`target`* is converted to a *tentative pathname* as follows.

If *`target`* specifies a pure relative pathname, as defined by the `pure-rel-path-p` function, then a special behavior applies. If an existing load operation is in progress, then the special variable `*load-path*` has a binding. In this case, `load` will assume that the relative pathname is a reference relative to the directory portion of that pathname. If `*load-path*` has the value `nil`, then a pure relative *`target`* pathname is used as-is, and thus resolved relative to the current working directory.

Once the tentative pathname is determined, `load` determines whether the name is suffixed. The name is suffixed if it ends in any of these four suffixes: `.tlo`, `.tlo.gz`, `.tl`, `.txr` or `.txr_profile`.

Depending on whether the tentative pathname exists, and whether or not it is suffixed, `load` tries to make one or more attempts to open several variations of that name. These variations are called *actual paths*. If any attempt fails due to an error other than non-existence, such as a permission error, then no further attempts are made; the error exception propagates to `load`'s caller.

Regardless of whether the tentative pathname is suffixed, `load` tries to open a file by that actual pathname first. If that attempt fails for a suffixed pathname, or fails due to a reason other than non-existence, no other names are tried.

If an unsuffixed tentative pathname refers to a nonexistent file, `.tlo` is appended to the name, and an attempt is made to open a file with the resulting path. If that file is not found, then the suffixes `.tlo.gz` and `.tl` are similarly tried.

If the above *initial attempts* to find the file fail, and the failure is due to the file not being found rather than some other problem such as a permission error, and *`target`* isn't an absolute path according to `abs-path-p`, then additional attempts are made by searching for the file in the list of directories given in the `*load-search-dirs*` variable. For each directory taken from this variable, the directory is combined with the relative *`target`* as if using the `path-cat` function, and the resulting path is tried, with all the same suffix probing that is performed by the initial attempts. If any such a path is pure relative, it is interpreted relative to the current working directory, and not relative `*load-path*`: only the initial attempts have that special behavior.

An exception is thrown if a file is not found, or if any attempt to open a file results in an error other than non-existence.

If an unsuffixed file is successfully opened, its contents are treated as interpreted Lisp. Files ending in `.txr_profile` are also treated as interpreted Lisp. Files ending in `.tlo` are treated as compiled Lisp, and those ending in `.txr` are treated as the **TXR** Pattern Language.

The `.tlo.gz` suffix denotes a file which is expected to be compressed in the `gzip` format, and to contain compiled Lisp.

If the file is treated as **TXR Lisp**, then Lisp forms are read from it in succession. Each form is evaluated as if by the `eval` function, before the next form is read. If a syntax error is encountered, an exception of type `eval-error` is thrown.

If a file is treated as a compiled **TXR Lisp** object file, then the compiled images of top-level forms are read from it, converted into compiled objects, and executed.

If the file treated as **TXR** Pattern Language code, then its contents are parsed in their entirety. If the parse is successful, the query is executed. Previous **TXR** pattern variable and function bindings are in effect. If the query binds new variables and functions, these emerge from the `load` and take effect. If the parse is unsuccessful, an exception of type `query-error` is thrown.

Parser error messages are directed to the `*stderr*` stream.

Over the evaluation of either a **TXR Lisp**, compiled file, or **TXR** file, `load` establishes a new dynamic binding for several special variables. The variable `*load-path*` is given a new binding containing the actual pathname. The `*package*` variable is also given a new dynamic binding, whose value is the same as the existing binding. Thus if the processing of the loaded file has the effect of altering the value of `*package*`, that effect will be undone when the binding is removed after the load completes.

When the `load` function terminates normally after processing a file, it returns `nil`. If the file contains a **TXR** pattern query which is processed to completion, the matching success or failure of that query has no bearing on the return value of `load`. Note that this behavior is different from the `@(load)` directive which itself fails if the loaded query fails, causing subsequent directives not to be processed.

A **TXR** pattern language file loaded with the Lisp `load` function does not have the usual implicit access to the command-line arguments, unlike a top-level **TXR** query. If the directives in the file try to match input, they work against the `*stdin*` stream. The `@(next)` directive behaves as it does when no more arguments are available.

If the source or compiled file begins with the characters `#!`, usually indicating a hash-bang script, `load` reads reads the first line of the file and discards it. Processing of the file then begins with the first byte following that line.

Two or more `.tlo` files produced by the same version of **TXR** may be catenated together (for instance, using the `cat-files` function) to produce a single `.tlo` file. Such a combined file can be loaded with the `load` function. The same is true of `.tlo.gz` files, because the `gzip` format supports catenation. Mixing is not possible: `.tlo` and `.tlo.gz` files cannot be catenated together.

Note: this is a single `load` operation: all of the binding and unbinding of variables like `*load-path*` and `*package*` is performed once over the entire contents of the combined file, and any `*load-hooks*` are performed one time after the load operation. Therefore it is possible that the load-time behavior differs from that of loading the original files individually. The `*load-path*` is bound to the name of the combined file.

9.87.4 Special variable `*load-path*`

Description:

The `*load-path*` special variable has a top-level value which is `nil`.

When a file is being loaded, it is dynamically bound to the pathname of that file. This value is

visible to the forms are evaluated in that file during the loading process.

The `*load-path*` variable is bound when a file is loaded from the command line.

If the `-i` command-line option is used to enter the interactive listener, and a file to be loaded is also specified, then the `*load-path*` variable remains bound to the name of that file inside the listener.

The `load` function establishes a binding for `*load-path*` prior to processing and evaluating all the top-level forms in the target file. When the forms are evaluated, the binding is discarded and `load` returns.

The `compile-file` function also establishes a binding for `*load-path*`.

The `@(load)` directive, also similarly establishes a binding around the parsing and processing of a loaded **TXR** source file.

Also, during the processing of the profile file (see Interactive Profile File), the variable is bound to the name of that file.

9.87.5 Special variable `*load-search-dirs*`

Description:

The `*load-search-dirs*` variable holds a list of directories which are searched for a file to be loaded by the `load` function, the `@(load)` and `@(include)` directives, as well as by **TXR**'s command line processing.

Each of these situations first searches for a file in its characteristic way. If that fails due to the file not being found, and the name is a relative path, then the directories in `*load-search-dirs*` are probed, in order.

The variable is initialized to a list which contains exactly one directory: a `lib/` directory dynamically calculated relative to **TXR** the executable location. Then intent is that third-party library modules may be installed there, and easily found by `load`. For more information, see the section Deployment Directory Structure.

The `*load-search-dirs*` isn't influenced by any environment variables, which is deliberate. If a system has multiple installations of different versions of **TXR** in different locations, an environment variable intended for one installation could be mistakenly used by the others, resulting in chaos.

9.87.6 Special variable `*load-hooks*`

Description:

The `*load-hooks*` variable is at the centre of a mechanism which associates the deferred execution of actions, associated with a loaded module or program termination.

The application may push values onto this list which are expected to be functions, or objects that may be called as functions. These objects must be capable of being called with no arguments.

In the situations specified below, the list of functions is processed as follows. First `*load-hooks*` is examined, the list which it holds is remembered. Then the variable is reset to `nil`, following which the remembered list is traversed in order. Each of the functions in the list is invoked, with no arguments.

The `*load-hooks*` list is processed, as described above, whenever the `load` function terminates, whether normally or by throwing an exception. In this situation, the `*load-hooks*` variable which is accessed is that binding which was established by that invocation of `load`. The execution of the functions from the `*load-hooks*` list takes place in the dynamic environment of the `load`: all of the dynamic variable bindings established by that `load` are still visible, including that of `*load-hooks*`.

The `*load-hooks*` list is also processed after processing a **TXR** or **TXR Lisp** file that is specified on the command line. If the interactive listener is also being entered, this processing of `*load-hooks*` occurs prior to entering the listener. This situation occurs in the context of the top-level dynamic environment, and so the global value of `*load-hooks*` is referenced.

Lastly, `*load-hooks*` is also processed if the **TXR** process terminates normally, regardless of its exit status. This processing executes in whatever dynamic environment is current at the point of exit, using its value of the `*load-hooks*` variable is used. It is unspecified whether, at exit time, the `*load-hooks*` functions are executed first, or whether the functions registered by `at-exit-call` are executed first. However, their executions do not interleave.

Note that `*load-hooks*` is not processed after the listener reads the `.txr_profile` file. Hooks installed by the profile file will activate when the process exits.

9.87.7 Macros `push-after-load` and `pop-after-load`

Syntax:

```
(push-after-load form*)
(pop-after-load)
```

Description:

The `push-after-load` and `pop-after-load` macros work with the `*load-hooks*` list.

The `push-after-load` macro's arguments are zero or more *forms*. These forms are converted into the body of an anonymous function, which is pushed onto the `*load-hooks*` list. The return value is the new value of `*load-hooks*`.

The `pop-after-macro` removes the first item from `*load-hooks*`. The return value is the new value of `*load-hooks*`.

The following equivalences hold:

```
(push-after-load ...)  <-->  (push (lambda () ...) *load-hooks*)
(pop-after-load)  <-->  (set *load-hooks* (cdr *load-hooks*))
```

9.87.8 Macro `load-for`

Syntax:

```
(load-for {(kind sym target)}*)
```

Description:

The `load-for` macro takes multiple arguments, each of which is a three-element clause. Each clause specifies that a given *target* file is to be conditionally loaded based on whether a symbol *sym* has a certain kind of binding.

Each argument clause has the syntax `(kind sym target)` where *kind* is one of the five symbols `var`, `fun`, `macro`, `struct` or `pkg`. The *sym* element is a symbol suitable for use as a

variable, function or structure name, and *target* is an expression which is evaluated to produce a value that is suitable as an argument to the `load` function.

First, all `target` expressions in all clauses are unconditionally evaluated in left-to-right order. Then the clauses are processed in that order. If the *kind* symbol of a clause is `var`, then `load-for` tests whether *sym* has a binding in the variable namespace using the `boundp` function. If a binding does not exist, then the value of the *target* expression is passed to the `load` function. Otherwise, `load` is not called. Similarly, if *kind* is the symbol `fun`, then *sym* is instead tested using `fboundp`, if *kind* is `macro`, then *sym* is tested using `mboundp`, if *kind* is `struct`, then *sym* is tested using `find-struct-type`, and if *kind* is `pkg`, then *sym* is tested using `find-package`.

When `load-for` invokes the `load` function, it confirms whether loading file has had the expected effect of providing a definition of *sym* of the right *kind*. If this isn't the case, an error is thrown.

The `load-for` function returns `nil`.

9.87.9 Variable `txr-exe-path`

Description:

This variable holds the absolute pathname of the executable file of the running **TXR** instance.

9.88 Function Tracing

9.88.1 Special variable `*trace-output*`

Description:

The `*trace-output*` special variable holds a stream to which all trace output is sent. Trace output consists of diagnostics enabled by the `trace` macro.

9.88.2 Macros `trace` and `untrace`

Syntax:

```
(trace function-name*)
(untrace function-name*)
```

Description:

The `trace` and `untrace` macros control function tracing.

When `trace` is called with one or more arguments, it considers each argument to be the name of a global function. For each function, it turns on tracing, if it is not already turned on. If an argument denotes a nonexistent function, or is invalid function name syntax, `trace` terminates by throwing an exception, without processing the subsequent arguments, or undoing the effects already applied due to processing the previous arguments.

When `trace` is called with no arguments, it lists the names of functions for which tracing is currently enabled. In other cases it returns `nil`.

When `untrace` is called with one or more arguments, it considers each argument to be the name of a global function. For each function, it turns off tracing, if tracing is enabled.

When `untrace` is called with no arguments, it disables tracing for all functions.

The `untrace` macro always returns `nil` and silently tolerates arguments which are not names of functions currently being traced.

Tracing a function consists of printing a message prior to entry into the function indicating its name and arguments, and another message upon leaving the function indicating its return value, which is syntactically correlated with the entry message, using a combination of matching and indentation. These messages are posted to the `*trace-output*` stream.

When traced functions call each other or recurse, these trace messages nest. The nesting is detected and translated into indentation levels.

Tracing works by replacing a function definition with a trace hook function, and retaining the previous definition. The trace hook calls the previous definition and produces the diagnostics around it. When `untrace` is used to disable tracing, the previous definition is restored.

Methods can be traced; their names are given using `(meth` *`struct slot`*`)` syntax: see the `func-get-name` function.

Macros can be traced; their names are given using `(macro` *`name`*`)` syntax. Note that `trace` will not show the destructured internal macro arguments, but only the two arguments passed to the expander function: the whole form, and the environment.

The `trace` and `untrace` functions return `nil`.

9.89 Dynamic Library Access

9.89.1 Function `dlopen`

Syntax:

```
(dlopen [{lib-name | nil} [flags])
```

Description:

The `dlopen` function provides access to the POSIX C library function of the same name.

The argument to the optional *`lib-name`* parameter may be a character string, or `nil`.

If it is `nil`, then the POSIX function is called with a null pointer for its name argument, returning the handle for the main program, if possible.

The *`flags`* argument should be expressed as some bitwise combination of the values of the variables `rtld-lazy`, `rtld-now`, or other `rtld-` variables which give names to the `dlopen`-related flags. If the *`flags`* argument is omitted, the default value used is `rtld-lazy`.

If the function succeeds, it returns an object of type `cptr` which represents the open library handle ("dlhandle").

Otherwise it throws an exception, whose message incorporates, if possible, error text retrieved from the `dlerror` POSIX function.

The `cptr` handle returned by `dlopen` will automatically be subject to `dlclose` when reclaimed by the garbage collector.

9.89.2 Function `dlclose`

Syntax:

```
(dlclose dlhandle)
```

Description:

The `dlclose` closes the library indicated by *`dlhandle`*, which must be a `cptr` object previously returned by `dlopen`.

The handle is closed by passing the stored pointer to the POSIX `dlclose` function. The internal pointer contained in the `cptr` object is then reset to null.

It is permissible to invoke `dlclose` more than once on a `cptr` object which was created by `dlopen`. The first invocation resets the `cptr` object's pointer to null; the subsequent invocations do nothing.

The `dlclose` function returns `t` if the POSIX function reports a successful result (zero), otherwise it returns `nil`. It also returns `nil` if invoked on a previously closed, and hence nulled-out `cptr` handle.

9.89.3 Functions `dlsym` and `dlvsym`

Syntax:

```
(dlsym dlhandle sym-name)
(dlvsym dlhandle sym-name ver-name)
```

Description:

The `dlsym` function provides access to the same-named POSIX function. The `dlvsym` function provides access to the same-named GNU C Library function, if available.

The *`dlhandle`* argument must be a `cptr` handle previously returned by `dlopen` and not subsequently closed by `dlclose` or altered in any way.

The *`sym-name`* and *`ver-name`* arguments are character strings.

If these functions succeed, they return a `cptr` value which holds the address of the symbol which was found in the library.

If they fail, they return a `cptr` object containing a null pointer.

9.89.4 Functions `dlsym-checked` and `dlvsym-checked`

Syntax:

```
(dlsym-checked dlhandle sym-name)
(dlvsym-checked dlhandle sym-name ver-name)
```

Description:

The `dlsym-checked` and `dlvsym-checked` functions are alternatives to `dlsym` and `dlvsym`, respectively. Instead of returning a null `cptr` on failure, these functions throw an exception.

9.89.5 Variables `rtld-lazy`, `rtld-now`, `rtld-global`, `rtld-local`, `rtld-nodelete`, `rtld-noload` and `rtld-deepbind`

Description:

These variables provide the same values as constants in the POSIX C library header `<dlfcn.h>` named `RTLD_LAZY`, `RTLD_NOW`, `RTLD_LOCAL`, etc.

9.90 Data Interchange Support

9.90.1 Macro `json`

Syntax:

```
(json [quote | sys:qquote] object)
```

Description:

The `json` macro exists in support of the JSON literal and quasiliteral `#J`*`json-syntax`* and `#J^`*`json-syntax`* notations, which use the macro as their target abstract syntax.

The macro transforms itself by deleting the `json` symbol, producing either the `(quote` *`object`*`)` quote syntax, or else the `(sys:qquote` *`object`*`)` quasiquote syntax, depending on which quoting symbol is present.

If the application produces and expands a `json` macro form which does not conform to this syntax, or does not specify one of the above two quoting symbols, the behavior is unspecified.

9.90.2 Functions `put-json` and `put-jsonl`

Syntax:

```
(put-json obj [stream [flat-p]])
(put-jsonl obj [stream [flat-p]])
```

Description:

The `put-json` function converts *`obj`* into JSON notation, and writes that notation into *`stream`* as a sequence of characters.

If *`stream`* is an external stream such as a file stream, then the JSON is rendered by conversion of the characters into UTF-8, in the usual manner characteristic of those streams.

The behavior is unspecified if *`obj`* or any component of *`obj`* is an object incompatible with the JSON representation conventions. An exception may be thrown.

An object conforms to the JSON representation conventions if it is:

1. one of the symbols `nil`, `t` or `null`, which map to the JSON keywords `false`, `true` and `null`, respectively.
2. a floating-point number.
3. a character string.
4. a vector of JSON-conforming objects.
5. a hash table whose keys and values are JSON-conforming objects.

Note that unless the keys in a hash table are all strings, nonstandard JSON is produced, since RFC 8259 requires JSON object keys to be strings.

If the `flat-p` argument is present and has a true value, then the JSON is generated without any line breaks or indentation. Otherwise, the JSON output is subject to such formatting.

The difference between put-json and put-jsonl is that the latter emits a newline character after the JSON output.

When a string object is output as JSON string syntax, the following rules

1. The characters \ (backslash, reverse solidus) and " (double quote) are preceded by a backslash escape.
2. The characters U+0008 (BS), U+0009 (TAB), U+000A (LF), U+000C (FF) and U+000D (CR) are rendered as, respectively, \b, \t, \n, \f and \r.
3. If the character sequence </script occurs in a string, then in the JSON representation the slash is escaped, such that the sequence is rendered as <\/script. Instances of / (forward slash, solidus) in other situations are unescaped. Rationale: this is a feature of JSON which allows for safer embedding of the resulting JSON into HTML script tags.
4 If the character sequence <!-- occurs in a string, then in the JSON representation, the sequence is rendered as <\u0021--. Instances of ! (exclamation mark) in other situations are not encoded. Rationale: safe embedding in HTML script tags.
5 If the character sequence --> occurs in a string, then in the JSON representation, the sequence is rendered as -\u002D>. Instances of - (hyphen) in other situations are not encoded. Rationale: safe embedding in HTML script tags.
6. The code point U+DC00 (**TXR**'s pseudo-null character) is translated into the \u0000 escape syntax.
7. The code points U+DC01 through U+DCFF are send to the stream as-is. If the stream performs UTF-8 encoding, these characters turn into individual bytes in the range 0 to 255.
8. Control characters in the U+0001 to U+001F other than the ones subject to rule 1 above are rendered as \u escape sequences. Likewise, code points in the range U+007F to U+00BF, the range U+D800 to U+DBFF, U+DD00 to U+DFFF, and the code points U+FFFE and U+FFFF are also encoded as \u escape sequences.
9. A character outside of the BMP (Basic Multilingual Plane) in the range U+10000 to U+10FFFF is encoded using as a pair of consecutive \u escape sequences, specifying the code points of a UTF-16 surrogate pair encoding that character. This representation is described in RFC 8259.

The put-json and put-jsonl functions return *t*.

Some of the JSON-related functions carry a *mode-opts* optional parameter. These functions open a file as if using the open-file function, using a *mode-string* appropriate to their direction of data transfer. If an argument is given to *mode-opts*, it specifies the *options* part to be added to the *mode-string*.

9.90.3 Function tojson

Syntax:

```
(tojson obj [flat-p])
```

Description:

The tojson function converts *obj* into JSON notation, returned as a character string.

The function can be understood as constructing a string output stream, calling the put-json function to write the object into that stream, and then retrieving and returning the constructed

string.

The *flat-p* argument is passed to put-json.

9.90.4 Function get-json

Syntax:

```
(get-json [source
           [err-stream
            [err-retval [name [lineno]]]]])
```

Description:

The get-json function closely resembles the read function, and follows the same argument and error reporting conventions.

Rather than reading a Lisp object from the input source, it reads a JSON object, with support for **TXR**'s JSON extensions.

If an object is successfully read, its Lisp representation is returned. JSON numbers produce floating-point number objects. JSON strings produce string objects. The keywords true, false and null map to the Lisp symbols t, nil, and null, respectively. JSON objects map to hash tables, and JSON arrays to vectors.

9.90.5 Function put-jsons

Syntax:

```
(put-jsons seq [stream [flat-p]])
```

Description:

The put-jsons function writes multiple JSON representations into *stream*. The objects are specified by the *seq* argument, which must be an iterable object. The put-jsons function iterates over *seq* and writes each element to the stream as if by using the put-jsonl function. Consequently, a newline character is written after each object.

If the *stream* argument is not specified, the parameter takes on the value of **stdout**.

The *flat-p* argument has the same meaning as in put-json with regard to the individual elements. If it is specified and true, then exactly as many lines of text are written to *stream* as there are elements in *seq*.

The put-jsons function returns *t*.

9.90.6 Function get-jsons

Syntax:

```
(get-jsons [source])
```

Description:

The *get-jsons* function reads zero or more JSON representations from *source* until an end-of-stream or error condition is encountered.

If *source* is a character string, then the input takes place from a stream created from the character string using make-string-byte-input-stream. Otherwise, if *source* is specified, it must be an input stream supporting byte input; input takes place from that stream. If the *source*

argument is omitted, it defaults to `*stdin*`.

The objects are read as if by calls to `get-json` and accumulated into a list.

If the end-of-stream condition is read, then the list of accumulated objects is returned. If an error occurs, then an exception is thrown and the list of accumulated objects is not available.

If an end-of-stream condition occurs before any character is seen other than JSON whitespace, then the empty list `nil` is returned.

9.90.7 Functions `file-get-json` and `file-get-jsons`

Syntax:

```
(file-get-json name [mode-opts])
(file-get-jsons name [mode-opts])
```

Description:

The `file-get-json` and `file-get-jsons` function open a text stream over the file indicated by the string argument *name* for reading. The functions ensure that the stream is closed when they terminate.

The `file-get-json` function invokes `get-json` to read a single JSON object, which is returned if that function returns normally.

The `file-get-jsons` function invokes `get-jsons` to retrieve a list of JSON objects from the stream, which is returned if that function returns normally.

9.90.8 Functions `file-put-json` and `file-put-jsons`

Syntax:

```
(file-put-json name obj [flat-p [mode-opts]])
(file-put-jsons name seq [flat-p [mode-opts]])
```

Description:

The `file-put-json` and `file-put-jsons` functions open a text stream over the file indicated by the string argument *name*, using the function `open-file` with a *mode-string* argument of `"w"`, write the argument object into the stream in their specific manner, and then close the stream.

The `file-put-json` function writes a JSON representation of *obj* using the `put-json` function. The *flat-p* argument is passed to that function, defaulting to `nil`. The value returned is that of `put-json`.

The `file-put-jsons` function writes zero or more JSON representations of objects from *seq*, which must be an iterable object, using the `put-jsons` function. The *flat-p* argument is passed to that function, defaulting to `nil`. The value returned is that of `put-jsons`.

9.90.9 Functions `file-put-json` and `file-put-jsons`

Syntax:

```
(file-append-json name obj [flat-p [mode-opts]])
(file-append-jsons name seq [flat-p [mode-opts]])
```

Description:

The `file-append-json` and `file-append-jsons` are identical in almost all requirements to the functions `file-put-json` and `file-put-jsons`.

The only difference is that when these functions open a text stream using `open-file`, they specify a *`mode-string`* argument of `"a"` rather than `"w"`, in order to append data to the target file rather than overwrite it.

9.90.10 Functions `command-get-json` and `command-get-jsons`

Syntax:

```
(command-get-json cmd [mode-opts])
(command-get-jsons cmd [mode-opts])
```

Description:

The `command-get-json` and `command-get-jsons` functions opens text stream over an input command pipe created for the command string *`cmd`*, as if by the `open-command` function. They ensure that the stream is closed when they terminate.

The `command-get-json` function calls `get-json` on the stream, and returns the value returned by that function.

Similarly, `command-get-jsons` function calls `get-jsons` on the stream, and returns the value returned by that function.

9.90.11 Functions `command-put-json` and `command-put-jsons`

Syntax:

```
(command-put-json cmd obj [flat-p [mode-opts]])
(command-put-jsons cmd seq [flat-p [mode-opts]])
```

Description:

The `command-put-json` and `command-put-jsons` functions open an output text stream over an output command pipe created for the command specified in the string argument *`cmd`*, using the function `open-command` function, write the argument object into the stream, in their specific manner, and then close the stream.

The `command-put-json` function writes a JSON representation of *`obj`* using the `put-json` function. The *`flat-p`* argument is passed to that function, defaulting to `nil`. The value returned is that of `put-json`.

The `command-put-jsons` function writes zero or more JSON representations of objects from *`seq`*, which must be an iterable object, using the `put-jsons` function. The *`flat-p`* argument is passed to that function, defaulting to `nil`. The value returned is that of `put-jsons`.

9.90.12 Variable `*print-json-format*`

Description:

The `*print-json-format*` variable controls the formatting style exhibited by `put-json` and related functions. The initial value of this variable is `nil`.

If the value is the keyword symbol `:standard`, then a widely-used format is used, in which the opening and closing braces and brackets of vectors and dictionaries are printed on separate lines,

as are the elements of those objects.

If the variable has any other value, including the initial value `nil`, then a default format is used in which braces, brackets and elements appear on the same line, subject to automatic breaking and indentation, similar to the way Lisp nested list structure is printed.

9.90.13 Variable `*read-bad-json*`

Description:

This dynamic variable, initialized to a value of `nil`, controls whether the parser is tolerant to certain non-conformances in the syntax of JSON data, which are ordinarily syntax errors.

If the value of this variable is true, then the last element in a JSON array or the last element pair in a JSON object may be followed by spurious trailing comma, which is ignored.

Note: in the future, the variable may be extended to enable other instances of tolerance in the area of JSON parsing.

Example:

```
(get-json "{ 3:4, }") -> ;; syntax error

(let ((*read-bad-json* t))
  (get-json "{ 3:4, }"))
--> #H(() (3.0 4.0))
```

10 FOREIGN FUNCTION INTERFACE

On platforms where it is supported, **TXR** provides a feature called the *foreign function interface*, or FFI. This refers to the ability to interoperate with programming interfaces which are defined by the binary data type representations and calling conventions of the platform's principal C language compiler.

TXR's FFI module provides a succinct Lisp-based type notation for expressing C data types, together with memory-management semantics pertinent to the transfer of data between software components. The notation is used to describe the arguments and return values of functions in external libraries, and of Lisp callback functions that can be called from those libraries. Driven by the compiled representation of the type notation, the FFI module performs transparent conversions between Lisp data types and C data types, and automatically manages memory around foreign calls and incoming callbacks, for many common interfacing conventions.

The FFI module consists of a library of functions which provide all of its semantics. On top of these functions, the FFI module provides a number of macros which comprise an expressive, convenient language for defining foreign interfaces.

The FFI module supports passing and returning both structures and arrays by value. Passing arrays by value isn't a feature of the C language syntax; from the C point of view, these by-value array objects in the **TXR** FFI type system are equivalent to C arrays encapsulated in `struct`s.

A `carray` type is provided for situations when foreign code generates arrays of undeclared, dynamic length, other than strings, and returns these arrays by the usual convention of pointer to the first element. The handling of `carray` requires more responsibility from the application.

10.1 Cautionary Notes

The FFI feature is inherently unsafe. If the FFI type language is used to write incorrect type definitions which do not match the actual binary interface of a foreign function, undefined behavior results. Improper use of FFI can corrupt memory, creating instability and security problems. It can also cause memory leaks and/or use-after-free errors due to inappropriate deallocation of memory.

The implicit memory management behaviors encoded in the FFI type system are convenient, but risky. A minor declarative detail such as writing `str` instead of `str-d` in the middle of some nested type can make the difference between correct code and code which causes a memory leak, or instability by freeing memory which is in use.

FFI developers are encouraged to unit test their FFI definitions carefully and use tools such as Valgrind to detect memory misuses and leaks.

10.2 Key Concepts

10.2.1 The *put* operation

When a function call takes place from the **TXR Lisp** arena into a foreign library function, argument values must be prepared into the foreign representation. This takes place by converting Lisp objects into stack-allocated temporary buffers representing C objects. For aggregate objects containing pointers, additional buffers are allocated dynamically. For instance, suppose a structure contains a string and is passed by value. The structure will be converted to a stack-allocated equivalent C structure, in which the string will appear as a pointer. That pointer may use dynamically allocated (via `malloc`) string data. The operation which prepares argument material before a foreign function call is the *put* operation. In FFI callback dispatch, the operation which propagates the callback return value to the foreign caller is also the put operation.

10.2.2 The *in* operation

After a foreign function call returns from a foreign library back to the **TXR Lisp** arena, the arguments have to be examined one more time, because two-way communication is possible, and because some of the material has temporary, dynamically allocated buffers associated with it which must be released. For instance a structure passed by pointer may be updated by the foreign function. FFI needs to propagate the changes which the foreign function performed to the C version of the structure, back to the original Lisp structure. Furthermore, a structure passed by pointer uses a dynamically allocated buffer. This buffer must be freed. The operation which handles the responsibility for propagating argument data back into **TXR Lisp** objects, and frees any temporary memory that had been arranged by the *put* operation is the *in* operation.

The in operation has two nuances: the by-value nuance and the by-pointer nuance. Data passed into a function by value such as function arguments or via `ptr-in` are subject to the by-value nuance. Updates to the foreign representation of these objects does not propagate back to the Lisp representation; however, those objects may contain pointers requiring the by-pointer nuance of the in operation of those pointers to be invoked.

10.2.3 The *get* operation

After a foreign call completes, it is also necessary to retrieve the call's return value, convert it to a Lisp object, and free any dynamic memory. This is performed by the *get* operation.

The *get* operation is also used by a Lisp callback function, called from a foreign library, to convert the arguments to Lisp objects.

10.2.4 The *out* operation

When a Lisp callback invoked by a foreign library completes, it must provide a return value, and also update any argument objects with new values. The return value is propagated using the put operation. Updates to arguments are performed by the `out` operation. This operation is like the reverse of the in operation. Like that operation, it has a by-value and by-pointer nuance.

For instance, if a callback receives a structure by value, upon return, there is no use in reconstructing a new version of the structure from the updated Lisp structure; the caller will not receive the change. However, if the structure contains pointers to data that was updated, by the callback, those changes must materialize. This is achieved by triggering the by-value nuance of the structure type's out operation, which will recursively invoke the out operation of embedded pointers, which will in turn invoke the by-pointer nuance.

10.3 The FFI Type System

The FFI type system consists of a notation built using Lisp syntax. Basic, unparametrized types are denoted by symbolic atoms. Similarly to a concept in the C language, `typedef` names can be globally defined, using the `ffi-typedef` function, or the `typedef` macro.

Like in the C language, `typedef` names are aliases for an existing type, and not distinct types. However, this is of no consequence, since the FFI doesn't perform any type checking between two foreign types, and thus never takes into consideration whether two such types are equal. The main concern in FFI is correspondence between Lisp values and foreign types. For instance, a Lisp string argument will not convert to a foreign function parameter of type `int`.

Compound expressions denote the construction of derived types, or types which are instantiated with parameters. Each such expression has a type constructor symbol in the operator position, from a limited, fixed vocabulary, which cannot be extended.

Some constituents of compound type syntax are expressions which evaluate to integer values: the dimension value for array types, the size for buffers, the width for bitfields and the value expressions for enumeration constants are such expressions. These expressions allow full use of **TXR Lisp**. They are evaluated without visibility into any apparent surrounding lexical scope.

Some predefined types which are provided are in fact typedef names. For instance, the `size-t` type is a typedef name for some other integral type, defined in a platform-specific way. Which type that is may be determined by passing the syntax to the type compiler function using the expression `(ffi-type-compile 'size-t)`. The type compiler converts the `size-t` syntax to the compiled type object, resolving the typedef name to the type which it denotes. The printed representation of that object reveals the identity of the type. For instance, it might be `#<ffi-type uint>`, indicating that `size-t` is an alias for the `uint` basic type, which corresponds to the C type `unsigned int`.

10.4 Simple FFI Types

10.4.1 FFI types `char`, `zchar`, `uchar` and `bchar`

These first two of these types, `char` and `zchar` correspond to the C character type `char`. The `uchar` and `bchar` types correspond to `unsigned char`. Both Lisp integers and character values convert to these representations if they are in their numeric range. Out-of-range values produce an exception. A foreign `char`, `zchar`, and `bchar` value converts to a Lisp character, whereas a `uchar` value converts to an integer.

If these types are used for representing individual scalar values, there is no difference among `char`, `zchar` and `bchar`.

What is different among these three types is that the `array` and `zarray` type constructors treat them

specially. Arrays of these types are subject to conversion to and from Lisp strings. The variation among these types expresses different conversion semantics. That is to say, an array of `bchar` converts between the foreign and native Lisp representation differently from an array of `zchar`, which in turn converts differently from an array of `char`.

Note: it is recommended to avoid using the types `bchar` and `zchar` other than for expressing the element type of an `array` or `zarray`.

10.4.2 FFI types `short`, `ushort`, `int`, `uint`, `long` and `ulong`

These types correspond to the C integer types `short`, `unsigned short`, `int`, `unsigned int`, `long` and `unsigned long`. Lisp characters and integers convert to these foreign representations, if they are in their numeric range. Foreign values of these types convert to Lisp integers.

10.4.3 FFI types `longlong` and `ulonglong`

These types are `typedef` names for integer types whose representation corresponds to the C types `long long` and `unsigned long long`.

10.4.4 FFI types `int8` and `uint8`

These types correspond to 8-bit signed and unsigned integers. They convert like integer types: both Lisp integers and characters convert to these types, if in a suitable range; and under the reverse conversion, the foreign values become Lisp integers.

10.4.5 FFI types `int16`, `uint16`, `int32`, `uint32`, `int64` and `uint64`

These types correspond denote precisely sized C integer types. They convert like integer types: both Lisp integers and characters convert to these types, if in a suitable range; and under the reverse conversion, the foreign values become Lisp integers.

10.4.6 FFI types `float` and `double`

These types correspond to the same-named C types. They convert Lisp integers, characters and floating-point numbers to these C types. Because the **TXR Lisp** `float` is represented as a C `double` it converts directly to `double` without the possibility of range error or loss of precision. A conversion to type `float` is subject to a range check; an exception is thrown if the Lisp floating-point value is out of range of this type. Even when the conversion is possible, it alters the value, results in a loss of precision. In the reverse direction, values of both types convert to the one and only **TXR Lisp** `float` type.

10.4.7 FFI type `bool`

The type `bool` is a typedef name for the `uchar` instance of the parametrized `bool` type, which is to say, `(bool uchar)`.

10.4.8 FFI type `val`

The FFI `val` type denotes the machine representation of a Lisp value cell, which is corresponds to a C pointer. Not all cell values are actually pointers, but values that are heap objects, such as vectors and conses, are. The `val` type transparently converts any Lisp object to a foreign pointer value with no representation change at all; and performs the reverse conversion from pointer to Lisp value.

Note: this is utterly dangerous. Lisp values that aren't pointers must not be dereferenced by foreign code. Foreign code must not generate Lisp pointer values that aren't objects which came from a Lisp heap. Interpreting a Lisp value in foreign code requires a correct decoding of its type tag, and, if necessary, stripping

the tag bits to recover a heap pointer and interpreting the type code stored in the heap object.

The conversion from foreign bit pattern to Lisp value is subject to a validity checks; an exception will be thrown if the bit pattern isn't a valid Lisp object. Nevertheless, the checks has cases which report as false positives: admit some invalid objects may be admitted into the Lisp realm, possibly with catastrophic results.

10.4.9 FFI type `cptr`

This type corresponds to a foreign pointer of any type, including a pointer to a function.

The `cptr` type converts between a foreign pointer and a Lisp object of type `cptr`.

Lisp objects of type `cptr` are tagged with a symbolic tag, which may be `nil`.

The unparametrized `cptr` converts foreign pointers to `cptr` objects which are tagged with `nil`.

In the reverse direction, it converts `cptr` Lisp objects of type `cptr` to foreign pointer, without regard for their type tag.

There is a parametrized version of the `cptr` FFI type, which provides a measure of type safety.

Note: the `cptr` type, in the context of FFI, is particularly useful for representing C pointers that are used in C library interfaces as "opaque" handles. For instance a FFI binding for the C functions `fopen` and `fclose` may use the `cptr` to represent the `FILE *` type. That is to say, `cptr` can be specified as the return type for `fopen`, thereby capturing the stream handle in a `cptr` object when that function is invoked through FFI. Then, the captured `cptr` object can be passed as the argument of `fclose` to close the stream.

10.4.10 FFI types `str`, `str-d` and `str-s`

These FFI types correspond to the C pointer type `char *`, providing automatic conversion between Lisp strings and null-terminated C strings. The null pointer corresponds to the **nil** symbol.

The related types `bstr`, `bstr-d`, `bstr-s`, `wstr`, `wstr-d` and `wstr-s` are also provided; these are described in the following sections.

The `str` type behaves as follows. The put operation allocates, using `malloc`, a buffer large enough to hold the UTF-8 encoded version of the Lisp string, encodes the string into that buffer, and then stores the `char *` pointer into the argument space. The in operation deallocates the buffer. If `str` is passed by pointer, the in operation also takes the current value of the `char *` pointer, which may have been replaced by a different pointer, and creates a new Lisp string by decoding UTF-8 from that buffer. The get operation retrieves the C pointer and duplicates a new string by decoding the UTF-8 contents. The type has no out operation: a string is not expected to be modified in-place.

The type `str-d` type differs in behavior from `str` as follows. Firstly, it has no in operation. Consequently, `str-d` doesn't deallocate the buffer that had been allocated by put. Under the get operation, the `str-d` type assumes that ownership over the C pointer has been granted, and after duplicating a new string from the decoded UTF-8 data in the C string, it deallocates that C string by invoking the C library function `free` on it.

Type type `str-` is similar to `str-d`; it also has no in-operation, and doesn't deallocate the buffer allocated in the put operation. Under the get operation, the `str-s` type does not assume ownership of memory, and therefore does not free the pointer received from the foreign function. The `str-s` type is intended for receiving strings via a pointer-to-pointer argument, in situations when the string must not be freed.

Like other types, the string types combine with the `ptr` type family. Because the `ptr` family has memory management semantics, as does the string family, it is important to understand the memory management implications of the combination of the two.

The derived pointer types `(ptr str-d)` and `(ptr str)` are effectively equivalent. They denote a string passed by pointer, with in-out semantics. The effect is that the string is dynamic in both directions. What that means is that the foreign function either must not free the pointer it was given, or else it must replace it with one which the caller can also free (or with a null pointer). The two are equivalent because `str-d` has no in operation, so its get operation is used instead; but that operation is similar to the in operation of the `str` type: both decode the string currently referenced by the `char *` pointer, and then pass that pointer to the C `free` function.

Receiving a string by pointer from a foreign function is achieved by treating the situation as a pointer to an array of one element. So that is to say, an argument like `char **pstr` can be treated as either `(ptr-out (array 1 str-d))` if the foreign function passes ownership of the string, or else `(ptr-out (array 1 str-s))` if the foreign function retains ownership of the string. In either case, the argument is a vector of one element, which will be updated to the returned string, or else `nil` if the function passes back a null pointer.

The type combination `(ptr-in str-d)` refers to a string pointer passed to a foreign function by pointer, whereby the foreign function will retain and free the pointer. The type combination `(ptr-in str)` passes the string pointer in the same way, but the foreign module mustn't use the pointer after returning. FFI will free the pointer that had been passed.

10.4.11 FFI types `bstr`, `bstr-d` and `bstr-s`

The `bstr` family corresponds to null-terminated `char *` C strings, like the `str` family, and the family members have memory management semantics similar to their `str` counterparts. Likewise, under these types also, the null pointer corresponds to `nil`.

The `b` prefix in the naming denotes "byte". It indicates that unlike the `str` family, the `bstr` family does not use UTF-8 encoding; only Lisp strings which contain strictly code points in the range U+0000 to U+00FF may convert to these types; out-of-range characters trigger an error exception.

Likewise, in the reverse direction, no UTF-8 decoding is performed: every byte value turns into the corresponding character code. The byte 0 is interpreted as the string terminator.

Note: the `bstr` type may be advantageous in situations when character handling is known to be confined to the ASCII range, since UTF-8 conversion is then unnecessary overhead. Because **TXR** strings use wide characters internally, converting to and from the `bstr` type still requires memory management overhead, just like in the case of the `str` type. The `wstr` type described in the next section avoids memory management and conversion overhead. Thus, even in situations in which characters are confined to the ASCII range, if wide functions are available in the foreign API, it may be more efficient to use them, particularly if the foreign component uses that representation internally.

10.4.12 FFI types `wstr`, `wstr-d` and `wstr-s`

The FFI type `wstr` corresponds to the C type `wchar_t *` pointing to the first character of a null terminated wide string. It converts between Lisp strings and symbols, and C strings. The family members of `wstr` have memory management semantics similar to their `str` counterparts, Likewise, under these types also, the null pointer corresponds to `nil`.

Note: because wide characters do not require UTF-8 conversion, the `wstr` family is more efficient. A `wstr` string passes into foreign code directly: the Lisp object already contains a null-terminated wide character string, and so the pointer to that is given directly to the foreign code. Similarly, ownership transfer in

either direction is a pointer passage with no memory management or conversion overheads. Whenever some foreign API offers a choice between UTF-8 strings, and wide strings, the wide version should be targeted by FFI, particularly if the API is known to works with wide strings internally also.

10.4.13 FFI types `buf` and `buf-d`

The `buf` type creates a correspondence between the **TXR Lisp** `buf` type and a C pointer to a block of arbitrary data. Note that there also exists a parametrized version of the `buf` and `buf-d` type syntax which specifies a size.

Under the `buf` type's put operation, no memory allocation takes place. The pointer to the buffer object's data is written into the argument space, so the foreign function can manipulate the buffer directly. If the object isn't a buffer but rather the symbol `nil`, then a null pointer is written.

The `buf` in operation has semantics as follows. In the pass-by-pointer nuance, the buffer pointer currently in the argument space is compared to the original one which had been written there from the buffer object. If they are identical, then the in operation yields the original buffer object. Otherwise, if the altered pointer is non-null, it allocates a new buffer equal in size to the original one and copies in the new data from the new pointer that was placed into the argument space by the foreign function. If the altered pointer is null, then instead of allocating a new buffer, the object `nil` is returned. The by-value nuance of the in operation does nothing.

The get operation is not meaningful for an unsized `buf`: it yields a zero length `buf` object. For this reason, parametrized `buf` type should be used for retrieving a buffer with a specific fixed size.

The `buf-d` type has different memory management from `buf`. The put operation of `buf-d` allocates a copy of the buffer and writes into the argument space a pointer to the copy. It is assumed that the foreign function takes ownership of the copy.

The in operation of `buf-d` is also different. The by-value nuance of the in operation is a no-op, like that of `buf`. The by-pointer nuance doesn't attempt to compare the previously written pointer to the current value. Rather, it assumes that if there is any non-null pointer value in the argument space, then it should take ownership of that object and return it as a new buffer. Thus if two-way dynamic buffer passing is requested using `(buf buf-d)` it means that the foreign function must replace the pointer with a null to indicate that it has consumed the buffer. Any non-null value in the argument space indicates that the foreign function has either rejected the pointer (not taken ownership), or has replaced it with a new object, whose ownership is being passed.

Unidirectional by-pointer passing of a `buf-d` can be performed using the types `(ptr-out buf-d)` or `(ptr-int buf-d)`. The former type will not invoke `buf-d`'s put operation. It will only allocate a pointer-sized space, without initializing it. After the foreign call, the by-pointer semantics of the in operation will be triggered If the foreign function places a non-null pointer into the space, its ownership will be seized by a newly instantiated buffer object. Otherwise the function must place a null pointer, which results in a `nil` value emerging from the in operation as documented above. The latter type will achieve a transfer of ownership in the other direction, by invoking the `buf-d` put operation, which places a copy of the buffer into the pointer-sized location prepared in the argument space. After the call, it will invoke the by-value in semantics of `buf-d`, which is a no-op: thus no attempt is made to extract a buffer, even if the foreign function alters the pointer.

10.4.14 FFI type `closure`

The `closure` type converts three kinds of Lisp objects to a C pointer: the object `nil`, the `cptr` type, or the special `ffi-closure` type.

When the `nil` symbol is converted to a `closure` type, it becomes a null function pointer.

A `cptr` object of any kind converts to a `closure`; the internal pointer is converted to a function pointer.

Instances of the `ffi-closure` type are produced by the `ffi-make-closure` function, or by calls to functions defined by the `deffi-cb` macro. The `closure` type is useful for passing callbacks to foreign functions: Lisp functions which appear to be C functions to foreign code.

In the reverse direction, when a `closure` object is converted from the foreign function pointer representation to a Lisp object, it becomes a `cptr` object whose tag is the `closure` symbol.

10.4.15 FFI type `void`

The `void` type is useful for indicating the return type of foreign functions and callbacks which return no value. It corresponds to a zero-sized object. It will convert any lisp value into zero bytes, and convert zero bytes into `nil`.

10.5 Parametrized FFI Type Operators

The following following parametrized type operators are available.

10.5.1 FFI type `enum`

Syntax:

```
(enum name {(sym value) | sym}*)
```

Description:

The type `enum` specifies an enumerated type, which establishes a correspondence between a set of Lisp symbols and foreign integer values of type `int`.

The *`name`* argument must either be `nil` or a symbol for which the `bindable` function returns true. It gives the tag name of the enumerated type. The remaining arguments specify the enumeration constants.

In the enumeration constant syntax, each occurrence of *`sym`* They must be a bindable symbol according to the `bindable` function. The symbols may not repeat within the same enumerated type. Unlike in the C language, different enumerations may use the same symbols; they are in separate spaces.

If a *`sym`* is given, it is associated with an integer value which is one greater than the integer value associated with the previous symbol. If there is no previous symbol, then the value is zero. If the previous symbol has been assigned the highest possible value of the FFI `int` type, then an error exception is thrown.

If `(`*`sym value`*`)` is given, then *`sym`* is given the specified value. The *`value`* is an expression which must evaluate to an integer value in range of the FFI `int` type. It is evaluated in an environment in which the previous symbols from the same enumeration appear as variables whose bindings are their enumeration values, making it possible to use earlier enumerations in the definition of later enumerations.

The FFI `enum` type converts two kinds of Lisp values to the foreign type `int`: symbols which are in the set defined by the type, and integer values which are in the range which that foreign type can represent. Out-of-range integer values, symbols not defined in the enumeration, and objects not of symbol or integer type all trigger an exception.

In the reverse direction, the `enum` type extracts from the foreign representation values of FFI type

`int`, and converts them, if possible, to symbols. If an integer value occurs which is not assigned to any enumeration symbol, then the conversion produces that integer value itself rather than a symbol. If an integer value occurs which is assigned to multiple enumeration symbols, it is not specified which of those symbols is produced.

10.5.2 FFI type `enumed`

Syntax:

```
(enumed type name {(sym value) | sym}*)
```

Description:

The `enumed` type operator is a generalization of `enum` which allows the base integer type of the enumeration to be specified. The following equivalence holds:

```
(enum n a b c ...)  <-->  (enumed int n a b c ...)
```

Any integer type or *`typedef`* name may be specified for *`type`*, including any one of the endian types. The enumeration inherits its size, alignment and other foreign representation details from *`type`*.

The values associated with the enumeration symbols must be in the representation range of *`type`*, which is not checked until the conversion of a symbol through the enumeration is attempted at run time.

The `enumed` type is a clone of the underlying type, inheriting most of its properties. In particular, it is possible to derive an `enumed` type from an underlying bitfield type. The resulting type is still a bitfield, and may only be used as a `struct` or `union` member. Moreover, because it is a bitfield type, there is a restriction against creating aliases for it with `typedef`.

An `enumed` bitfield allows the values of a bit field to be specified symbolically.

10.5.3 FFI type `struct`

Syntax:

```
(struct name {(slot type [init-form])}*)
```

Description:

The FFI `struct` type maps between a Lisp `struct` and a C `struct`. The *`name`* argument of the syntax gives the structure type's name, known as the tag. If this argument is the symbol *`nil`* then the structure type is named by a newly generated uninterned symbol (with `gensym`).

The *`name`* is entered into a global namespace of tags which is shared by structures and unions.

The *`name`* also specifies the Lisp `struct` name associated with the FFI type.

The *`slot`* and *`type`* pairs specify the structure members. The *`slot`* elements must be symbols, and the *`type`* elements must be FFI type expressions.

A `struct` definition with no members refers to a previously defined `struct` or `union` type which has the same *`name`* in the global `struct`/`union` tag space.

If no prior `struct` or `union` exists, then a definition with no slots specifies a new, incomplete structure type. A `struct` definition with no members never causes a Lisp structure type to be created.

A `struct` definition that specifies one or more members either defines a new structure type, or completes an existing one. If an incomplete structure or union type which has the same *name* exists, then the newly appearing definition is understood to provide a completion of that type. If the incomplete type is a `union`, it thereby converted to a `struct` type.

If a complete structure type which has the same *name* already exists, then the newly appearing definition replaces that type in the tag namespace.

A `struct` definition with members is entered into the `struct/union` tag space immediately as an incomplete type (if it isn't already), before the members are processed. Therefore, the member definitions can refer to the `struct` type. The type becomes complete when the last member is processed, except in the special situation when that member causes the type to become a flexible structure, described several paragraphs below.

A `struct` definition that specifies members causes a Lisp `struct` having the same `name` to exist, if such a type doesn't already exist. If such a type is created, instance slots are defined for it which correspond to the member definitions in the FFI `struct` definition.

For any *slot* which specifies an *init-form* expression, that expression is evaluated during the processing of the type syntax, in the global environment. The resulting value then becomes the initial value for the slot. The semantics of this value is similar to that of a quoted object appearing as an *init-form* in the `defstruct` macro's *slot-specifier* syntax. For example, if the type expression `(struct s (a int expr))`, which specifies a slot `a` initialized by `expr`, generates a Lisp struct type, the manner in which that type is generated will resemble that of `(defstruct s nil (a (quote [value-of-expr])))` where `[value-of-expr]` denotes the substitution of the value of `expr` which had been obtained by evaluation in the global environment. Note: if more flexible initialization semantics is required, the application must define the Lisp struct type first with the desired characteristics, before processing the FFI struct type. The FFI struct type will then related to the existing Lisp struct type.

Those members whose *slot* name is specified as `nil` is ignored; no instance slots are created in the Lisp type. If a *init-form* is specified for such a slot, there exists is no situation in which that form will be evaluated.

When a Lisp object is converted to a struct, it must, firstly, be of the struct type specified by *name*. Secondly, that type must have all of the slots defined in the FFI type. The slots are pulled from the Lisp structure in the order that they appear in the FFI `struct` definition. They are placed into the target memory area in that order, with all required padding between the members, and possibly after the last member, for alignment.

Whenever a member is defined using `nil` as the *slot* name, that member represents anonymous padding. The corresponding *type* expression is used only to determine the size of the padding only. Its data transfer semantics is completely suppressed. When converting from Lisp, the anonymous padding member simply generates a skip of the number of byte corresponding to the size of its type, plus any necessary additional padding for the alignment of the subsequent member.

Structure members may be bitfields, which are described using the `ubit`, `sbit` and `bit` compound type operators.

A structure member must not be an incomplete or zero-sized array, unless it is the last member. If the last member of FFI structure is an incomplete array, then it is a flexible structure.

A structure member must not be a flexible structure, unless it is the last member; the containing structure is then itself a flexible structure.

Flexible structures correspond to the C concept of a "flexible array member": the idea that the last member of a structure may be an array of unknown size, which allows for variable-length data at the end of a structure, provided that the memory is suitably allocated.

Flexible structures are subject to special restrictions and requirements. See the section Flexible Structures below. In particular, flexible structures may not be passed or returned by value.

See also: the `make-zstruct` function and the `znew` macro.

10.5.4 FFI type `union`

Syntax:

```
(union name {(slot type)}*)
```

Description:

The FFI `union` type resembles the `struct` type syntactically. It provides handling for foreign objects of C `union` type.

The *`name`* argument specifies the name for the union type, known as a tag. If this argument is the symbol *`nil`* then the union type is named by a newly generated uninterned symbol (with `gen-sym`).

The *`name`* is entered into a global namespace of tags which is shared by structures and unions.

The *`slot`* and `type` pairs specify the union members. The *`slot`* elements must be symbols, and the *`type`* elements must be FFI type expressions.

A *`union`* definition with no member refers to a previously defined `struct` or `union` type which has the same *`name`* in the global `struct/union` tag space.

If no prior `struct` or `union` exists, then a definition with no slots specifies a new, `union` type that is incomplete.

A *`union`* definition that specifies one or more members either defines a new structure type, or completes an existing one. If an incomplete structure type which has the same *`name`* exists, then the newly appearing definition is understood to provide a completion of that type. If the prior incomplete type is a `struct`, it is converted to `union` type. If a complete structure or union type which has the same *`name`* already exists, then the newly appearing definition replaces that type in the tag namespace.

A struct `union` definition with members is entered into the `struct/union` tag space immediately as an incomplete type (if it isn't already), before the members are processed. Therefore, the member definitions can refer to the `union` type. The type becomes complete when the last member is processed.

Unlike the FFI `struct` type, the `union` type doesn't provide automatic conversion between C and Lisp data. This is because the `union` is inherently unsafe, due to its placement of multiple types into the same storage, and lack of any information to discriminate which type is currently stored. Instead, the FFI `union` creates a correspondence between a C union that is regarded as just a region of memory, and a **TXR Lisp** data type called `union`.

An instance of the Lisp `union` type holds a copy of the C union memory, and also contains type information about the unions members. Functions are provided to store and retrieve the members; it is these functions which provide the conversion between the Lisp types and the foreign

representations stored in the C union. This is done under control of the application, because due to the inherent lack of safety of the C `union`, only the application program knows which member of the union may be accessed.

Conversion between the C `union` and the Lisp `union` consists of just a memory copying operation.

The following functions are provided for manipulating unions: `make-union` instantiates a new union object; `union-members` retrieves a list of the symbols serving as the union's member names; `union-get` retrieves a specified member from the union's storage, converting it to a Lisp object; `union-put` places a Lisp object into a union, using the specified member's type to convert it to a foreign representation; `union-in` performs the "in semantics" on the specified member of a union, propagating modifications in that member back to a Lisp object; and `union-out` performs "out semantics" on the specified member of a union, propagating modifications done on a previously retrieved Lisp object back into the union.

10.5.5 FFI type `array`

Syntax:

```
(array dim type)
(array type)
```

Description:

The FFI `array` type creates a correspondence between Lisp sequences and "by value" fixed size arrays in C. It converts Lisp sequences to C arrays, and C arrays to Lisp vectors.

Arrays passed by values do not exist in the C language syntax. Rather, the C type which corresponds to the FFI array is a C array that is encapsulated in a `struct`. For instance the type `(array 3 char)` can be visualized as corresponding to the C type `struct { char anonymous[3]; }`.

Thus, in the FFI syntax, we can specify arrays as function parameters passed by value and as return values.

On conversion from Lisp to the foreign type, the FFI `array` simply iterates over the Lisp sequence, and performs an element for element conversion to *`type`*.

If the sequence is shorter than the array, then the remaining elements are filled with zero bits. If the sequence is longer than the array, then the excess elements in the sequence are ignored.

Since Lisp arrays and C arrays do not share the same representation, temporary buffers are automatically created and destroyed by FFI to manage the conversion.

The *`dim`* argument is an ordinary Lisp expression expanded and evaluated in the top-level environment. It must produce a nonnegative integer value.

In addition, several types are treated specially: when *`type`* is one of `char`, `zchar`, `bchar` or `wchar`, the array type establishes a special correspondence with Lisp strings. When the C array is decoded, a Lisp string is created or updated in place to reflect the new contents. This is described in detail below.

The second form, whose syntax omits the *`dim`* element, it denotes a variable length array. It corresponds to the concept of an incomplete array in the C language, except that no implicit array-to-pointer conversion concept is implemented in the FFI type system. This type may not be used as

an array element or structure member, other than as the last structure member. It also may not be passed or returned by value, only by pointer. If the last member of a structure has this type, then it is a flexible array member; see the Flexible Structures section below.

Since the type has unknown length, it has a trivial get operation which returns `nil`. It is useful for passing a variable amount of data into a foreign function by pointer.

An array of `char` represents non-null-terminated UTF-8 character data, which converts to and from a Lisp string. Any null bytes in the data correspond to the pseudo-null character `#\xDC00` also notated as `#\pnul`.

An array of `zchar` represents a field of optionally null-terminated UTF-8 character data. If a null byte occurs in the data then the text terminates before that null byte, otherwise the data comprises the entire foreign array. Thus, null bytes do not occur in the data. A null byte in the array will not generate a pseudo-null character in the Lisp string.

An array of `bchar` values represents 8-bit character data that isn't UTF-8 encoded, and is not null terminated. Each byte holds a character whose code is in the range 0 to 255. If a null byte occurs in the data, is interpreted as a string terminator.

10.5.6 FFI type `zarray`

Syntax:

```
(zarray dim type)
(zarray type)
```

Description:

The `zarray` type is a variant of `array`. When converting from Lisp to C, it ensures that the array is null-terminated. This means that if the *zarray* is dimensioned, then the `[dim - 1]` element of the C array is written out as all zero bytes, ignoring the corresponding Lisp value in the Lisp array. If the *zarray* is undimensioned, then the size of the C array is deemed to be one greater than the actual length of the Lisp array. The elements in the Lisp array are converted to the corresponding elements of the C array, and then the last element of the C array is filled with null bytes. The `zarray` type is useful for handling null terminated character arrays representing strings, and for null terminated vectors. Unlike `array`, `zarray` allows the Lisp object to be one element short. For instance, when a `(zarray 5 int)` passed by pointer a foreign function is converted back to Lisp, the Lisp object is required to have only four elements. If the Lisp object has five elements, then the fifth one will be decoded from the C array in earnest; it is not expected to be null. However, when that Lisp representation is converted back to C, that extra element will be ignored and output as a zero bytes.

Lastly, the `zarray` further extends the special treatment which the `array` type applies to the types `zchar`, `char`, `wchar` and `bchar`. The `zarray` type assumes, and depends on the incoming data being null-terminated, and converts it to a Lisp string accordingly. The regular `array` type doesn't assume null termination. In particular, this means that whereas `(array 42 char)` will decode 42 bytes of UTF-8, even if some of them are null, converting those null bytes to the U+DC00 pseudo-null, in contrast, a `zarray` will treat the 42 bytes as a null-terminated string, and decode UTF-8 only up to the first null. In the other direction, when converting from Lisp string to foreign array, `zarray` ensures null termination.

Note that the type combination `zarray` of `zchar` behaves in a manner indistinguishable from a `zarray` of `char`.

The one-argument variant of the `zarray` syntax which omits the *dim* argument specifies a null-

terminated variant of the variable-length array. Like that type, it corresponds to the concept of an incomplete array in the C language. It may not be used as an array element, and may not be used as a structure member other than the last member. It cannot be passed as an argument or returned as a value. If the last member of a structure has this type, then it is a flexible array member; see the Flexible Structures section below.

Unlike the ordinary variable-length `array`, the `zarray` type supports the get operation, which extracts elements, accumulating them into a resulting vector, until it encounters an element consisting of all zero bytes. That element terminates the decoding, and isn't included in the resulting array.

The variable-length `zarray` also has a special in operation. Like the get operation, the in operation extracts all elements until a terminating null, decoding them to a vector. Then, the entire original vector is replaced with the new vector, even if the original vector is longer.

10.5.7 FFI type `ptr`

Syntax:

```
(ptr type)
```

Description:

The *`ptr`* denotes the passage of a value by pointer. The *`type`* argument gives the pointer's target type. The `ptr` type converts a single Lisp value, to and from the target type, using a C pointer as the external representation.

When used for passing a value to a foreign function, the `ptr` type has in-out semantics: it supports the interfacing concept that the called function can update the datum which has been passed to it "by pointer", thereby altering the caller's object. Since a Lisp value requires a conversion to the FFI external representation, it cannot be directly passed by pointer. Instead, this semantics is simulated. The put semantics of `ptr` allocates a temporary buffer, large enough to hold the representation of *`type`*. The Lisp value is then encoded into this buffer, recursively relying on the type's put semantics. After the foreign call, `ptr` triggers the in semantics of *`type`* to update the Lisp object from the temporary buffer, and releases the buffer.

The get semantics of `ptr` is used in retrieving a `ptr` return value, or, in a FFI callback, for retrieving the values of incoming arguments that are of `ptr` type. The get semantics assumes that the memory referenced by the C pointer is owned by foreign code. The Lisp object is merely decoded from the data area, which is then not touched.

The `out` semantics of `ptr`, used by callbacks for updating the values of arguments passed by pointer, assumes that the argument space already contains a valid pointer. The pointer is retrieved from the argument space, and the Lisp value is encoded into the memory referenced by that pointer.

Note that only Lisp objects with mutable slots can be meaningfully passed by pointer with in-out semantics. If a Lisp object without immutable slots, such as an integer, is passed using `ptr` the incoming updated value of the external representation will be ignored. Concretely, if a C function has the argument signature `(int *)` with in-out semantics such that it updates the `int` object which is passed in, this function can be called as a foreign function using a `(ptr int)` FFI type for the argument. However, the argument of the foreign call on the **TXR Lisp** side is just an integer value, and that cannot be updated.

On the other hand, if a FFI `struct` member is declared as of type `(ptr int)` then the Lisp `struct` is expected to have an integer-valued slot corresponding to that member. The slot is then

subject to a bidirectional transfer. FFI will create an `int`-sized temporary data area, encode the slot into that area and place that area's pointer into the encoded structure. After the call, the new value of the `int` will be extracted from the temporary buffer, which will then be released. The Lisp structure's slot will be updated with the new integer. This will happen even if the Lisp structure is being passed as a by-value argument.

10.5.8 FFI type `ptr-in`

Syntax:

```
(ptr-in type)
```

Description:

`ptr-in` type is a variation of `ptr` which denotes the passing of a value by pointer into a function, but not out. The put semantics of `ptr-in` is the same as that of `ptr`, but after the completion of the foreign function call, the in semantics differs. The `ptr-in` type only frees the temporary buffer, without decoding from it.

The out semantics of `ptr-in` differs also. It effectively treats the object as if it were "by value", since the reverse data transfer is ruled out. In other words, `ptr-in` simply triggers the by-value nuance of *`type`*'s out semantics.

The get semantics of `ptr-in` is the same as that of `ptr`.

10.5.9 FFI type `ptr-out`

Syntax:

```
(ptr-out type)
```

Description:

The `ptr-out` type is a variant of `ptr` which denotes a by pointer data transfer out of a function only, not into. The put semantics of `ptr-out` prepares a data area large enough to hold *`type`* and stores a pointer to that area into the argument space. The Lisp value isn't encoded into the data area.

The in semantics is the same as that of `ptr`: the by-pointer nuance of *`type`*'s in semantics is invoked to decode the external representation to Lisp data.

10.5.10 FFI type `ptr-in-d`

Syntax:

```
(ptr-in-d type)
```

Description:

The `ptr-in-d` type is a variant of `ptr-in` which transfers ownership of the allocated buffer to the invoked function. That is to say, the in semantics of `ptr-in-d` doesn't involve the freeing of memory that was allocated by put semantics.

The `ptr-in-d` type is useful when a function expects a pointer to an object that was allocated by `malloc` and expects to take responsibility for freeing that object.

Since the function may free the object even before returning, the pointer must not be used once the function is called. This is ensured by the in semantics of `ptr-in-d` which is the same as that of `ptr-in`.

The `ptr-in-d` type also has get semantics which assumes that ownership of the C object is to be seized. FFI will automatically free the C object when get semantics is invoked to retrieve a value through a `ptr-in-d`.

10.5.11 FFI type `ptr-out-d`

Syntax:

```
(ptr-out-d type)
```

Description:

The `ptr-out-d` type is a variant of `ptr-out` which is useful for capturing return values or, in a callback producing return values.

The `ptr-out-d` type has empty put semantics. If it put semantics is invoked, it does nothing: no area is allocated for *type* and no pointer is stored into the argument space.

The in semantics is the same as that of `ptr`: a pointer is retrieved from the argument space, the object is subject to *type*'s in semantics to recover the updated Lisp value, and then the object is freed.

The get semantics of `ptr-out-d` is identical to that of `ptr-in-d`.

The out semantics is identical to that of `ptr`.

10.5.12 FFI type `ptr-out-s`

Syntax:

```
(ptr-out-s type)
```

Description:

The `ptr-out-d` type is a variant of `ptr-out` similar to `ptr-out-d`, which assumes that the C object being received has an indefinite lifetime, and doesn't need to be freed. The suffix stands for "static".

Like `ptr-out-d`, the `ptr-out-s` has no put semantics.

Its in semantics recovers a Lisp value from the external object whose pointer has been stored by the foreign function, but doesn't free the external object.

The get semantics retrieves a Lisp value without freeing.

10.5.13 FFI type `bool`

Syntax:

```
(bool type)
```

Description:

The parametrized type `bool` can be derived from any integer or floating-point type. There is also an unparametrized `bool` which is a `typedef` for the type `(bool uchar)`.

The `bool` type family represents Boolean values, converting between a Lisp Boolean and foreign Boolean. A given instance of the `bool` type inherits all of its characteristics from *type*, such as its size, alignment and foreign representation. It alters the get and put semantics, however. The get semantics converts a foreign zero value of *type* to the Lisp symbol `nil`, and all other values to

the symbol `t`. The put semantics converts the Lisp symbol `nil` to a foreign value of zero. Any other Lisp object converts to the foreign value one.

The `bool` types are not integers, and cannot be used as the basis of bitfields: syntax like `(bit 3 (bool uint))` is not permitted. However, Boolean bitfields are possible when this syntax is turned inside out: the `bool` type can be derived from a bitfield type, as exemplified by `(bool (bit 3 uint))`. This simply applies the above described Boolean conversion semantics to a three-bit field. A zero/nonzero value of the field converts to `nil`/`t` and a `nil` or non-`nil` Lisp value converts to a 0 or 1 field value.

10.5.14 FFI types `ubit` and `sbit`

Syntax:

```
({ubit | sbit} width)
```

Description:

The `ubit` and `sbit` types denote C-language-style bitfields. These types can only appear as members of structures. A bitfield type cannot be the argument or return value of a foreign function or closure, and cannot be a foreign variable. Arrays of bitfields and pointers, of any kind, to bitfields are a forbidden type combination that is rejected by the type system.

The `ubit` type denotes a bitfield of type `uint`, corresponding to an `unsigned` bitfield in the C language.

The `sbit` type denotes a bitfield of type `int`. Unlike in the C language, it is not implementation-defined whether such a bitfield represents signed values; it converts between Lisp integers that may be positive or negative, and a foreign representation which is two's complement.

Bitfields based on some other types are supported using the more general `bit` operator, which is described below.

The *`width`* parameter of is an expression evaluated in the top-level environment, indicates the number of bits. It may range from zero to the number of bits in the `uint` type.

In a structure, bitfields produced by `sbit` and `ubit` are allocated out in storage units which have the same width and alignment requirements as a `uint`. These storage units themselves can be regarded as anonymous members of the structure. When a new unit needs to be allocated in a structure to hold bitfields, it is allocated in the same manner as a named member of type `uint` would be at the same position.

A zero-length bitfield is permitted. It may be given a name, but the field will not perform any conversions to and from the corresponding slot in the Lisp structure. Note that in situations when the FFI struct definition causes the corresponding Lisp structure type to come into existence, the Lisp structure type will have slots for all the zero width named bitfields, even though those slots don't participate in any conversions in conjunction with the FFI type.

The presence of a zero-length bitfield ensures that a subsequent structure member, whether bitfield or not, is placed in a new storage unit of the size of the bitfield's base type.

Details about the algorithm by which bitfields are allocated within a structure are given in the paragraph below entitled **Bitfield Allocation Rules**.

A `ubit` field stores values which follow a pure binary enumeration. For instance, a bitfield of width 4 stores values from 0 to 15. On conversion from the Lisp structure to the foreign structure,

the corresponding member must be a integer value in this range, or an error exception is thrown.

On conversion from the foreign representation to Lisp, the integer corresponding to the bit pattern is recovered. Bitfields follow the bit order of the underlying storage word. That is to say, the most significant binary digit of the bitfield is the one which is closest to the most significant bit of the underlying storage unit. If a four-bit field is placed into an empty storage unit and the value 8 its stored, then on a big-endian machine, this has the effect of setting to 1 the most significant bit of the underlying storage word. On a little-endian machine, it has the effect of setting bit 3 of the word (where bit 0 is the least significant bit).

The `sbit` field creates a correspondence between a range of Lisp integers, and a foreign representation based on the two's complement system. The most significant bit of the bitfield functions as a sign bit. Values whose most significant bit is clear are positive, and use a pure binary representation just like their `ubit` counterparts. The representation of negative values is defined by the "two's complement" operation, which maps each value to its additive inverse. The operation consists of temporarily treating the entire bitfield as unsigned, and inverting the logical value of all the bits, and then adding 1 with "wraparound" to zero if 1 is added to a field consisting of all 1 bits. (Thus zero maps to zero, as expected.) An anomaly in the two's complement system is that the most negative value has no positive counterpart. The two's complement operation on the most negative value produces that same value itself.

A `sbit` field of width 1 can only store two values: -1 and 0, represented by the bit patterns 1 and 0. An attempt to convert any other integer value to a `sbit` field of width 1 results in an error.

A `sbit` field of width 2 can represent the values -2, -1, 0 and 1, which are stored as the bit patterns 10, 11, 00 and 01, respectively.

10.5.15 FFI type `bit`

Syntax:

```
(bit width type)
```

Description:

The `bit` operator is more general than `ubit` and `sbit`. It allows for bitfields based on on any integer type up to 64 bits wide.

When the character types `char` and `uchar` are used as the basis of bitfields, they convert integer values, not characters. In the case of `char`, the bitfield is signed.

All remarks about `ubit` and `sbit` apply to `bit` also.

Details about the algorithm by which bitfields are allocated within a structure are given in the paragraph below entitled **Bitfield Allocation Rules**.

Under the `bit` operator, the endian types such as `be-int32` or `le-int16` may also be used as the basis for bitfields. If *`type`* is an endian type, the bitfield is then allocated in the same way that a bitfield of the corresponding ordinary type would be allocated on a target machine which has the byte order of that endian type.

When a bitfield member follows a member which has a different byte order, the bitfield is placed into a new allocation cell. This is true even if the previous member has the same alignment.

Note: the allocation of bits within a bitfield based on a byte storage cells also differs between different endian systems. However, the FFI type system does not offer one byte endian types such as

be-uint8. The workaround is to switch to a wider type.

Note: endian bitfields may be used to match the image of a C structure which contains bitfields, without having to conditionally define the FFI struct type differently based on whether the current machine is big or little endian. Conditionally defining a structure for two different byte orders adds verbiage to the program and is highly error-prone, since the bitfields change order within an allocation unit.

For instance, on a big endian system, the definition of a structure representing an IPv4 packet might begin like this:

```
(struct ipv4-header
  (ver (bit 4 uint16))
  (ihl (bit 4 uint16))
  (dscp (bit 6 uint16))
  (ecn (bit 2 uint16))
  (len uint16)
  ...)
```

to port this to a little endian system, the programmer has to recognize that the first pair of fields is packed into one byte, and the next pair of fields into a second byte. The bytes stay in the same order, but the pairs are reversed:

```
(struct ipv4-header
  (ihl (bit 4 uint16)) ;; reversed pair
  (ver (bit 4 uint16))
  (ecn (bit 2 uint16)) ;; reversed pair
  (dscp (bit 6 uint16))
  (len be-uint16)
  ...)
```

Endian bitfields allow this to be defined naturally. The IPv4 header is based on network byte order, which is big-endian, so big endian types are used. The little endian version above already uses be-uint16 for the *len* field. This just has to be done for the bitfields also:

```
(struct ipv4-header
  (ver (bit 4 be-uint16))
  (ihl (bit 4 be-uint16))
  (dscp (bit 6 be-uint16))
  (ecn (bit 2 be-uint16))
  (len be-uint16)
  ...)
```

10.5.16 FFI types buf and buf-d

Syntax:

({buf | buf-d} *size*)

Description:

The parametrized buf and buf-d types are variants of the unparametrized buf and buf-d, respectively. The *size* argument is an expression which is evaluated in the top-level environment, and must produce a nonnegative integer.

Because they have a size, these types have useful get semantics.

The get semantics of `buf-d` is that a Lisp object of type `buf` is created which takes direct ownership of the memory.

The get semantics of `buf` is that a Lisp object is created using a dynamically allocated copy of the memory.

10.5.17 FFI type `carray`

Syntax:

```
(carray type)
```

Description:

The `carray` type corresponds to a C pointer, in connection with the concept of representing a variable length array that is passed and returned as a pointer to the base element. On the Lisp side, the `carray` FFI type corresponds to the `carray` Lisp type. The `carray` Lisp type is similar to `cptr`, but supports array indexing operations, and some other features. It can be regarded as a semantic cross between `cptr` and `buf`.

The get semantics of `carray` is simply that a pointer is retrieved from memory and converted to a freshly allocated `carray` object which holds that pointer, and is marked as having an unknown size. No copy is made of the underlying array. When the application determines the size of the array, it can inform that object by calling the `carray-set-length` function.

The put semantics of the `carray` FFI type is simply to write, into the argument space, the pointer which the object holds. The object must be a `carray` whose element type matches that of the FFI type.

The `carray` type has in semantics. When a `carray` is passed to a foreign function as an argument to a `ptr` or `ptr-out` parameter to either a `carray` or `cptr` type, what is passed to the function is a pointer to the `carray`'s pointer. The foreign function may update this pointer to a new value, and this value is stored back into the `carray` object. The array's length is reset to zero. If it is an owned `carray`, arranged by `carray-own`, then the current array freed before the new pointer is assigned, and the object's type is reset to borrowed array. The `carray` object must not be memory mapped `carray` coming from the `mmap` function.

The `carray` type lacks out semantics, since Lisp code cannot change its address; so there is no new pointer to propagate back to a foreign caller which passes a `carray` to a Lisp callback, and no other memory management tasks to perform.

The `carray` type is particularly useful in situations when foreign code generates such an array, and the size of that array isn't known from the object itself.

It is also useful, instead of a variable-length `zarray`, for passing a dynamic array to foreign code in situations when the application benefits from managing the memory for the array. The variable-length `zarray` FFI type's disadvantage relative to `carray` is that the `zarray` converts an entire Lisp sequence to a temporarily allocated array, which is used only for one call. By contrast, the `carray` object holds the C representation which Lisp code can manipulate; and that representation is passed directly, just like in the case of `buf`.

Unlike `buf`, there is no dynamic variant of `carray`. The transfer of ownership of a `carray` requires the use of explicit operations like `carray-free` and `carray-own`.

It is possible to create a `carray` view over a buffer, using `carray-buf`.

Lastly, the `carray` type is the basis for the **TXR Lisp** `mmap` function, which is documented in the section **Unix Memory Mapping**.

10.5.18 FFI type `cptr`

Syntax:

```
(cptr type-sym)
```

Description:

The parametrized `cptr` type is similar to the unparametrized `cptr`. It also converts between Lisp objects of type `cptr` and foreign pointers. Unlike the unparametrized type, it provides a measure of type safety, and also supports the conversion of `carray` objects.

When a foreign pointer is converted to a Lisp object under control of the parametrized `cptr`, the resulting Lisp `cptr` object is tagged with the *`type-sym`* symbol.

In the reverse direction, when a Lisp `cptr` object is converted to the parametrized type, its type tag must match *`type-sym`*, or else the conversion fails with an error exception. This rule contains a slight relaxation: a `cptr` object with a `nil` tag can be converted to a foreign representation using any parametrized type, if its value is null. In other situations, the `cptr-cast` function must be used to coerce the pointer object to the matching type.

Note that if *`type-sym`* is specified as `nil`, then this is precisely equivalent to the unparametrized `cptr` which doesn't provide the above safety measure.

A `carray` object may also be converted to a foreign pointer under the control of a parametrized `cptr` type. The `carray` object's internal pointer becomes the foreign pointer value. The conversion is only permitted if the following two restrictions are not met, otherwise an error exception is thrown. Firstly, the *`type-sym`* of the `cptr` type must be the name of an FFI type, at the time when the `cptr` type expression is processed, otherwise the `cptr` is not associated with a type. Secondly, the `carray` object being converted must have an element type which matches the FFI type denoted by the `cptr` object's *`type-sym`*.

Pointer type safety is useful, because FFI can be used to create bindings to large application programming interfaces (APIs) in which objects of many different kinds are referenced using pointer handles. The erroneous situation can occur that a FFI call passes a handle of one kind to a function expecting a different kind of handle. If all pointer handles are represented by a single `cptr` type, then such a situation proceeds without diagnosis. If handles of different types are all mapped to `cptr` types with different tags, the situation is intercepted and diagnosed with an error exception.

10.5.19 FFI types `align` and `pack`

Syntax:

```
(align [width] type)
(pack [width] type)
```

Description:

The FFI type operators `align` and `pack` define a type which is a copy of *`type`*, but with adjusted alignment requirements. In some cases, `pack` (but not `align`) works by replacing itself with a transformed version of the `type` syntax.

If the *`width`* argument is present, it is an expression which is evaluated in the top-level environment. It must produce a positive integer which is a power of two.

If *width* is absent, a different default value is used depending on which type operator is specified. For `align`, it defaults to some platform-specific maximum useful alignment value, typically 16. For `pack`, a missing *width* defaults to 1.

The `align` operator can be used to create a version of *type* which is aligned at least as strictly as the specified *width*. That is to say, values of *width* which are less than or equal to *type*'s existing alignment have no effect on alignment, except when the type is used as a bitfield.

The `pack` operator can be used to create a version of *type* which is less strictly aligned than its existing alignment.

Alignment affects the placement of the type as a structure member, and as an array element.

A type with alignment 1, like the default alignment for `pack`, can be placed at any byte offset, and thus is effectively unaligned. A type with alignment 2 can be placed only at even addresses and offsets.

Alignment can be applied to all types, including arrays and structs. It may also be applied to bitfields, but special considerations have to be observed to obtain the intended effect, described below. However, out of the elementary types, only the integer and floating point types are required to support a weakening of alignment. Whether a type which corresponds to a pointer, such as a `str` or `buf`, can be written at an offset which doesn't meet that type's default alignment is machine-dependent.

If a FFI struct type is declared with a weakened alignment, whether or not such a structure can be read or written at the misaligned offsets depends on whether the individual members support it. If they are integer or floating-point types, or aggregates thereof, the usage is supported in a machine-independent manner.

Alignment interacts with the allocation of bitfields in special ways. If *width* is greater than 1, or regardless of *width* if the operator is `align`, the type is marked with a Boolean attribute indicating that it has altered alignment. Then, when a bitfield is based on a type which has altered alignment, then that bitfield isn't packed together with the previous field, even if the allocation rules otherwise call for it. Due to the alignment request, the byte offset is first adjusted according to the requested alignment and the bit offset is reset to zero. The bit field is then allocated at the new alignment. This requirement applies even if the requested alignment is 1, which is possible via a combination of both `pack` and `align`, both specified with a *width* of 1. If the requested alignment for the type of a bitfield is 1, and the previous member is a bitfield which has left a byte partially filled, then the new bitfield starts on a fresh byte, even if it would otherwise be packed with the previous bitfield. If a named bitfield has weakened alignment, other than one byte alignment produced by `pack`, the bitfield's original type's alignment is used for the purposes of determining its contribution to the alignment of the structure.

When *type* is one of two kinds of types, the `pack` type operator exhibits special behaviors, as follows. In these situations, the `pack` operator has no semantics other than these behaviors.

1. When *type* is `struct` or `union` syntax which defines at least one member, then the `pack` operator performs the following syntactic transformation: each member of *type* is edited, by specifying a `pack` operator around its type, with the given *width*. The surrounding `pack` operator is deleted. The effect is that `pack` is applied not to the struct or union type itself, but to its members. For example `(pack (struct s (x int) (y double))` is transformed into `(struct s (pack 1 (x int)) (pack 1 (y double)))`. The 1 comes from the defaulting of *width*.

   The rationale for this behavior is that alignment weakening is often required for all

members of a structure, rather than select members. Moreover, specifying weak alignment for a structure type itself, while leaving members with strict alignments, rarely makes sense. Weakening the alignment of a structure will not eliminate the padding between the members or at the end; it will only have any useful effect when that structure is itself used as the member of another structure. An important rationale also is that the GNU C `packed` attribute works this way, and so C structures declarations using that attribute are easier to translate to the **TXR Lisp** FFI type system.

Deriving a less strictly aligned version of a structure or union type without any effect on the alignment of its members may be obtained by applying the `bit` operator to either `typedef` name for a structure or union type, or else to syntax which refers to an existing type without defining members. Given the definition `(typedef s (struct s (a int) (b char)))`, the type `s` is an eight byte structure with three bytes of padding at the end, which has four byte alignment. The type expression `(pack s)` produces a version of this type which has one byte alignment. The expression `(pack (struct s))`, likewise. The resulting unaligned type is still eight bytes wide, and has three padding bytes. In other words, the `pack` operator does not transform the syntax of a structure which is already defined as an object,

2. When *`type`* is a `align` operation, then `pack` transforms the syntax as follows: the `pack` operator surrounding the `align` expression is removed, and introduced around the type expression that is `align`'s own operand. Thus `(pack 2 (align 16 int))` is transformed into `(align 16 (pack 2 int))`.

   The rationale for this transformation is that when both `align` and `pack` are applied to a type, the combination only makes sense when `pack` is first. For a non-structure type like `int`, `(pack x (align y int))` is equivalent to just `(pack x int)`, because `pack` will set the alignment to `x` regardless of the effect of `align`. Whereas `(align y (pack x int))` is meaningful in that the `align` takes precedence over `pack` if `(> y x)`. The main rationale is that `pack` may be applied to structure members via a code transformation. Those members may already have types which use `align`. This transformation ensures that the semantics is applied in a useful order. For example `(pack (struct s (a char) (x (align 2 int))))` is first transformed into `(struct s (a (pack 1 char)) (x (pack 1 (align 2 int)))))`. If this is left as-is, then the `align` on `x` is obliterated by the `pack`, rendering it useless. A further transformation takes place to `(struct s (a (pack 1 char)) (x (align 2 (pack 1 int)))))`. Now the `align` directive is increasing the alignment of `x` to 2, so that `x` will be placed at offset 2, leaving one byte of padding after the `a` member. This is how attributes work in GNU C also: the `aligned` attribute on the member of a packed structure can take precedence and increase its alignment.

After these transformations are applied, the nested `pack` forms which occur in the transformed syntax may perform more such transformations, depending on their operands.

Note that the two-argument form of `pack` with a *`width`* value greater than 1 doesn't directly correspond to any single attribute specifier in GNU C. The GNU C `packed` attribute is Boolean, implicitly reducing alignment to 1. A combination of the GNU C attributes `aligned` and `packed` is used to produce the effect of `(pack` *`n type`*`)` for values of $n > 1$. In GNU C, the `packed` attribute, when applied to a structure, distributes to its members, but isn't capable of distributing an alignment exceeding 1. So the `(pack` *`n`* `(struct ...))` expression, for values of $n > 1$, doesn't correspond to anything in GNU C; its effect can be simulated by attributing the structure type with `packed`, and then individually applying the required alignment to the member declarations.

10.6 Additional Types

10.6.1 FFI types `size-t`, `ptrdiff-t`, `int-ptr-t`, `uint-ptr-t`, `intmax-t`, `uintmax-t`, `wint-t` `sig-atomic-t`, `time-t` and `clock-t`

Description:

These additional FFI types for common C language types are provided as `typedef` aliases. The `intmax-t` and `uintmax-t` types are provided only if the host platform's `intmax_t` is no wider than 64 bits. If the host platform lacks `intmax_t` then the above two FFI types are defined as aliases for `longlong` and `ulonglong`, respectively.

10.6.2 FFI type `qref`

Syntax:

```
(qref struct-type member1 [member2 ...])
```

Description:

The FFI type operator `qref` provides a way to reference the type of a member of a struct or union. The *`struct-type`* argument must be a type expression denoting a struct or union. The *`member1`* argument and any additional arguments must be symbols.

If `S` is a struct or union type, and `M` is a member, then `(qref S M)` is a type expression denoting the type of `M`. Moreover, if `M` itself is a struct or union, which has a member named `N` then the type of `N` can be denoted by the expression `(qref S M N)`. Similarly, additional symbols reference through additional struct/union nestings.

Note: the referencing dot syntax can be used to write `qref` expressions. For instance, `(qref S M N)` can be written as `S.M.N` instead.

10.6.3 FFI type `elemtype`

Syntax:

```
(elemtype type)
```

Description:

The FFI type operator `elemtype` denotes the element type of *`type`*, which must be a pointer, array or enum.

Note: there is also a macro `elemtype`. The macro expression `(elemtype X)` is equivalent to the expression `(ffi (elemtype X))`.

10.6.4 FFI types `blkcnt-t`, `blksize-t`, `clockid-t`, `dev-t`, `fsblkcnt-t`, `fsfilcnt-t`, `gid-t`, `id-t`, `ino-t`, `key-t`, `loff-t`, `mode-t`, `nlink-t`, `off-t`, `pid-t`, `ssize-t`, `uid-t` and `socklen-t`

The additional names of various common POSIX types may also be available, depending on platform. They are provided as `typedef` aliases.

10.7 Endian Types

In addition to the type system described in the previous section. the FFI type system supports *endian types*, which are useful for dealing with data formats defined by networking protocols and other kinds of standards, or data structure definitions from other machines.

There are two kinds of *endianness*: *Little endian* refers to the least-significant byte of a data type being

stored at the lowest address in memory, lowest offset in a buffer, lowest offset in a file, or earlier byte in a communication stream. *Big endian* is the opposite: it refers to the most significant byte occurring at the lowest address, offset or stream position. For each of the signed integral types

`int16` through `int64`, the corresponding unsigned types `uint16` through `uint64`, and the two floating-point types `float` and `double`, the FFI type system provides a big-endian and little-endian version, whose names are derived by prefixing the `be-` or `le-` prefix to its related type.

Thus, the exhaustive list of the endian types is: `be-int16`, `be-uint16`, `be-int32`, `be-uint32`, `be-int64`, `be-uint64`, `be-float`, `be-double`, `le-int16`, `le-uint16`, `le-int32`, `le-uint32`, `le-int64`, `le-uint64`, `le-float` and `le-double`.

These types have the same size and alignment as their plain, unprefixed counterparts. Alignment can be overridden with the `align` type construction operator to create versions of these types with alternative alignment.

Endian types are supported as arguments to functions, return values, members of structs and elements of arrays.

TXR Lisp's FFI performs the automatic conversion from the abstract Lisp integer representation to the foreign representations exhibiting the specified endianness.

10.8 Incomplete Types

In the **TXR Lisp** FFI type system, the following types are *incomplete*: the type `void`, arrays of unspecified size, and any `struct` whose last element is of incomplete type.

An incomplete type cannot used as a function parameter type, or a return value type. It may not be used as an array element or union member type. A struct member type may be incomplete only if it is the last member.

An incomplete structure whose last member is an array is a *flexible structure*.

10.9 Flexible Structures

If a FFI `struct` type is defined with an incomplete array (an array of unspecified size) as its last member, then it specifies an incomplete type known as a *flexible structure*. That array is the *terminating array*. The terminating array corresponds to a slot in the Lisp structure; that slot is the *last slot*.

A structure which has a flexible structure as its last member is also, effectively, a flexible structure.

When a Lisp structure is being converted to the foreign representation under the control of a flexible structure FFI type, the number of elements in the terminating array is determined from the length of the object stored in the last slot of the Lisp structure. The length includes the terminating null element for `zarray` types. The conversion is consistent with the semantics of an incomplete array that is not a structure member.

In the reverse direction, when a foreign representation is being converted to a Lisp structure under the control of a flexible structure FFI type, the size of the array that is accessed and extracted is determined from the length of the object stored in the last slot, or, if the array type is a `zarray` from detecting null-termination of the foreign array. The conversion of the array itself is consistent with the semantics of an incomplete array that is not a structure member. Before the conversion takes place, all of the members of the structure prior to the terminating array, are extracted and converted to Lisp representations. The corresponding slots of the Lisp structure are updated. Then if the Lisp structure type has a `length` method, that method is invoked. The return value of the method is used to perform an adjustment on the object in the last slot. If

the existing object in the last slot is a vector, its length is adjusted to the value returned by the method. If the existing object isn't a vector, then it is replaced by a new `nil`-filled vector, whose length is given by the return value of `length`. The conversion of the terminating array to Lisp representation the proceeds after this adjustment, using the adjusted last slot object.

10.10 Bitfield Allocation Rules

The **TXR Lisp** FFI type system follows rules for bitfield allocation which were experimentally derived from the behavior of the GNU C compiler on several mainstream architectures.

The allocation algorithm can be imagined to walk through the structure from the first member to the last, maintaining a byte offset *O* which indicates how many whole bytes have been allocated to members so far, and a bit offset *B* which indicates, additionally, how many bits have been allocated in the byte which follows these *O* bytes, between 0 and 7.

When a non-bitfield member is placed, then there are two cases: either *B* is zero (only *O* bytes have been allocated, with no fractional byte) or else *B* is nonzero. In this latter case, *B* is reset to zero and *O* is incremented by one. In either case, *O* is adjusted up to the required alignment boundary for the new member. The member is placed, and *O* is incremented again by the size of that member.

When a bitfield member is placed, the algorithm considers the structure to be allocated in units of the base type of that bitfield member. For instance if the bitfield is derived from type `uint16` then the structure's layout is considered to have been allocated in `uint16` units. The algorithm examines the value of *O* and *B* to determine the first available unit in which at least one bit of unallocated space remains. Then, if the unit at that offset has enough space to hold the new bitfield, according to the bitfield's width, then the bitfield is placed into that unit. Otherwise, the bitfield is placed into the next available unit.

After a bitfield is placed, the values of *O* and *B* are adjusted so that *O* reflects the whole number of bytes which have been allocated to the structure so far, and *B* indicates the 0 to 7 additional bits of any bitfield material protruding past those whole bytes.

A zero-width bitfield is also considered with regard to the storage unit size indicated by its type. As in the case of the nonzero-width bitfield, the offset of the first available unit is found which has at least one bit of unallocated space. Then, if that unit is entirely empty, the zero-width bitfield has no effect. If that unit is partially filled, then *O* is adjusted to point to the next unit after that, and *B* is reset to zero. Note that according to this semantics, a zero-width bitfield can have an effect even if placed between non-bitfield members, or appears as the last member of a structure. Also, a structure containing only a zero-width bitfield has size zero.

If, after the placement of all structure members, *B* has a nonzero value, then the offset *O* is incremented by one to cover that byte.

As the last allocation step, the size of the structure is then padded up to a size which is a multiple of the alignment of the most strictly aligned member.

A named bitfield contributes to the alignment of the structure, according to its type, the same way as a non-bitfield member of the same type. An unnamed bitfield doesn't contribute alignment, or else may be regarded as having the weakest possible alignment, which is byte alignment. If all of the members of a structure are unnamed bitfield members of any type, it exhibits byte alignment.

The description isn't complete without a treatment of byte and bit order. Bitfield allocation follows an imaginary "bit endianness" whose direction follows the machine's byte order: most-significant bits are allocated first on big endian, least significant bits first on little endian.

If a one-bit-wide bitfield is allocated into a hitherto empty structure, it will be placed into the first byte of

that structure, regardless of the machine's endianness, and regardless of the underlying storage unit size for that bitfield. Within that first byte, it will be placed into the most significant bit position on a big-endian machine (bit 7); and on a little-endian machine, it will be placed into the least significant bit position (bit 0). If another one-bit-wide is allocated, it is placed into bit 6 on big endian, and bit 1 on little endian.

More generally, whenever a bitfield is allocated for a big-endian machine, and the storage unit is determined into which that bitfield shall be placed, the most significant bits of that storage unit are filled first on a big-endian machine, whereas the least significant bits are filled first on a little-endian machine. From this it follows that on either type of machine, that field shall be placed at the lowest-addressed byte or bytes in which unallocated bits remain.

10.11 Returning Scalar Objects by Pointer

There are situations in which an a foreign function takes the address of a storage location, and writes a new value into that location. Informally, this referred to as an "out parameter" or "in-out parameter", in the case of bidirectional data transfer. In the C language, the familiar pattern looks like this:

```
void function(int *ptr);
int val = 0;
function(&val);
```

In the case of an aggregate type, such as a structure, being an in-out or out parameter, this pattern is easily handled in FFI because the corresponding Lisp object is also an aggregate, and therefore has reference semantics: it can be updated to receive the new value. In the case of a scalar, however, such as `int` in the above example, this may not be not possible. A Lisp integer doesn't have the referential semantics required to receive a new value by pointer, and there is no "address-of" concept to create a reference to its location.

To understand the following FFI trick, it helps to first rework the C into a different form:

```
void function(int *ptr);
int val[1] = { 0 };
abc_function(val);
```

Instead of a scalar value, we can declare an array of 1 element of that same type, and pass the array (which converts into a pointer to that element). This approach inspires a similar trick in the FFI domain:

```
(with-dyn-lib (...)
  (deffi abc-function "abc_function" void ((ptr (array 1 int)))))

(let ((val (vec 0)))
  (abc-function val)

  ;; [vec 0] has updated value coming from function
  )
```

We define the parameter of `abc-function` as being a pointer to an array of 1 `int` rather than an int, and then pass a vector as the argument. If the parameter is in-out, then the vector must be constructed or initialized to contain a value that will convert to the C type. If the parameter is out only, then the FFI definition can use `ptr-out` and the vector can contain the `nil` value.

10.12 FFI Call Descriptors

The FFI mechanism makes use of a type-like representation called the "call descriptor". A call descriptor is an object which uses FFI types to describe function arguments and return values. A FFI descriptor is required to call a foreign function, and to create a FFI closure to use as a callback function from a foreign

function back into **TXR Lisp**.

A FFI descriptor object can be constructed from a return value type, and a list of argument types, and several other pieces of information using the function `ffi-make-call-desc`.

This object can then be passed to `ffi-call` to specify the C type signature of a foreign function, or to `ffi-make-closure` to specify the C type signature of a FFI closure to bind to a Lisp function.

The FFI macros `deffi` and `deffi-cb` provide a simplified syntax for expressing FFI call descriptors, which includes a notation for expressing variadic calls.

A note about variadic foreign functions: although there is support in the call descriptor mechanism for expressing a variadic function, it expresses a particular **instance** of a variadic function, rather than the variadic function's type per se. To call the same variadic function using different variadic arguments, different call descriptors are required. For instance to perform the equivalent of the C function call `printf("hello\n")` requires a certain descriptor. To perform the equivalent of `printf("hello, %s\n", name)` requires a different descriptor.

10.13 Foreign Function Type API

This group of functions comprises the basic interface to the **TXR Lisp**'s FFI type system module.

10.13.1 Function `ffi-type-compile`

Syntax:

```
(ffi-type-compile syntax)
```

Description:

The `ffi-type-compile` function produces and returns a compiled type object from a *syntax* argument which specifies valid FFI syntax. If the type syntax is invalid, or specifies a nonexistent type specifier or operator, an exception is thrown.

Note: whenever a function argument is required to be of FFI type, what it means is that it must be a compiled object, and not a Lisp expression denoting FFI syntax.

Examples:

```
(ffi-type-compile 'int) -> #<ffi-type int>
(ffi-type-compile
  '(array 3 double)) -> #<ffi-type (array 3 double)>
(ffi-type-compile 'blarg) -> ;; error
```

10.13.2 Function `ffi-make-call-desc`

Syntax:

```
(ffi-make-call-desc ntotal nfixed rettype
   argtypes [name])
```

Description:

The `ffi-make-call-desc` function constructs a FFI call descriptor.

The *ntotal* argument must be a nonnegative integer; it indicates the number of arguments in the call.

If the call denotes a variadic function, the *nfixed* argument must be an integer at least 1 and less than *ntotal*, denoting the number of fixed arguments. If the call denotes an ordinary, non-variadic function, then *nfixed* must either be specified specified as `nil` or else equal to the *ntotal* argument.

The *rettype* parameter must be an FFI type. It specifies the function return type. Functions which don't return a value are specified by the (compiled version of) the return type `void`.

The *argtypes* argument must be a list of types, containing at least *ntotal* elements. If the function takes no arguments, this list is empty. If the function is variadic, then the first *nfixed* elements of this list specify the types of the fixed arguments; the remaining elements specify the variadic arguments.

The *name* argument gives the name of the function for which this description is intended, or some other identifying symbol. This symbols is used in diagnostic messages related to errors in the construction of the descriptor itself or its subsequent use. If this parameter is omitted, then the involved FFI functions use their own names in reporting diagnostics.

Note: variadic functions must not be called using a non-variadic descriptor, and vice versa, even if the return types and argument types match.

Note: unlike the `deffi` and `deffi-cb` macros, the `ffi-make-call-desc` function doesn't perform any special treatment of variadic parameter types. When any of the types `float`, `be-float` or `le-float` occur in the variadic portion of *argtypes*, it is unspecified whether a descriptor is successfully produced and returned or whether an exception is thrown. If a descriptor is successfully produced, and then subsequently used for making or accepting calls, the behavior is undefined.

Example:

```
;;
;; describe a call to the variadic function
;;
;;   type void (*)(char *, ...)
;;
;; with these actual arguments
;;
;;   (char *, int)
;;
(ffi-make-call-desc
  2 ;; two arguments
  1 ;; one fixed
 (ffi-type-compile 'void)        ;; returns nothing
 (list (ffi-type-compile 'str)   ;; str -> char *
       (ffi-type-compile 'int))) ;; int
-->
#<ffi-call-desc #<ffi-type void>
  (#<ffi-type str> #<ffi-type int>)>
```

10.13.3 Function `ffi-type-operator-p`

Syntax:

```
(ffi-type-operator-p symbol)
```

Description:

The `ffi-type-operator-p` function return `t` if *`symbol`* is a type operator symbol: a symbol used in the first position of a recognized compound type form in the FFI type system.

Otherwise, it returns `nil`.

10.13.4 Function `ffi-type-p`

Syntax:

```
(ffi-type-p symbol)
```

Description:

The `ffi-type-p` function returns `t` if *`symbol`* denotes a type in the FFI type system: either a built-in type or an alias type name established by `typedef`.

Otherwise, it returns `nil`.

10.13.5 Function `ffi-make-closure`

Syntax:

```
(ffi-make-closure lisp-fun call-desc
                  [safe-p [abort-val]])
```

Description:

The `ffi-make-closure` function binds a Lisp function *`lisp-fun`*, which may be a lexical closure, or any callable object, with a FFI call descriptor *`call-desc`* to produce a FFI closure.

A FFI closure is an object of type `ffi-closure` which is suitable as an argument for the type denoted by the `closure` type specifier keyword in the FFI type language.

This type appears a C function pointer in the foreign code, and may be called as such. When it is called by foreign code, it triggers a call to *`lisp-fun`*.

The optional *`safe-p`* parameter controls whether the closure dispatch is "safe", the meaning of which is described shortly. The default value is `t` so that unsafe closure dispatch must be explicitly requested with a `nil` argument for this parameter.

A callback closure which is safely dispatched, firstly, does not permit the capture of delimited continuations across foreign code. Delimited continuations can be captured inside a closure dispatched that way, but the delimiting prompt must be within the callback's local stack frame, without traversing across the foreign stack frames. Secondly, a callback closure which is safely dispatched doesn't permit direct nonlocal control transfers across foreign code, such as exception handling. Such transfers, however, appear to work anyway (with caveats): this is because they are specially handled. The closure dispatch mechanism intercepts all dynamic control transfers, converts them to an ordinary return from the callback to the foreign code, and resumes the control transfer when the foreign code itself finishes and returns. If the callback returns a value (its return type is other than `void`) then in this situation, the callback returns an all-zero-bits return value to the foreign caller. If the *`abort-val`* parameter is specified and its value is other than `nil`, then that value will be used as the return value instead of an all-zero bit pattern.

An unsafely dispatched closure permits the capture of continuations from the callback across the foreign code and direct dynamic control transfers which abandon the foreign stack frames.

Unsafe closure dispatch is only compatible with foreign code which is designed with that usage in mind. For instance foreign code which holds dynamic resources in stack variables will leak those resources if abandoned this way. There are also issues with capturing continuations across foreign code.

Note: the C function pointer is called a "closure" because it carries environment information. For instance, if `lisp-fun` is a lexical closure, invocations of it through the FFI closure occur in its proper lexical environment, even though its external representation is a simple C function pointer. This requires a special trampoline trick: a piece of dynamically constructed machine code with the closure binding embedded inside it, with the C function pointer pointing to the machine code.

Note: the same call descriptor can be reused multiple times to create different closures. The same Lisp function can be involved in multiple FFI closures.

Example:

```
;; Package the TXR cmp-str function as a string
;; comparison callback compatible with:
;;
;;   int (*)(const char *, const char *)
;;
(ffi-make-closure
  (fun cmp-str)
  (ffi-make-call-desc 2 nil ;; two args, non-variadic
    (ffi-type-compile 'int) ;; int return
    [mapcar ffi-type-compile '(str str)])) ;; args
```

10.13.6 Function `ffi-call`

Syntax:

```
(ffi-call fun-cptr call-desc {arg}*)
```

Description:

The `ffi-call` function invokes a foreign function.

The *`fun-cptr`* argument which must be a `cptr` object. It is assumed to point to a foreign function.

The *`call-desc`* argument must be a FFI call descriptor, produced by `ffi-make-call-desc`.

The *`call-desc`* must correctly describe the foreign function.

The zero or more *`arg`* arguments are values which are converted into foreign argument values. There must be exactly as many of these arguments as are required by *`call-desc`*.

The `ffi-call` function converts every *`arg`* to a corresponding foreign object. If these conversions are successful, the converted foreign arguments are passed by value to the foreign function indicated by *`fun-cptr`*. An unsuccessful conversion throws an error.

When the call returns, the foreign function's return value is converted to a Lisp object and returned, in accordance with the return type that is declared inside *`call-desc`*.

10.13.7 Function `ffi-typedef`

Syntax:

```
(ffi-typedef name type)
```

Description:

The `ffi-typedef` function installs the compiled FFI type given by *`type`* as a typedef name under the symbol given by *`name`*.

After this registration, whenever the type compiler encounters that symbol being used as a type specifier, it will replace it by the type object it represents.

The `ffi-typedef` function returns *`type`*.

Example:

```
;; define refcount-t as an alias for uint32
(ffi-typedef 'refcount-t (ffi-type-compile 'uint32))
```

10.13.8 Function `ffi-size`

Syntax:

```
(ffi-size type)
```

Description:

The `ffi-size` function returns an integer which gives the storage size of the given FFI type: the amount of storage required for the external representation of that type.

Bitfield types do not have a size; it is an error to apply this function to a bitfield.

The size is machine:specific.

Example:

```
(ffi-size '(ffi-type-compile 'double)) -> 8
(ffi-size '(ffi-type-compile 'char)) -> 1
(ffi-size '(ffi-type-compile
             '(array 42 char))) -> 42
```

10.13.9 Function `ffi-alignof`

Syntax:

```
(ffi-alignof type)
```

Description:

The `ffi-alignof` function returns an integer which gives the alignment the given FFI type. When an instance of *`type`* is placed into a structure as a member, it is placed after the previous member at the smallest available offset which is divisible by the alignment. The bytes skipped from the smallest available offset to the smallest available aligned offset are referred to as *padding*.

Bitfield types do not have an alignment; it is an error to apply this function to a bitfield. Bitfields are allocated in storage cells, and those cells have alignment which is the same as that of the type `int`.

The alignment is machine-specific. It may be more strict than what the hardware architecture requires, yet at the same time be smaller than the size of the type. For instance, the size of the type `double` is commonly 8, yet the alignment is often 4, and this is so even on processors like Intel x86 which can load and store a double at a misaligned address.

The alignment of an array is the same as that of its element type.

The alignment of a structure is that of its member which has the most strict (largest-valued) alignment.

It is a property of arrays, derived from requirements governing the C language, that if the first element of an array is at a correctly aligned address, then all elements are. To ensure that this property holds for for arrays of structures, structures sometimes must include padding at the end. This is because the size of a structure without any padding might not be multiple of its alignment, which is derived from the most strictly aligned member. For instance, if we assume an architecture on which the size and alignment of `int` is 4, the size of the structure type `(struct ab (a int) (b char))` would be 5 if no padding were included. However, in an array of these structures, the second element's `a` member would be placed at offset 5, rendering it misaligned. To ensure that every `a` is placed at an offset which is multiple of 4, the struct type is extended with anonymous padding so that its size is 8.

Example:

```
(ffi-alignof (ffi double)) -> 4
```

10.13.10 Function `ffi-offsetof`

Syntax:

```
(ffi-offsetof type member)
```

Description:

The `ffi-alignof` function calculates the byte offset of *`member`* within the FFI type *`type`*.

If *`type`* isn't a FFI struct type, or if *`member`* isn't a symbol naming a member of that type, the function throws an exception.

An exception is also thrown if *`member`* is a bitfield.

Example:

```
(ffi-offsetof (ffi (struct ab (a int) (b char))) 'b) -> 4
```

10.13.11 Function `ffi-arraysize`

Syntax:

```
(ffi-arraysize type)
```

Description:

The `ffi-arraysize` function reports the number of elements in *`type`*, which must be an array type: an `array`, `zarray` or `carray`.

Example:

```
(ffi-arraysize (ffi (array 5 int))) -> 5
```

10.13.12 Function `ffi-elemsize`

Syntax:

```
(ffi-elemsize type)
```

Description:

The `ffi-elemsize` function reports the size of the element type of an array, of the target type of a pointer, or of the base integer type of an enumeration. The *type* argument must be an array, pointer or enumeration type: a type constructed by one of the operators `array`, `zarray`, `carray`, `ptr`, `ptr-in`, `ptr-out`, `enum` or `enumed`.

Example:

```
(ffi-elemsize (ffi (array 5 int))) -> 4 ;; (sizeof int)
```

10.13.13 Function `ffi-elemtype`

Syntax:

```
(ffi-elemtype type)
```

Description:

The `ffi-elemtype` function retrieves the element type of an array type, target type of a pointer type, or base integer type of an enumeration. The *type* argument must be an array, pointer or enumeration type: a type constructed by one of the operators `array`, `zarray`, `carray`, `ptr`, `ptr-in`, `ptr-out`, `enum` or `enumed`.

Example:

```
(ffi-elemtype (ffi (ptr int))) -> #<ffi-type int>
```

10.14 Foreign Function Macro Language

This group of macros provides a higher-level language for working with FFI types and defining foreign function bindings. The macros are implemented using the Foreign Function Type API described in the previous section.

10.14.1 Macro `with-dyn-lib`

Syntax:

```
(with-dyn-lib lib-expr body-form*)
```

Description:

The `with-dyn-lib` macro works in conjunction with the `deffi`, `deffi-sym` and `deffi-var` macros.

When a `deffi` form appears as one of the *body-form*s of the `with-dyn-lib` macro, that `deffi` form is permitted to use the simplified forms of the *fun-expr* argument, to refer to library functions succinctly, without having to specify the library. The same remark applies to `deffi-sym` and `deffi-var`, regarding their *var-expr* parameter.

A form invoking the `with-dyn-lib` macro should be a top-level form. The macro creates a global variable named by a symbol generated by `gensym` whose initializing expression binds it to a dynamic library handle. The macro then creates an environment in which the enclosed `deffi`, `deffi-var` and `deffi-sym` forms can implicitly refer to that library via the global variable.

The *lib-expr* argument can take on three different forms:

`nil` If *lib-expr* is `nil`, then `with-dyn-lib` arranges for the library to refer to the **TXR** executable itself.

string
If *lib-expr* is a literal string, then `with-dyn-lib` will arrange for the hidden variable to be initialized with an expression which opens a handle to the specified library.

form If *lib-expr* is any other form, then it is assumed to denote syntax for opening the handle to a library. That syntax is used verbatim as the initializing expression for the generated global variable which holds the library handle.

The result value of a `with-dyn-lib` form is the symbol which names the generated variable which holds the library handle.

Examples:

```
;; refer to malloc and free functions
;; in the executable

(with-dyn-lib nil
  (deffi malloc "malloc" cptr (size-t))
  (deffi free "free" void (cptr)))

;; refer to "draw" function in fictitious
;; "libgraphics" library:

(with-dyn-lib "libgraphics.so.5"
  (deffi draw "draw" int (cptr cptr)))

;; refer to "init_foo" function via specific
;; library handle.

(defvarl foo-lib (dlopen "libfoo.so.1"))

(with-dyn-lib foo-lib
  (deffi init-foo "init_foo" void (void)))
```

10.14.2 Macro `deffi`

Syntax:

```
(deffi name fun-expr rettype argtypes)
```

Description:

The `deffi` macro arranges for a Lisp function to be defined, via `defun`, which calls a foreign function.

The *name* argument must be a symbol suitable as a function name in a `defun` form. This specifies the function's Lisp name.

The *fun-expr* parameter specifies the foreign function which is to be called. The syntactic variants permitted for its argument are described below.

The *rettype* argument must specify the return type, using the FFI type syntax, as an unquoted

literal. The macro arranges for the compilation of this syntax via `ffi-type-compile`.

The *argtypes* argument must specify a list of the argument types, as an unquoted literal list, using FFI type syntax. The macro arranges for these types to be compiled. Furthermore, a special convention may be used for specifying a variadic function: if the **:** (colon) keyword symbol appears as one of the elements of *argtypes*, then the `deffi` form specifies a fixed call to a foreign function which is variadic. The argument types before the colon keyword are the types of the fixed arguments. The types after the colon, if any, are of the variadic arguments. Special considerations apply to some variadic argument types, described below.

The following syntactic variants are permitted of the *fun-expr* argument:

name-string
: If *fun-expr* is a literal string, then the `deffi` form must be enclosed in the `with-dyn-lib` macro, appearing as one of that macro's *body-form*s. In this situation the literal character string *name-string* specifies a symbol to be found within the library established by the *with-dyn-lib* macro.

(*name-string ver-string*)
: This manner of specifying the *fun-expr* also requires the `deffi` form to be enclosed in a `with-dyn-lib`. It selects a particular version of a symbol from the library.

form
: If *fun-expr* is any other form, then it must specify an expression which evaluates to a `cptr` object giving the address of a foreign library symbol. If this form is used, then the `deffi` form need not be surrounded by a call to the `with-dyn-lib` macro.

When the FFI type `float` is used as the type of a variadic parameter, `deffi` replaces it by the FFI type `double`. This treatment is necessary because the C variadic argument mechanism promotes `float` values to `double`. Note: due to this substitution, it is possible to pass floating-point values which are out of range of the `float` type, without any diagnosis. The behavior of is undefined in the Lisp-to-C direction, if the C function extracts an out-of-range `double` argument as if it were of type `float`.

The FFI types `be-float` and `le-float` cannot be used for specifying the types of a variadic argument. If any of these occur in that position, `deffi` throws an error. Rationale: these types are related to the C type `float` type, which requires promotion in variadic passing. Promotion cannot be performed on floating-point values whose byte order has been rearranged, because promotion is a value-preserving conversion.

The result value of a `deffi` form is *name*.

10.14.3 Macros `deffi-cb` and `deffi-cb-unsafe`

Syntax:

```
(deffi-cb name rettype argtypes [abort-val])
(deffi-cb-unsafe name rettype argtypes)
```

Description:

The `deffi-cb` macro defines, using `defun` a Lisp function called *name*.

Thus the *name* argument must be a symbol suitable as a function name in a `defun` form.

The *rettype* and *argtypes* arguments are processed exactly as in the corresponding arguments in the `deffi` macro.

The `deffi-cb` macro arranges for *rettype* and *argtypes* to be compiled into a FFI call descriptor. The generated function called *name* then serves as a combinator which takes a Lisp function as its argument, and binds it to the FFI call descriptor to produce a FFI closure. That closure may then be passed to foreign functions as a callback. The `deffi-cb` macro generates a callback which uses safe dispatch, which is explained in the description of the `ffi-make-closure` function. The optional *abort-val* parameter specifies an expression which evaluates to the value to be returned by the callback in the event that a dynamic control transfer is intercepted. The purpose of this value is to indicate to the foreign code that the callback wishes to abort operation; it is useful in situations when a suitable return value will induce the foreign code to cooperate and itself return to the Lisp code which will then continue the dynamic control transfer.

The `deffi-cb-unsafe` macro is a variant of `deffi-cb` with the same argument conventions. The difference is that it arranges for `ffi-make-closure` to be invoked with `nil` for the *safe-p* parameter. This macro has no *abort-val* parameter, since unsafe callbacks do not use it.

Example:

```
;; create a closure combinator which binds
;; Lisp functions to a call descriptor has the C type
;; signature void (*)(int).

(deffi-cb void-int-closure void (int))

;; use the combinator
;; some-foreign-function's second arg is
;; of type closure, specifying a callback:

(some-foreign-function
  42
  (void-int-closure (lambda (x)
                      (puts `callback! @x`))))
```

10.14.4 Macro `deffi-var`

Syntax:

```
(deffi-var name var-expr type)
```

Description:

The `deffi-var` macro defines a global symbol macro which expands to an expression accessing a foreign variable, creating the illusion that the variable is available as a Lisp variable holding a Lisp data type.

The *name* argument gives the name of the symbol macro to be defined.

The *var-expr* argument is one of several permitted syntactic forms which specify the address of the foreign variable. They are described below.

The *type* argument expresses the variable type in FFI type syntax.

Once the variable is defined, accessing the macro symbol *name* performs a get operation on the foreign variable, yielding the conversion of that variable to a Lisp value. An assignment to the symbol performs a put operation, converting a Lisp object to a value which overwrites the object.

Note: FFI memory management is not helpful in the use of variables. Suppose a string value is stored in a variable of type `str`. This means that FFI dynamically allocates a buffer which stores the UTF-8 encoded version of the string, and this buffer is placed into the foreign variable. Then suppose another such assignment takes place. The previous value is simply overwritten without being freed.

The following syntactic variants are permitted of the *var-expr* argument:

name-string
: If *var-expr* is a literal string, then the `deffi-var` form must be enclosed in the `with-dyn-lib` macro, appearing as one of that macro's *body-form*s. In this situation the literal character string *name-string* specifies a symbol to be found within the library established by the *with-dyn-lib* macro.

(*name-string ver-string*)
: This manner of specifying the *fun-expr* also requires the `deffi` form to be enclosed in a `with-dyn-lib`. It selects a particular version of a symbol from the library.

form
: If *var-expr* is any other form, then it must specify an expression which evaluates to a `cptr` object giving the address of a foreign library symbol. If this form is used, then the `deffi` form need not be surrounded by a call to the `with-dyn-lib` macro.

10.14.5 Macro `deffi-sym`

Syntax:

```
(deffi-sym name var-expr [type-sym])
```

Description:

The `deffi-sym` macro defines a global lexical variable called `name` whose value is a `cptr` object that refers to a symbol in a foreign library.

The *name* argument gives the name for the variable to be defined. This definition takes place place as if by the `defparml` macro.

The *var-expr* is syntax which specifies the foreign pointer, using exactly the same conventions as described for the `deffi-var` macro, allowing for a shorthand notation if this form is enclosed in a `with-dyn-lib` macro invocation.

The optional *type-sym* argument must be a symbol. If it is absent, it defaults to nil. This argument specifies the type label for the `cptr` object which holds the pointer to the foreign symbol.

The result value of *deffi-sym* is the symbol *name*.

10.14.6 Macro `typedef`

Syntax:

```
(typedef name type-syntax)
```

Description:

The `typedef` macro provides a convenient way to define type aliases.

The *type-syntax* expression is compiled as FFI syntax, and the *name* symbol is installed as an alias denoting that type.

The `typedef` macro yields the compiled version of *type-syntax* as its value.

10.14.7 Macros `deffi-struct` and `deffi-union`

Syntax:

```
(deffi-struct name {(slot type [init-form])}*)
(deffi-union name {(slot type [init-form])}*)
```

Description:

The `deffi-struct` and `deffi-union` macros provide a more compact notation for defining FFI structure and union types together with matching `typedef` names.

The semantics follows from these equivalences:

```
(deffi-struct S ...)  <-->  (typedef S (struct S ...))
(deffi-union U ...)  <-->  (typedef U (union U ...))
```

Example:

```
(deffi-struct point
  (x double)
  (y double))
```

10.14.8 Macro `sizeof`

Syntax:

```
(sizeof type-syntax [object-expr])
```

Description:

The macro `sizeof` calculates the size of the FFI type denoted by `type-syntax`.

The *`type-syntax`* expression is compiled to a type using `ffi-type-compile`. The *`object-expr`* expression is evaluated to an object value.

If `type-syntax` denotes an incomplete array or structure type, and the *`object-expr`* argument is present, then a *dynamic size* is computed: the actual number of bytes required to store that object value as a foreign representation.

The `sizeof` macro arranges for the size calculation to be carried out at macro-expansion time, if possible, so that the `sizeof` form is replaced by an integer constant. This is possible when the *`object-expr`* is omitted, or if it is a constant expression according to the `constantp` function.

For the type `void`, incomplete array types, and bitfield types, the one-argument form of `sizeof` reports zero.

For incomplete structure types, the one-argument `sizeof` reports a size which is equivalent to the offset of the last member. The size of an incomplete structure does not include padding for the most strictly aligned member.

10.14.9 Macro `alignof`

Syntax:

```
(alignof type-syntax)
```

Description:

The macro `alignof` calculates the alignment of the FFI type denoted by `type-syntax` at macro-expansion time, and produces that integer value as its expansion, such that there is no run-time computation. It uses the `ffi-alignof` function.

10.14.10 Macro `offsetof`

Syntax:

(`offsetof` *`type-syntax member-name`*)

Description:

The macro `sizeof` calculates the offset of the structure member indicated by *`member-name`*, a symbol, inside the FFI struct type indicated by *`type-syntax`*. This calculation is performed by a macro-expansion-time call to the `ffi-offsetof` function, and produces that integer value as its expansion, such that there is no run-time computation.

10.14.11 Macro `arraysize`

Syntax:

(`arraysize` *`type-syntax`*)

Description:

The macro `arraysize` calculates the number of elements of the array type indicated by *`type-syntax`*. This calculation is performed by a macro-expansion-time call to the `ffi-arraysize` function, and produces that integer value as its expansion, such that there is no run-time computation.

10.14.12 Macro `elemsize`

Syntax:

(`elemsize` *`type-syntax`*)

Description:

The macro `elemsize` calculates the size of the element type of an array type, or the size of target type of a pointer type indicated by *`type-syntax`*. This calculation is performed by a macro-expansion-time call to the `ffi-elemsize` function, and produces that integer value as its expansion, such that there is no run-time computation.

10.14.13 Macro `elemtype`

Syntax:

(`elemtype` *`type-syntax`*)

Description:

The macro `elemtype` produce the element type of an array type, or the target type of a pointer type indicated by *`type-syntax`*. Note: the `elemtype` macro may be understood in terms of several possible implementations. The form `(elemtype X)` is equivalent to `(ffi-elemtype (ffi-type-compile X))`. Since there exists an `elemtype` type operator, the expression is also equivalent to `(ffi-type-compile '(elemtype X))`.

10.14.14 Macro `ffi`

Syntax:

```
(ffi type-syntax)
```

Description:

The `ffi` macro provides a shorthand notation for compiling a literal FFI type expression to the corresponding type object. The following equivalence holds:

```
(ffi expr)  <-->  (load-time (ffi-type-compile 'expr))
```

10.15 Zero-filled Object Support

Communicating with foreign interfaces sometimes requires representations to be initialized consisting of all zero bits, or mostly zero bits.

TXR provides convenient ways to prepare Lisp objects such that when those objects are converted to a foreign representation, they generate zero-filled representations.

10.15.1 Function `make-zstruct`

Syntax:

```
(make-zstruct type {slot-sym init-value}*)
```

Description:

The `make-zstruct` function provides a convenient means of instantiating a structure for use in foreign function calls, imitating a pattern of initialization often seen in the C language. It instantiates a Lisp `struct` by conversion of zero-filled memory through FFI, thus creating a Lisp structure which appears zero-filled when converted to the foreign representation.

This simplifies application code, which is spared from providing individual slot initializations which have this effect.

The *`type`* argument must be a compiled FFI `struct` type. The remaining arguments must occur pairwise. Each *`slot-sym`* argument must be a symbol naming a slot in the FFI `struct` type. The *`init-value`* argument which follows it specifies the value for that slot.

The `make-zstruct` function operates as follows. Firstly, the Lisp `struct` type is retrieved which corresponds to the FFI type given by *`type`*. A new instance of the Lisp type is instantiated, as if by a one-argument call to `make-struct`. Next, each slot indicated by a *`slot-sym`* argument is set to the corresponding *`init-value`*. Finally, each slot of the struct which is not initialized via *`slot-sym`* and *`init-value`* pair, and which is known to the FFI type, is reinitialized by a conversion from a foreign object of all-zero bits to a Lisp value. argument. The `struct` object is then returned.

Note: the `znew` macro provides a less verbose notation based on `make-zstruct`.

Note: slots which are not known to the FFI `struct` type may be initialized by `make-zstruct`. Each *`slot-sym`* must be a slot of the Lisp `struct` type; but need not be declared as a member in the FFI `struct` type.

10.15.2 Macro `znew`

Syntax:

```
(znew type-syntax {slot-sym init-value}*)
```

Description:

The `znew` macro provides a convenient way of using `make-zstruct`, using syntax which resembles that of the `new` macro.

The `znew` macro generates a `make-zstruct` call, arranging for the *`type-syntax`* argument to be compiled to a FFI type object, and applies quoting to every *`slot-sym`* argument.

The following equivalence holds:

```
(znew s a i b j ...)  <-->  (make-zstruct (ffi s)
                                          'a i 'b j ...)
```

Example

Given the following FFI type definition

```
(typedef foo (struct foo (a (cptr bar)) (b uint) (c bool)))
```

the following results are observed:

```
;; ordinary instantiation
(new foo) -> #S(foo a nil b nil c nil)

;; Under znew, a is null cptr of correct type:
(znew foo) -> #S(foo a #<cptr bar: 0> b 0 c nil)

;; value of b is specified; others come from zeros:
(znew foo b 42) -> #S(foo a #<cptr bar: 0> b 42 c nil)
```

10.15.3 Function `zero-fill`

Syntax:

```
(zero-fill type obj)
```

Description:

The `zero-fill` function invokes the by-reference in semantics of FFI type *`type`* against a zero-filled buffer, and a Lisp object *`obj`*.

This means that if *`obj`* is an aggregate such as a vector, list or structure, it is updated as if from an all-zero-bit foreign representation. In that situation, *`obj`* is also returned.

An object which has by-value semantics, such as an integer, is not updated. In this case, nevertheless, the return value is a Lisp object produced by converting an all-zero-bit buffer to *`type`*.

10.16 Foreign Unions

The following group of functions provides the means for working with foreign unions, in conjunction with the `union` FFI type.

10.16.1 Function `make-union`

Syntax:

```
(make-union type [initval [member]])
```

Description:

The `make-union` function instantiates a new object of type `union`, based on the FFI type specified by the *`type`* parameter, which must be compiled FFI `union` type.

The object provides storage for the foreign representation of `type`, and that storage is initialized to all zero bytes.

Additionally, if *`initval`* is specified, but *`member`* is not, then *`initval`* is stored into the union's via the first member, as if by `union-put`. If the union type has no members, an error exception is thrown.

If both *`initval`* and *`member`* are specified, then *`initval`* is stored into the union using the specified member, as if by `union-put`.

10.16.2 Function `union-members`

Syntax:

```
(union-members union)
```

Description:

The `union-members` function retrieves the list of symbols which name the members of *`union`*. These are derived from the object's FFI type. It is unspecified whether the list is freshly allocated on each call, or whether the same list is returned; applications shouldn't destructively manipulate this list.

10.16.3 Function `union-get`

Syntax:

```
(union-get union member)
```

Description:

The `union-get` function performs the get semantics (conversion from a foreign representation to Lisp) on the member of *`union`* which is specified by the *`member`* argument. That argument must be a symbol corresponding to one of the member names.

The *`union`* object's storage buffer is treated as an object of the foreign type indicated by that member's type information, and converted accordingly to a Lisp object that is returned.

10.16.4 Function `union-put`

Syntax:

```
(union-put union member new-value)
```

Description:

The `union-put` function performs the put semantics (conversion from a Lisp object to foreign representation) on the member of *`union`* which is specified by the *`member`* argument. That argument must be a symbol corresponding to one of the member names.

The object given as *`new-value`* is converted to the foreign representation according to the type

information of the indicated member, and that representation is placed into the *union* object's storage buffer.

The return value is *new-value*.

10.16.5 Functions `union-in` and `union-out`

Syntax:

```
(union-in union memb memb-obj)
(union-out union memb memb-obj)
```

Description:

The `union-in` and `union-out` functions perform the FFI in semantics and out semantics, respectively. These semantics are involved in two-way data transfers between foreign representations and Lisp objects.

The *union* argument must be a `union` object and the *memb* argument a symbol which matches one of that object's member names.

In the case of `union-in`, *memb-obj* is a Lisp object that was previously stored into *union* using the `union-put` operation, into the same member that is currently indicated by *member*.

In the case of `union-out`, *memb-obj* is a Lisp object that was previously retrieved from *union* using the `union-get` operation, from the same member that is currently indicated by *member*.

The `union-in` performs the by-value nuance of the in semantics on the indicated member: if the member contains pointers to any objects, those objects are updated from their counterparts in *memb-obj* using their respective by-reference in semantics, recursively.

Similarly `union-out` performs the by-value nuance of the out semantics on the indicated member: if the member contains pointers to any objects, those objects are updated with their Lisp counterparts in *memb-obj* using their respective by-reference out semantics, recursively.

Note: `union-in` is intended to be used after a FFI call, on a union-typed by-value argument, or a union-typed object contained in an argument, in situations when the function is expected to have updated the contents of the union. The `union-out` function is intended to be used in a FFI callback, on a union-typed callback argument or union-typed object contained in such an argument, in cases when the callback has updated the Lisp object corresponding to a union member, and that change needs to be propagated to the foreign caller.

10.17 FFI-type-driven I/O Functions

These functions provide a way to perform I/O on stream using the foreign representation of Lisp objects, performing conversion between the Lisp representations in memory and the foreign representations in a stream.

The *stream* argument used with these functions must be a stream object which, in the case of input functions, supports `get-byte` and, in the case of output, supports `put-byte`.

10.17.1 Function `put-obj`

Syntax:

```
(put-obj object type [stream])
```

Description:

The `put-obj` function encodes *`object`* into a foreign representation, according to the FFI type *`type`*. The bytes of the foreign representation are then written to *`stream`*.

If *`stream`* is omitted, it defaults to `*stdout*`.

If the operation successfully writes all bytes of the representation to *`stream`*, the value `t` is returned. A partial write causes the return value to be `nil`.

All other stream error situations throw exceptions.

10.17.2 Function `get-obj`

Syntax:

```
(get-obj type [stream])
```

Description:

The `get-obj` function reads from *`stream`* the bytes corresponding to a foreign representation according to the FFI type *`type`*.

If *`stream`* is omitted, it defaults to `*stdin*`.

If the read is successful, these bytes are decoded, producing a Lisp object, which is returned.

If the read is incomplete, the value returned is *`nil`*.

All other stream error situations throw exceptions.

10.17.3 Function `fill-obj`

Syntax:

```
(fill-obj object type [stream])
```

Description:

The `fill-obj` function reads from *`stream`* the bytes corresponding to a foreign representation according to the FFI type *`type`*.

If the read is successful, then *`object`* is updated, if possible, from that representation, using the by-value in semantics of the FFI type and returned. If a by-value update of *`object`* isn't possible, then a new object is decoded from the data and returned.

If the read is incomplete, the value returned is *`nil`*.

All other stream error situations throw exceptions.

10.18 Buffer Functions

Functions in this area provide a way to perform conversion between Lisp objects and foreign representation to and from objects of the `buf` type.

10.18.1 Functions `ffi-put` and `ffi-put-into`

Syntax:

```
(ffi-put obj type)
(ffi-put-into dst-buf obj type [offset])
```

Description:

The `ffi-put` function encodes the Lisp object *obj* according to the FFI type *type* and returns a new buffer object of type `buf` which holds the foreign representation.

The `ffi-put-into` function is similar, except that it uses an existing buffer *dst-buf* which must be large enough to hold the foreign representation.

The *type* argument must be a compiled FFI type.

If *type* is has a variable length, then the actual size of the foreign representation is calculated from *obj*.

The *obj* argument must be an object compatible with the conversions implied by *type*.

The optional *offset* argument specifies a byte offset from the beginning of the data area of *dst-buf* where the foreign representation of *obj* is stored. The default value is zero.

These functions perform the "put semantics" encoding action similar to what happens to the arguments of an outgoing foreign function call.

Caution: incorrect use of this function, or its use in isolation without a matching `ffi-in` call, can cause memory leaks, because, depending on *type*, temporary resources may be allocated, and pointers to those resources will be stored in the buffer.

10.18.2 Function `ffi-out`

Syntax:

```
(ffi-out dst-buf obj type copy-p [offset])
```

Description:

The `ffi-out` function performs the "out semantics" encoding action, similar to the treatment applied to the arguments of a callback prior to returning to foreign code.

It is assumed that `obj` is an object that was returned by an earlier call to `ffi-get`, and that the *dst-buf* and *type* arguments are the same objects that were used in that call.

The *copy-p* argument is a Boolean flag which is true if the buffer represents a datum that is being passed by pointer. If *copy-p* is true, then *obj* is converted to a foreign representation which is stored into *dst-buf*. If it is false, it indicates that the buffer itself is a pass-by-value object. This means that the object itself will not be copied, but if it is an aggregate which contains pointers, the operation will recurse on those objects, invoking their "out semantics" action with pass-by-pointer semantics. The required pointers to these indirect objects are obtained from *dst-buf*.

The optional *offset* argument specifies a byte offset from the beginning of the data area of *dst-buf* where the foreign representation of *obj* is understood to be stored, and where it is updated if requested by *copy-p*. The default value is zero.

The `ffi-out` function returns *dst-buf*.

10.18.3 Function `ffi-in`

Syntax:

```
(ffi-in src-buf obj type copy-p [offset])
```

Description:

The `ffi-in` function performs the "in semantics" decoding action, similar to the treatment applied to the arguments of a foreign function call after it returns, in order to free temporary resources and recover the new values of objects that have been modified by the foreign function.

It is assumed that *`src-buf`* is a buffer that was prepared by a call to `ffi-put` or `ffi-put-into`, and that *`type`* and *`obj`* are the same values that were passed as the corresponding arguments of those functions.

The `ffi-in` function releases the temporary memory resources that were allocated by `ffi-put` or `ffi-put-into`, which are obtained from the buffer itself, where they appear as pointers. The function recursively performs the in semantics across the entire type, and the entire object graph rooted at the buffer.

The *`copy-p`* argument is a Boolean flag which is true if the buffer represents a datum that is being passed by pointer. If it is false, it indicates that the buffer itself is a pass-by-value object. Under pass-by-pointer semantics, either a whole new object is extracted from the buffer and returned, or else the slots of *`obj`* are updated with new values from the buffer. Under pass-by-value semantics, no such extraction takes place, and *`obj`* is returned. However, regardless of the value of `copy-p`, if the object is an aggregate which contains pointers, the recursive treatment through those pointers involves pass-by-pointer semantics.

This is consistent with the idea that we can pass a structure by value, but that structure can have pointers to objects which are updated by the called function. Those indirect objects are passed by pointer. They get updated, but the parent structure cannot.

If *`type`* is has a variable length, then the actual size of the foreign representation is calculated from *`obj`*.

The optional *`offset`* argument specifies a byte offset from the beginning of the data area of *`src-buf`* from which the foreign representation of *`obj`* is taken.

The `ffi-in` function returns either *`obj`* or a new object which is understood to have been produced as its replacement.

10.18.4 Function `ffi-get`

Syntax:

```
(ffi-get src-buf type [offset])
```

Description:

The `ffi-get` function extracts a Lisp value from buffer *`src-buf`* according to the FFI type *`type`*. The *`src-buf`* argument is an object of type *`buf`* large enough to hold a foreign representation of *`type`*, at the byte offset indicated by the *`offset`* argument. The *`type`* argument is compiled FFI type. The optional *`offset`* argument defaults to zero.

The external representation in *`src-buf`* at the specified offset is scanned according to *`type`* and converted to a Lisp value which is returned.

The `ffi-get` operation is similar to the "get semantics" performed by FFI in order to extract the return value of foreign function calls, and by the FFI callback mechanism to extract the arguments coming into a callback.

The *type* argument may not be a variable length type, such as an array of unspecified size.

10.19 Foreign Arrays

Functions in this area provide a means for working with foreign arrays, in connection with the FFI `carray` type.

10.19.1 Functions `carray-vec` and `carray-list`

Syntax:

```
(carray-vec vec type [null-term-p])
(carray-list list type [null-term-p])
```

Description:

The `carray-vec` and `carray-list` functions allocate storage for the representation of a foreign array, and return a `carray` object which holds a pointer to that storage.

The argument *type*, which must be a compiled FFI type, is retained as the `carray` object's element type.

Prior to returning, the functions initializes the foreign array by converting the elements of *vec* or, respectively, *list* into elements of the foreign array. The conversion is performed using the put semantics of *type*, which is a compiled FFI type.

The length of the returned `carray` is determined from the length of *vec* or *list* and from the value of the Boolean argument *null-term-p*.

If *null-term-p* is `nil`, then the length of the `carray` is the same as that of the input *vec* or *list*.

A true value of *null-term-p* indicates null termination. This causes the length of the `carray` to be one greater than that of *vec* or *list*, and the extra element allocated to the foreign array is filled with zero bytes.

10.19.2 Function `carrayp`

Syntax:

```
(carrayp object)
```

Description:

The `carrayp` function returns `t` if *object* is a `carray`, otherwise it returns `nil`.

10.19.3 Function `carray-blank`

Syntax:

```
(carray-blank length type)
```

Description:

The `carray-blank` function allocates storage for the representation of a foreign array, filling that storage with zero bytes, and returns a `carray` object which holds a pointer to that storage.

The argument *type*, which must be a compiled FFI type, is retained as the `carray` object's element type.

The *length* argument must be a nonnegative integer; it specifies the number of elements in the foreign array and is retained as the `carray` object's length.

The size of the foreign array is the product of the size of *type* as reported by the `ffi-size` function, and of *length*.

10.19.4 Function `carray-buf`

Syntax:

```
(carray-buf buf type [offset])
```

Description:

The `carray-buf` function creates a `carray` object which refers to the storage provided and managed by the buffer object *buf*, providing a view of that storage, and manipulation thereof, as an array.

The optional *offset* parameter specifies an offset from the start of the buffer to the location which is interpreted as the start of the `carray`, which extends from that offset to the end of the buffer.

The default value is zero: the `carray` covers the entire buffer.

If a value is specified, it must be in the range zero to the length of *buf*.

The *type* argument must be a compiled FFI type whose size is nonzero.

The `carray` is overlaid onto the storage of *buf* as follows:

First, *offset* is subtracted from the bytewise length of *buf*, as reported by `length-buf` function to produce the effective length of the storage to be used for the array.

The effective length is divided by the size of *type*, as reported by `ffi-size`. The resulting quotient represents the length (number of elements) of the `carray` object.

Note: the returned `carray` object holds a reference to *buf*, preventing *buf* from being reclaimed by garbage collection, thereby protecting the underlying storage from becoming invalid. A subsequent invocation of `carray-own` operation releases this reference.

Note: the relationship between the `carray` object and *buf* is inherently unsafe: if *buf* is subsequently subject to operations which reallocate the storage, such as `buf-set-length` the pointer stored inside the referencing `carray` object becomes invalid, and operations involving that pointer have undefined behavior.

Note: if the length of the buffer is not evenly divisible by the size of the type, the calculated number of elements is rounded down. The trailing portion of the buffer corresponding to the division remainder, being insufficient to constitute a whole array element, is excluded from the array view.

10.19.5 Function `carray-buf-sync`

Syntax:

```
(carray-buf-sync carray)
```

Description:

The `carray-buf-sync` function requires *carray* to be a `carray` object which refers to a `buf` object for its storage. Such objects are created by the function `carray-buf`.

The `carray-buf-sync` function retrieves and returns the buffer object associated with *carray* and at the same time also updates the internal properties of *carray* using the current information: the pointer to the data, and the length of *carray* are altered to reflect the current state of the buffer.

10.19.6 Function `buf-carray`

Syntax:

```
(buf-carray carray)
```

Description:

The `buf-carray` function duplicates the underlying storage of *carray* and returns that storage represented as an object of `buf` type.

The storage size is calculated by multiplying the `carray` object's element size by the number of elements. Only that extent of the storage is duplicated.

10.19.7 Function `carray-cptr`

Syntax:

```
(carray-cptr cptr type [length])
```

Description:

The `carray-cptr` function creates a `carray` object based on a pointer derived from a `cptr` object.

The *cptr* argument must be of type `cptr`. The object's `cptr` type tag is ignored.

The *type* argument must specify a compiled FFI type, which will become the element type of the returned `carray`.

If *length* is specified as `nil`, or not specified, then the returned `carray` object will be of unknown length. Otherwise, *length* must be a nonnegative integer which will be taken as the length of the array.

Note: this conversion is inherently unsafe.

10.19.8 Function `cptr-carray`

Syntax:

```
(cptr-carray carray [type-symbol])
```

Description:

The `cptr-carray` function returns a `cptr` object which holds a pointer to a `carray` object's storage area. The *carray* argument must be of type `carray`.

The *type-symbol* argument should be a symbol. If omitted, it defaults to `nil`. This symbol becomes the `cptr` object's type tag.

The lifetime of the returned `cptr` object is independent from that of *`carray`*. If the lifetime of *`carray`* reaches its end before that of the `cptr`, the pointer stored inside the `cptr` becomes invalid.

10.19.9 Function `length-carray`

Syntax:

```
(length-carray carray)
```

Description:

The `length-carry` function returns the length of the *`carray`* argument, which must be an object of type `carray`.

If *`carray`* has an unknown length, then `nil` is returned.

10.19.10 Function `copy-carray`

Syntax:

```
(copy-carray carray)
```

Description:

The `copy-carray` function returns a duplicate of *`carray`*.

The duplicate has the same element type and length, but has its own copy of the underlying storage. This is true whether or not *`carray`* owns its storage or not. In either case, the duplicate owns *its* copy of the storage.

10.19.11 Function `carray-set-length`

Syntax:

```
(carray-set-length carray length)
```

Description:

The `carry-set-length` attempts to change the length of *`carray`*, which must be an object of `carray` type.

The *`length`* argument indicates the new length, which must be a nonnegative integer.

The operation throws an `error` exception if *`length`* is negative.

An `error` exception is also thrown if *`carray`* is an object which owns the underlying storage. There is no provision in the `carray` type to change the storage size.

It is permissible to change the length of a `carray` object which acts as a view into a buffer (as constructed via the `carray-buf` operation).

This creates a potentially unsafe situation in which the length requires a larger amount of backing storage than is provided by the buffer.

10.19.12 Accessor `carray-ref`

Syntax:

```
(carray-ref carray idx)
```

```
(set (carray-ref carray idx) new-val)
```

Description:

The `carray-ref` function accesses an element of the foreign array *carray*, converting that element to a Lisp value, which is returned.

The *idx* argument must be a nonnegative integer. If *carray* has a known length, *idx* must be less than the length.

If *carray* has an unknown length, then the access is permitted regardless of how positive is the value of *idx*. Whether the access has well-defined behavior depends on the actual extent of the underlying array storage.

The validity of any access to the underlying storage depends on the validity of the pointer to that storage.

The access to the array storage proceeds as follows. Every `carray` object has an element type, which is a compiled FFI type. A byte offset address is calculated by multiplying the size of the element type of *carray* by *idx*. Then, the get semantics of the element type is invoked to convert, to a Lisp object, a region of data starting at calculated byte offset in the array storage. The resulting object is returned.

Assigning an a value to a `carray-ref` form is equivalent to using `carray-refset` to store the value.

10.19.13 Function `carray-refset`

Syntax:

```
(carray-refset carray idx new-val)
```

Description:

The `carray-refset` function accesses an element of the foreign array *carray*, overwriting that element with a new value obtained from a conversion of the Lisp value *new-val*.

The return value is *new-val*.

The *idx* argument must be a nonnegative integer. If *carray* has a known length, *idx* must be less than the length.

If *carray* has an unknown length, then the access is permitted regardless of how positive is the value of *idx*. Whether the access has well-defined behavior depends on the actual extent of the underlying array storage.

The validity of any access to the underlying storage depends on the validity of the pointer to that storage.

The access to the array storage proceeds as follows. Every `carray` object has an element type, which is a compiled FFI type. A byte offset address is calculated by multiplying the size of the element type of *carray* by *idx*. Then, the put semantics of the element type is invoked to convert *new-val* to a foreign representation, which is written into the array storage started at the calculated byte offset.

If *new-val* has a type which is not compatible with the element type, or a value which is out of range or otherwise unsuitable, an exception is thrown.

10.19.14 Functions `carray-dup` and `carray-own`

Syntax:

```
(carray-dup carray)
(carray-own carray)
```

Description:

The `carray-dup` function acts upon a `carray` object which doesn't own its underlying array storage. It allocates a duplicate copy of the array storage referenced by *carray*, and assigns to *carray* the new copy. Then it marks *carray* as owning that storage. Lastly, if *carray* references another object, that reference is removed; *carray* no longer prevents the other object from being reclaimed by the garbage collector.

If *carray* already owns its storage, then this function has no effect.

If *carray* has an unknown size, then an error exception is thrown.

A `carray` produced by the functions `carray-vec` or `carray-blank` already owns its storage.

A `carray` object does not own its storage if it is produced by `carray-buf` or by the conversion of a foreign pointer under the control of the `carray` FFI type.

Because `carray` objects derived from foreign pointers via FFI have an unknown size, before using `carray-dup`, the application must determine the length of the array, and call `carray-set-length` to establish that length.

After `carray-dup`, the length may not be altered.

The `carray-dup` function returns `t` if it has performed the duplication operation. If it has done nothing, it returns `nil`.

The `carray-own` function resembles `carray-dup`, differing from that function only in two ways. Instead of allocating a duplicate copy of the underlying array storage, `carray-own` causes *carray* to **assume** ownership of the existing storage. Secondly, it is an error to use `carray-own` on a *carray* which references a buffer object.

The *carray-own* function always returns `nil`.

In all other regards, the descriptions of `carray-dup` apply to `carray-own`.

10.19.15 Function `carray-free`

Syntax:

```
(carray-free carray)
```

Description:

If *carray* is a `carray` object which owns the storage to which it refers, then `carray-free` function liberates that storage by passing the pointer to the C library function `free`. It then replaces that pointer with a null pointer, and changes the size to zero.

If *carray* doesn't own the storage, an exception is thrown.

10.19.16 Function `carray-type`

Syntax:

```
(carray-type carray)
```

Description:

The `carray-type` function returns the element type of *`carray`*, a compiled FFI type.

10.19.17 Functions `vec-carray` and `list-carray`

Syntax:

```
(vec-carray carray [null-term-p])
(list-carray carray [null-term-p])
```

Description:

The `vec-carray` and `list-carray` functions convert the array storage of *`carray`* to a freshly constructed object representation: vector, and list, respectively. The new vector or list is returned.

The *`carray`* object must have a known size; an `error` exception is thrown if these functions are invoked on a `carray` object of unknown size.

The effective length of the new vector or list is derived from the length of *`carray`*, taking into account the value of *`null-term-p`*.

The *`null-term-p`* Boolean parameter defaults to `nil`. If specified as true, then it has the effect that the effective length of the returned vector or list is one less than that of *`carray`*: in other words, a true value of *`null-term-p`* indicates that *`carray`* holds storage which represents a null-terminated array, and the terminating null element is to be excluded from the conversion.

If *`null-term-p`* is true, but the length of *`carray`* is already zero, then it has no effect; the effective length remains zero, and a zero-length vector or list is returned.

Conversion of the foreign array to the vector or list is performed by iterating over all of its elements, starting from element zero, up to the element before the effective length.

10.19.18 Functions `carray-get` and `carray-getz`

Syntax:

```
(carray-get carray)
(carray-getz carray)
```

Description:

The `carray-get` and `carray-getz` functions treat the contents of *`carray`* as a FFI `array` and `zarray` type, respectively.

They invoke the get semantics to convert the FFI array to a Lisp object, and return that object.

If the element type is one of `char`, `bchar` or `wchar`, then the expected string conversion semantics applies.

10.19.19 Functions `carray-put` and `carray-putz`

Syntax:

```
(carray-put carray new-val)
(carray-putz carray new-val)
```

Description:

The `carray-put` and `carray-putz` functions treat the contents of *carray* as a FFI `array` and `zarray` type, respectively.

They invoke the put semantics to convert the Lisp object *new-val* array to the foreign array representation, which is placed into the array storage referenced by *carray*.

If the element type is one of `char`, `bchar` or `wchar`, then the expected string conversion semantics applies.

Both of these functions return *carray*.

10.19.20 Accessor `carray-sub`

Syntax:

```
(carray-sub carray [from [to]])
(set (carray-sub carray [from [to]]) new-val)
```

Description:

The `carray-sub` function extracts a subrange of a *carray* object, returning a new `carray` object denoting that subrange.

The semantics of *from* and *to* work exactly like the corresponding arguments of the `sub` accessor, following the same conventions.

The returned `carray` shares the array has the same element type as the original and shares the same array storage. If, subsequently, elements of the original array are modified which lie in the range, then the modifications will affect the previously returned subrange `carray`. The returned `carray` references the original object, to ensure that as long as the returned object is reachable by the garbage collector, so is the original. This relationship can be severed by invoking `carray-dup` on the returned object, after which the two no longer share storage, and modifications in the original are not reflected in the subrange.

If `carray-sub` is used as a syntactic place, the argument expressions *carray*, *from*, *to* and *new-val* are evaluated just once. The prior value, if required, is accessed by calling `carray-sub` and *new-val* is then stored via `carray-replace`.

10.19.21 Function `carray-replace`

Syntax:

```
(carray-replace carray item-sequence [from [to]])
```

Description:

The `carray-replace` function is a specialized version of `replace` which works on `carray` objects. It replaces a sub-range of *carray* with elements from *item-sequence*. The replacement sequence need not have the same length as the range which it replaces.

The semantics of *from* and *to* work exactly like the corresponding arguments of the `replace` function, following the same conventions.

The semantics of the `carray-replace` operation itself differs from the `replace` semantics on sequences in one important regard: the `carray` object's length always remains the same.

The range indicated by *from* and *to* is deleted from *carray* and replaced by elements of *item-sequence*, which undergo conversion to the foreign type that defines the elements of *carray*.

If this operation would make the `carray` longer, any elements in excess of the object's length are discarded, whether they are the original elements, or whether they come from *item-sequence*. Under no circumstances does `carray-replace` write an element beyond the length of the underlying storage.

If this operation would make the *carray* shorter (the range being replaced is longer than *item-sequence*) then the downward relocation of items above the replacement range creates a gap at the end of *carray* which is filled with zero bytes.

The return value is *carray* itself.

10.19.22 Function `carray-pun`

Syntax:

```
(carray-pun carray type [offset [size-limit]])
```

Description:

The `carray-pun` creates a new `carray` object which provides an aliased view of the same data that is referenced by the original *carray* object.

The *type* argument specifies the element type used by the returned aliasing array.

If the *offset* argument is specified, then the aliased view is displaced by that many bytes from the start of the *carray* object. The *offset* argument must not be larger than the bytewise length of the array, or an error exception is thrown. The bytewise length of the array is the product of the number of elements and the element size. The default value of *offset* is zero: no displacement.

If *size-limit* is specified, it indicates the size, in bytes, of the aliased view. This limit must not be such that the aliased view would extend beyond the array, or an error exception is thrown. If omitted, *size-limit* defaults to the entire remainder of the array, after the offset. The number of elements of the returned array are then calculated from *size-limit*.

The `carray-pun` function calculates how many elements of *type* fit into *size-limit*. This value becomes the length of the aliasing array which is returned.

Since the returned aliasing array and the original refer to the same storage, modifications performed in one view are reflected in the other.

The aliasing array holds a reference to the original, so that as long as it is reachable by the garbage collector, so is the original. That relationship is severed if `carray-dup` is invoked on the aliasing array.

The meaning of the aliasing depends entirely on the bitwise representations of the types involved.

Note: `carray-pun` does not check whether *offset* is a value that is suitably aligned for accessing elements of *type*; on some platforms that must be ensured.

The `carray-pun` function may be invoked on an object that was itself returned by `carray-pun`.

10.19.23 Functions `carray-uint` and `carray-int`

Syntax:

```
(carray-uint number [type])
(carray-int number [type])
```

Description:

The `carray-uint` and `carray-int` functions convert *number*, an integer, to a binary image, which is then used as the underlying storage for a `carray`.

The *type* argument, a compiled FFI type, determines the element type for the returned `carray`. If it is omitted, it defaults to the `uchar` type, so that the array is effectively of bytes.

Regardless of *type*, these functions first determine the number of bytes required to represent *number* in a big endian format. Then the number of elements is determined for the array, so that it provides at least as that many bytes of storage. The representation of *number* is then placed into this storage, such that its least significant byte coincides with the last byte of that storage. If the number is smaller than the storage provided by the array, it extended with padding bytes on the left, near the beginning of the array.

In the case of `carray-uint`, *number* must be a nonnegative integer. An unsigned representation is produced which carries no sign bit. The representation is as many bytes wide as are required to cover the number up to its most-significant bit whose value is 1. If any padding bytes are required due to the array being larger, they are always zero.

The `carray-int` function encodes negative integers also, using a variable-length two's complement representation. The number of bits required to hold the number is calculated as the smallest width which can represent the value in two's complement, including a sign bit. Any unused bits in the most-significant byte are filled with copies of the sign bit: in other words, sign extension takes place up to the byte size. The sign extension continues through the padding bytes if the array is larger than the number of bytes required to represent *number*; the padding bytes are filled with the value `#b11111111` (255) if the number is negative, or else 0 if it is nonnegative.

10.19.24 Functions `uint-carray` and `int-carray`

Syntax:

```
(uint-carray carray)
(int-carray carray)
```

Description:

The `uint-carray` and `int-carray` functions treat the storage bytes *carray* object as the representation of an integer.

The `uint-carray` function simply treats all of the bytes as a big-endian unsigned integer in a pure binary representation, and returns that integer, which is necessarily always nonnegative.

The `int-carray` function treats the bytes as a two's complement representation. The returned number is negative if the first storage byte of *carray* has a 1 in the most significant bit position: in other words, is in the range `#x80` to `#xFF`. In this case, the two's complement of the entire representation is calculated: all of the bits are inverted, the resulting positive integer is extracted. Then 1 is added to that integer, and it is negated. Thus, for example, if all of the bytes are `#xFF`,

the value -1 is returned.

10.19.25 Functions `fill-carray` and `put-carray`

Syntax:

```
(fill-carray carray [pos [stream]])
(put-carray carray [pos [stream]])
```

Description:

The `fill-carray` and `put-carray` functions perform stream output using the `carray` object as a buffer.

The semantics of these functions is as follows. A temporary buffer is created which aliases the storage of *carray* and this buffer is used as an argument in an invocation of, respectively, the buffer I/O function `fill-buf` or `put-buf`.

The value returned by the buffer I/O function is returned.

The *pos* and *stream* arguments are defaulted exactly in the same manner as by `fill-buf` and `put-buf`, and have the same meaning. In particular, *pos* indicates a byte offset into the `carray` object's storage, not an array index.

11 LISP COMPILATION

11.1 Overview

TXR supports two modes of processing of Lisp programs: evaluation and compilation.

Expressions entered into the listener, loaded from source files via `load`, processed by the `eval` function, or embedded into the **TXR** pattern language, are processed by the *evaluator*. The evaluator expands all macros, and then interprets the program by traversing its raw syntax tree structure. It uses an inefficient representation of lexical variables consisting of heap-allocated environment objects which store variable bindings as Lisp association lists. Every time a variable is accessed, the chain of environments is searched for the binding.

TXR also provides a compiler and virtual machine for more efficient execution of Lisp programs. In this mode of processing, top-level expressions are translated into the instructions of Lisp-oriented virtual machine. The virtual machine language is traversed more efficiently compared to the traversal of the cons cells of the original Lisp syntax tree. Moreover, compiled code uses a much more efficient representation for lexical variables which doesn't involve searching through an environment chain. Lexical variables are always allocated on the stack (the native one established by the operating system). They are transparently relocated to dynamic storage only when captured by lexical closures, and without sacrificing access speed.

TXR provides the function `compile` for compiling individual functions, both anonymous and named. File compilation is supported via the function `compile-file`. The function `compile-toplevel` is provided for compiling expressions in the global environment. This function is the basis for both `compile` and `compile-file`.

The `disassemble` function is provided to list the compiled code in a more understandable way; `disassemble` takes a compiled code object and decodes it into an assembly language presentation of its virtual-machine code, accompanied by a dump of the various information tables.

File compilation via `compile-file` refers to a processing step whereby a source file containing **TXR Lisp** forms (typically named with a `.tl` file name suffix) is translated into an object file (named with a `.tlo` suffix) containing a compiled version of those forms.

The compiled object file can then be loaded via the `load` function instead of the source file. Usually, loading the compiled file produces the same effect as if the source file were loaded. However, note that the behavior of compiled code can differ from interpreted code in a number of ways. Differences in behavior can be deliberately induced. Certain erroneous or dubious situations can also cause compiled code to behave differently from interpreted code.

Compilation not only provides faster execution; compiled files also load much faster than source files. Moreover, they can be distributed unaccompanied by the source files, and resist reverse engineering.

11.2 Top-Level Forms

An important concept in file compilation via `compile-file` is that of the *top-level form*, and how that term is defined. The file compiler individually processes top-level forms; for each such form, it emits a translated image.

In the context of file compilation, a top-level form isn't simply any Lisp form which is not enclosed by another one. Rather, in this specific context, it has this specific definition, which allows some enclosed forms to still be considered top-level forms:

1. If a form appearing in a **TXR Lisp** source file isn't enclosed in another form, it is a top-level form.
2. If a `progn` form is top-level form, then each of its constituent forms is also a top-level form.
3. If a `compile-only` form is top-level form, then each of its constituent forms is also a top-level form.
4. If an `eval-only` form is top-level form, then each of its constituent forms is also a top-level form.
5. If a `load-time` form is top-level form, then its argument is a top-level form.
6. When a macro form is identified as a top-level form, it is macro-expanded as if by `macroexpand` before considering whether it contains top-level forms under rules 2–5.
7. Rules 2–6 are applied recursively.
8. No other forms are top-level forms.

   A top-level form is a *primary* top-level form if it doesn't contain any other top-level forms. This means that it is not a form based on any of the operators `progn`, `compile-only` or `eval-only`.

   Note that the constituent body forms of a `macrolet` or `symacrolet` top-level form are not individual top-level forms, even if the expansion of the construct combines the expanded versions of those forms with `progn`.

   Note: the `eval` function implements a similar concept, specially recognizing `progn`, `compile-only` and `eval-only` top-level forms, taking care to macro-expand and evaluate their constituents separately. In turn, the `load` function, when processing Lisp source, evaluates each primary top-level form as if by using the `eval` function. The result is that the behavior of loaded source and compiled files is consistent in this regard.

11.3 File Compilation Model

The file compiler reads each successive forms from a file, performs a partial expansion on that form, then traverses it to identify all of the top-level forms which it contains. Each top-level form is subject to three actions, either of the latter two of which may be omitted: compilation, execution and emission. Compilation refers to the translation to compiled form. Execution is the invocation of the compiled form. Emission refers to appending an externalized representation of the compiled form (its image) to the output which is

written into the compiled file.

By default, all three actions take place for every top-level form. Using the operators `compile-only` or `eval-only`, execution or emission, or both, may be suppressed. If both are suppressed, then compilation isn't performed; the forms processed in this mode are effectively ignored.

When a compiled file is loaded, the images of compiled forms are read from it and converted back to compiled objects, which are executed in sequence.

Partial expansion means that file compilation doesn't fully expand each form that is encountered. Rather, an incremental expansion is performed, similar to the algorithm used by the `eval` function:

1. First, if *form* is a macro, it is macro-expanded as if by an application of the function `macroexpand`.
2. If the resulting expanded form is a `progn`, `compile-only`, or `eval-only` form, then `compile-file` iterates over that form's argument expressions, compiling each expression recursively as if it were a separate expression.
3. Otherwise, if the expanded form isn't one of the above three kinds of expressions, it is subject to a full expansion and compilation.

Note: the structure of these three processing rules above closely resembles that of the three rules given in the description of the `eval` function, which is the basis for handling source files in `load`. Consequently, macro expansion behaves consistently between `compile-file` and `load` of a source file.

11.4 Treatment of Literals

Programs specify not only code, but also data. Data embedded in a program is called *literal data*. There are restrictions on what kinds of object may be used as literal data in programs subject to file compilation. Programs which stray outside of these restrictions will produce compiled files which load incorrectly or fail to load.

Literal objects arise not only from the use of literal such as numbers, characters and strings, and not only from quoted symbols or lists. For instance, compiled forms which define or reference free variables or global functions require the names of these variables or functions to be represented as literals.

An object used as a literal in file-compiled code must be *externalizable* which means that it has a printed representation which can be scanned to produce a similar object. An object which does not have a readable printed representation will give rise to a compiled file which triggers an exception. Literals which are themselves read from program source code naturally meet this restriction; however, with the use of macros, it is possible to embed arbitrary objects into program code.

If the same object appears in two or more places in the code specified in a single file, the file compilation and loading mechanism ensures that the multiple occurrences of that object in the compiled file become a single object when the compiled file is loaded. For example, if macros are used in such a way that the compiled file defines a function which has a name generated by `gensym`, and there are calls to that function throughout that file, this will work properly: the multiple occurrences of the gensym will appear as the same symbol. However: that symbol in the loaded file will not be identical to any other symbol in the **TXR** image; it will be newly allocated each time the compiled file is loaded.

Interned symbols are recorded in a compiled file by means of their textual names and package prefixes. When a compiled file is loaded, the interned symbols which occur as literals in it are entered into the specified packages under the specified names. The value of the `*package*` special variable has no influence on this.

Circular structures in compiled literals are preserved; on loading, similar circular structures are reproduced.

11.5 Treatment of the Hash-Bang Line

TXR supports the hash-bang mechanism in compiled `.tlo` files, thereby allowing compiled scripts to be executable.

When a source file begins with the `#!` (hash bang) character sequence, the file compiler propagates that line (all characters up to and including the terminating newline) to the compiled file, subject to the following transformation steps:

1. The line is divided into arguments, on the assumption that they are separated by exactly one space.
2. Then, all occurrences of the argument `"--lisp"` are replaced by `"--compiled"`.
3. Next, all arguments which end in the suffix `"txrlisp"` have that suffix replaced by `"txrvm"`.
4. The hash bang line is then reconstituted by joining the arguments with a single space.

Furthermore, certain permissions are propagated from a hash-bang source file to the target file. If the source file is executable to its owner, then the target file is made executable as if by using `chmod` with the `+x` mode: all the executable permissions that are allowed by the current `umask` are are enabled on the target file. If the target file is thus being marked executable, then additional permissions are also treated as follows. If the target file has the same owner as the source file, and the source file's setuid permission bit is set, then this is propagated to the target file. Similarly, if the target file has the same group owner as the source file, and the source file's group execute bit and setgid permission bit are set, then the setgid bit is set on the target file.

11.6 Compiled File Compatibility

TXR's virtual-machine architecture for executing compiled code is evolving, and that evolution has implications for the compatibility between compiled files and the **TXR** executable image.

The basic requirement is that a given version of **TXR** can load and execute the compiled files which that same version has produced.

Furthermore, these files are architecture-independent, except that their encoding is in the local byte order ("endianness") of the host machine. The byte order is explicitly indicated in the files, and the `load` function resolves it. Thus a file produced by **TXR** running on a 64-bit big-endian Power PC can be loaded by **TXR** running on 32-bit x86, which is little endian.

A given **TXR** version may also be capable of loading files produced by an older version, or even ones produced by a newer version. Whether this is possible depends on the versions involved. Furthermore, there is a general issue at play: code compiled by newer versions of **TXR** may require functions that are not present in older versions, preventing that code from running. Newer **TXR** may support new syntax not recognized by older **TXR**, and that syntax may end up in compiled files.

Compiled files contain a minor and major version number (which is independent of the **TXR** version). The `load` function examines these numbers and decides whether the file is loadable, or whether it must be rejected.

The first version of **TXR** which featured the compiler and virtual machine was 191. Older versions therefore cannot load compiled files.

Versions 191 and 192 produce version 1 compiled files, and load only that version.

Versions 193 through 198 produce version 2 compiled files and load only that version.

Version 199 produces version 3 files and loads versions 2 and 3.

Versions 200 through 215 produce version 4 files and load versions 2, 3 and 4.

Versions 216 through 243 produce version 5.0 files and load versions 2, 3, 4 and 5, regardless of minor version.

Versions 244 through 251 produce version 5.1 files and load versions 2, 3, 4 and 5, regardless of minor version.

Versions 252 through 259 produce version 6.0 files and load only version 6, regardless of minor version.

Versions 260 through 285 produce version 7.0 files and load versions 6 and 7, regardless of minor version. Version 261 introduces JSON `#J` syntax. Compiled code which contains embedded JSON literals is not loadable by **TXR** 260 and older.

11.7 Recommendations for Unused Variable Diagnostics

By default, the `unused` diagnostic option is enabled in `*compile-opts*`, causing unused variables to be diagnosed.

The first step in resolving an unused variable diagnostic is to determine whether it is caused by a bug in the code. If so, the resolution is to address the bug.

If the situation isn't a bug, then the diagnostic is a false positive, and may be silenced. There are multiple ways to do that, six of which are given here:

1. **Disable the diagnostic**: for instance, compile `"foo.tl"` with unused warnings disabled:

```
(with-compile-opts (nil unused)
   (compile-file "foo.tl"))
```

2. **Use the `ignore` function**: the compiler specially recognizes the `ignore` function such that when any of its arguments are lexical variables, they are marked used:

```
 (defun stub-function (arg1 arg2)
   (ignore arg1 arg2))
```

   Note that an `ignore` call may be elided if it occurs in dead code, in which case it won't have the right effect:

```
(defun unused-arg (arg)    ;; diagnosed as unused
  (when (= (+ 2 2) 5)
    (ignore arg)           ;; wrongly placed
    (dead-code)))

(defun unused-arg (arg)    ;; no diagnostic
  (ignore arg)             ;; correctly placed
  (when (= (+ 2 2) 5)
    (dead-code)))
```

3. **Use the `use` function**. In the following code, parameter `arg` is diagnosed as unused on platforms in which the equality being tested is false, since the expression is constant. In situations like this,

the variable is not unused, but only conditionally so. Therefore the name of the `ignore` function doesn't express the intent very well. The `use` function may be stylistically preferred:

```
(defun platform-specific-action (arg)
   (use arg)
   (if (eql (sizeof wchar) 2)
      (do-something arg)))
```

However, unlike `ignore`, `use` takes exactly one argument, and returns that argument rather than `nil`.

4. **Use an uninterned symbol:** unused variable diagnostics are not reported against variables named by uninterned symbols.

```
(lambda (x y) y)    ;; unused x diagnosed
(lambda (#:x y) y)  ;; no diagnostic
```

5. **In pattern matching, put catch-all variable to use**:

Examples:

```
(tree-case obj
   ((a b) (calculate-something a b))
   (else (transform obj))) ;; unused else

(tree-case obj
   ((a b) (calculate-something a b))
   (else (transform else))) ;; diagnostic gone

(match-case obj
   ((@a @b) (calculate-something a b))
   (@else (transform obj))) ;; unused else

(match-case obj
   ((@a @b) (calculate-something a b))
   (@else (transform else))) ;; diagnostic gone
```

6. **In macro parameter lists use t symbol :** in macro-style parameter lists, any variable may be replaced by the `t` symbol to consume a value without binding a variable. This is intended for suppressing unused variable warnings:

```
(defmacro foo (x y)  ;; y unused
  ^(a b c ,x))

(defmacro foo (x t)  ;; no diagnostic
  ^(a b c ,x))

(tree-bind (a . (b . c)) obj ;; a, b, unused
  c)

(tree-bind (t . (t . c)) obj ;; no diagnostic
  c)
```

11.8 Semantic Differences between Compilation and Interpretation

The `compile-only` and `eval-only` operators can be used to deliberately produce code which behaves differently when compiled and interpreted. In addition, unwanted differences in behavior can also occur. The situations are summarized below.

11.8.1 Differences due to `load-time`

Forms evaluated by `load-time` are treated differently by the compiler. When a top-level form is compiled, its embedded `load-time` forms are factored out such that the compiled image of the top-level form will evaluate these forms before other evaluations take place. The interpreter doesn't perform this factoring; it evaluates a `load-time` form when it encounters it for the first time.

11.8.2 Treatment of literals

The compiler identifies multiple occurrences of equivalent strings and bignum integers that occur as literals, and condenses each one to a single instance, within the scope of the compilation. The scope is possibly as wide as a file.

If the literal `"abc"` appears in multiple places in the same file that is processed by `compile-file`, in the resulting compiled file, there may be just a single `"abc"` object. For instance, if the file contains two functions:

```
(defun f1 () "abc")
(defun f2 () "abc")
```

when compiled, these will return the same object such that

```
(eq (f1) (f2)) -> t
```

No such de-duplication is performed for interpreted code.

Consequently, code which depends on multiple occurrences of these objects to be distinct objects may behave correctly when interpreted, but misbehave when compiled. Or vice versa. One example is code which modifies a string literal. Under compilation, the change will affect all occurrences of that literal that have been merged into one object. Another example is an expression like `(eq "abc" "abc")`, which yields `nil` under interpretation because the two strings are distinct object in spite of appearing side by side in the syntax, but `t` when compiled, since they denote the same string object.

In the future, objects other than strings and bignums may be similarly consolidated, such as lists and vectors, which means that interpreted code which works today when compiled may misbehave in the future.

Note that objects which are literally notated in source code are not the only kinds of objects considered to be literals. Objects which are constructed by macros and inserted into macro-expansions are also literals. Literals are self-evaluating objects that appear as expressions in the syntax which remains after macro-expansion, as well as arguments of the `quote` operator. If a macro calculates a new string each time it is expanded, and inserts it into the expansion as a literal, the compiler will identify and consolidate groups of such strings that are identical.

11.8.3 Treatment of symbols

A source file may contain unqualified symbol tokens which are interned in the current package.

In contrast, a compiled file encodes symbols with full package qualification. When a compiled file is loaded, the current package at that time has no effect on the symbols in the compiled file, even if those

symbols were specified as unqualified in the original source file.

This difference can lead to surprising behaviors. Suppose a source file contains references to functions or variables or other entities which do not exist. Furthermore, suppose the entities were referenced, in that file, using unqualified symbols which didn't exist, and were expected to come from a different package from the one where they ended up interned. For instance, supposed the file is being processed in a package called `abc` and is expecting to use a function `calc` which should come from the `xyz` package. Unfortunately, no such symbol exists. Therefore, the symbol is interned as `abc:calc` and not `xyz:calc`. In that case, it should be sufficient to ensure that the `xyz:calc` function exists, and then reload the source file. The unqualified symbol token `calc` in that file will be correctly resolved to `xyz:calc` that time. However, if the file is compiled, reloading will not be sufficient. Even though the symbol `xyz:calc` exists, the file will continue to try to refer a function using the symbol `abc:calc` which comes from a fully qualified representation stored in the compiled file. The file will have to be recompiled to fix the issue.

11.8.4 Treatment of unbound variables

Unbound variables are treated differently by the compiler. A reference to an unbound variable is treated as a global lexical access. This means that if a variable access is compiled first and then a `defvar` is processed which introduces the variable as a dynamically scoped ("special") variable, the compiled code will not treat the variable as special; it will refer to the global binding of the variable, even when a dynamic binding for that variable exists. The interpreter treats all variable references that do not have lexical bindings as referring to dynamic variables. The compiler treats a variable as dynamic if a `defvar` has been processed which marked that variable as special.

11.8.5 Unbound symbols in `dwim`

Arguments of a `dwim` form (or the equivalent bracket notation) which are unbound symbols are treated differently by the compiler. The code is compiled under the assumption that all such symbols refer to global functions. For instance, if neither `f` nor `x` are defined, then `[f x]` will be compiled under the assumption that they are functions. If they are later defined as variables, the compiled code will fail because no function named `x` exists. The interpreter resolves each symbol in a `dwim` form at the time the form is being executed. If a symbol is defined as a variable at that time, it is accessed as a variable. If it defined as a function, it is accessed as a function.

11.8.6 Bound symbols in `dwim`

The symbolic arguments of a `dwim` form that refer to global bindings are also treated differently by the compiler. For each such symbol, the compiler determines whether it refers to a function or variable and, further, whether the variable is global lexical or special. This treatment of the symbol is then cemented in the compiled code; the compiled code will treat that symbol that way regardless of the run-time situation. By contrast, the interpreter performs this classification each time the arguments of a `dwim` form are evaluated. The rules are otherwise the same: if the symbol is bound as a variable, it is treated as a variable. If it is bound as a function, it is treated as a function. If it has both bindings, it is treated as a variable. The difference is that this is resolved at compile time for compiled code, and at evaluation time for interpreted code.

11.8.7 File-wide insertion of gensyms

The following degenerate situation occurs, illustrated by example. Suppose the following definitions are given:

```
(defvarl %gensym%)

(defmacro define-secret-fun ((. args) . body)
  (set %gensym% (gensym))
  ^(defun ,%gensym% (,*args) ,*body))
```

```
(defmacro call-secret-fun (. args)
  ^(,%gensym% ,*args))
```

The idea is to be able to define a function whose name is an uninterned symbol and then call it. An example module might use these definitions as follows:

```
(define-secret-fun (a) (put-line `a is @a`))

(call-secret-fun 42)
```

The effect is that the second top-level form calls the function, which prints 42 to standard out. This works both interpreted and compiled with `compile-file`. Each of these two macro calls generates a top-level form into which the same gensym is inserted. This works under file compilation due to a deliberate strategy in the layout of compiled files, which allows such uses. Namely, the file compiler combines multiple top-level forms into a single object, which is read at once, and which uses the circle notation to unify gensym references.

However, suppose the following change is introduced:

```
(define-secret-fun (a) (put-line `a is @a`))

(defpackage foo) ;; newly inserted form

(call-secret-fun 42)
```

This still works when interpreted, and compiles successfully. However, when the compiled file is loaded, the compiled version of the `call-secret-fun` form fails with an error complaining that the `#:g0039` (or other gensym name) function is not defined.

This is because for this modified source file, the file compiler is not able to combine the compiled forms into a single object. It would not be correct to do so in the presence of the `defpackage` form, because the evaluation of that form affects the subsequent interpretation of symbols. After the package definition is executed, it is possible for a subsequent top-level form to refer to a symbol in the `foo` package such as `foo:bar` to occur, which would be erroneous if the package didn't exist.

The file compiler therefore arranges for the compiled forms after the `defpackage` to be emitted into a separate object. But that division in the output file consequently prevents the occurrences of the gensym to resolve to the same symbol object.

In other words, the strategy for allowing global gensym use is in conflict with support for forms which have a necessary read-time effect such as `defpackage`.

The solution is to rearrange the file to unravel the interference, or to use interned symbols instead of gensyms.

11.8.8 Delimited Continuations

There are differences in behavior between compiled and interpreted code with regard to delimited continuations. This is covered in the Delimited Continuations section of the manual.

11.9 Compilation Library

11.9.1 Function `compile-toplevel`

Syntax:

```
(compile-toplevel form expanded-p)
```

Description:

The `compile-toplevel` function takes the Lisp form *form* and compiles it. The return value is a *virtual-machine description* object representing the compiled form. This object isn't of function type, but may be invoked as if it were a function with no arguments.

Invoking the compiled object is expected to produce the same effect as evaluating the original *form* using the `eval` function.

The *expanded-p* argument indicates that *form* has already been expanded and is to be compiled without further expansion.

If *expanded-p* is `nil`, then it is subject to a full expansion.

Note: in spite of the name, `compile-toplevel` makes no consideration whether or not *form* is a "top-level form" according to the definition of that term as it applies to `compile-file` processing.

Note: a form like `(progn (defmacro foo ()) (foo))` will not be processed by `compile-toplevel` in a manner similar to the processing by `eval` or `compile-file`. In this example, `defmacro` form will not be evaluated prior to the expansion of `(foo)` (and in fact not evaluated at all) and so the latter expression isn't correctly referring to that macro. The form `(progn (macro-time (defmacro foo ())) (foo))` can be processed by `compile-toplevel`; however, the macro definition now takes place during expansion, and isn't compiled. The `compile-file` function has no such issue when it encounters such a form at the top-level, because that function will consider a top-level `progn` form to consist of multiple top-level forms that are compiled individually, and also executed immediately after being compiled.

Example

```
;; compile (+ 2 2) form and execute to calculate 4
;;
(defparm comp (compile-toplevel '(+ 2 2)))

(call comp) -> 4

[comp] -> 4
```

11.9.2 Function `compile`

Syntax:

```
(compile function-name)
(compile lambda-expression)
(compile function-object)
```

Description:

The `compile` function compiles functions.

It can compile named functions when the argument is a *function-name*. A function name is a symbol denoting an existing interpreted function, or compound syntax such as `(meth type name)` to refer to methods. The code of the interpreted function is retrieved, compiled in a manner which produces an anonymous compiled function, and then that function replaces the original

function under the same name.

If the argument is a lambda expression, then that function is compiled.

If the argument is a function object, and that object is an interpreted function, then its code and lexical environment are retrieved and compiled.

In all cases, the return value of `compile` is the compiled function.

Note: when an interpreted function object is compiled, the compiled environment does not share bindings with the original interpreted environment. Modifications to the bindings of either environment have no effect on the other. However, the objects referenced by the bindings are shared. Shared bindings may be arranged using the `hlet` or `hlet*` macros.

11.9.3 Functions `compile-file` and `compile-update-file`

Syntax:

```
(compile-file input-path [output-path])
(compile-update-file input-path [output-path])
```

Description:

The `compile-file` function reads forms from an input file, and produces a compiled output file.

First, *`input-path`* is converted to a *tentative pathname* as follows.

If *`input-path`* specifies a pure relative pathname, as defined by the `pure-rel-path-p` function, then a special behavior applies. If an existing load operation is in progress, then the special variable `*load-path*` has a binding. In this case, `load` will assume that the relative pathname is a reference relative to the directory portion of that pathname.

If `*load-path*` has the value `nil`, then a pure relative *`input-path`* pathname is used as-is, and thus resolved relative to the current working directory.

The tentative pathname is converted to an *actual input pathname* as follows. Firstly, if the tentative pathname ends with one of the suffixes `.tl` or `.txr` then it is considered suffixed, otherwise it is considered unsuffixed. If it is suffixed, then the actual pathname is the same as the tentative pathname. In the unsuffixed case, two possible actual input pathnames are considered. First, if the unsuffixed path refers to a file that can be opened, then that unsuffixed path is taken as actual path. Otherwise, the suffix `.tl` is added to the tentative pathname, and that becomes the actual path.

If the actual path ends in the suffix `.txr` then the behavior is unspecified.

If the *`output-path`* parameter is given an argument, then that argument specifies the output path. Otherwise the output path is derived from the tentative input path as follows. If the tentative input path is unsuffixed, then `.tlo` is added to it to produce the output path. Otherwise, the suffix is removed from the tentative input path and replaced with the `.tlo` suffix.

The `compile-file` function binds the variables `*load-path*` and `*package*` similarly to the `load` function.

Over the compilation of the input file, `compile-file` establishes a new dynamic binding for several special variables. The variable `*load-path*` is given a new binding containing the actual input pathname. The `*package*` variable is also given a new dynamic binding, whose

value is the same as the existing binding. Thus if the compilation of the file has side the effect of altering the value of `*package*`, that effect will be undone when the binding is removed after the compilation completes.

Compilation proceeds according to the File Compilation Model.

If the compilation process fails to produce a successful translation for each form in the input file, the output file is removed.

The `compile-update-file` function differs from `compile-file` in the following regard: compilation is performed only if the input file is newer than the output file, or else if the output file doesn't exist.

The `compile-file` always returns `t` if it terminates normally, which occurs if it successfully translates every form in the input file, depositing the translation into the output file. If compilation fails, `compile-file` terminates by throwing an exception.

The `compile-update-file` function returns `t` if it successfully compiles, similarly to `compile-file`. If compilation is skipped, the function returns `nil`.

Note: the following idiom may be used to load a file, compiling it if necessary:

```
(or (compile-update-file "file")
    (load-file "file"))
```

However, note that it relies on the effect of compiling a source file being the same as the effect of loading the compiled file. This can only be true if the source file contains no `compile-only` or `eval-only` top-level forms.

Two or more compiled files that are compiled by the same version of **TXR** may be catenated together to produce a single `.tlo` file. Such a file may be loaded by the `load` function. The behavior of loading such a file may differ from loading the individual files, because such a `load` is treated as a single operation.

11.9.4 Special variable `*opt-level*`

Description:

The special variable `*opt-level*` provides control over compiler optimizations.

The variable takes on integer values. If the value is `nil`, it is interpreted as zero. The meaningful range is from 0 to 6. The initial value of the variable is 6.

The meanings of the values are as follows:

0 Almost all optimizations are disabled, except for some strength reductions of instances of he `equal` function, to take advantage of certain conditional instructions.

1 Constant folding is applied, as well as algebraic reductions to list processing and arithmetic code. Two-argument calls to several common arithmetic operators are translated into calls to more efficient two-argument internal functions.

2 Blocks which can be easily confirmed not to be used as exit points are removed. Variable frames in which no lexically captured variables are bound, and no dynamic variables are bound, are eliminated.

3 Lambda expressions and calls to combinator functions such as `chain` and `andf` are lifted to load time, if possible.

4 Control flow optimizations are applied: jump threading and elimination of unreachable code. Some peephole optimizations are applied to improve certain instruction patterns.

5 Data flow optimizations are applied, such as elimination of dead register moves, or useless propagations of values from one register to another. More peephole optimizations are applied.

6 Certain more rarely applicable optimizations are applied which reduce code size by merging some identical code blocks, or improving some more rarely occurring instruction patterns.

11.9.5 Macro `with-compilation-unit`

Syntax:

```
(with-compilation-unit form*)
```

Description:

When a file is processed by `compile-file`, certain actions, such as the issuance of diagnostics about undefined functions and variables, are delayed until the file is completely processed.

The `with-compilation-unit` macro allows these actions to be collectively deferred until multiple files are completely processed.

The macro evaluates each enclosed *form* in a single compilation environment. After the last *form* is evaluated, deferred actions of any enclosed `compile-file` forms are performed, and then the value of the last *form* is returned.

It is permissible to nest `with-compilation-unit` forms, lexically or dynamically. The outermost invocation of `with-compilation-unit` dominates; all deferred `compile-file` actions are held until the outermost enclosing `with-compilation-unit` terminates.

11.9.6 Operators `compile-only` and `eval-only`

Syntax:

```
(compile-only form*)
(eval-only form*)
```

Description:

These operators take on a special behavior only when they appear as top-level forms in the context of file compilation. When a `compile-only` or `eval-only` form is processed by the evaluator rather than the compiler, or when it is processed outside of file compilation, or when it is appears as other than a top-level form even under file compilation, then these operators behave in a manner identical to `progn`.

When a `compile-only` form appears as a top-level form under file compilation, it indicates to the file compiler that the *form*s enclosed in it are not to be evaluated. By default, the file compiler executes each top-level form after compiling it. The `compile-only` operator suppresses this evaluation.

When an `eval-only` form appears as a top-level form under file compilation, it indicates to the file compiler that the *form*s enclosed in it are not to be emitted into the output file. By default, the file compiler includes the compiled image in the output written to the output file. The `eval-only` operator suppresses this inclusion.

Forms which are surrounded by both an `eval-only` form and a `compile-only` form are neither executed nor emitted into the output file. In this situation, the forms are skipped entirely; no compilation takes place.

Notes:

The `compile-file` function not only compiles, but also executes every form for the following reason: the correct compilation of forms can depend on the execution of earlier forms. For instance, code may depend on macros. Macros may in turn depend on functions and variables. All those definitions are required in order to compile the dependent code. Those dependencies may be in a separate file which is loaded by a `load` form; that `load` form must be executed.

Note that execution of a form implies that the `load-time` forms that it contains are evaluated (prior to other evaluations). Suppression of the execution of a form also suppresses the evaluation of `load-time` forms.

Situations in which `compile-only` is useful are those in which it is desirable to stage the execution of some top-level form into the compiled file, and not have it happen during compilation. For instance:

```
;; in a main module
(compile-only (start-application))
```

It is not desirable to have the file compiler try to start the application as a side effect of compiling the main module. The right behavior is to compile the `(start-application)` top-level form so that this will happen when that module is loaded.

Situation in which `eval-only` is useful is for specifying forms which have a compile-time effect only, but are not propagated into the compiled file.

For example, since the correct treatment of literal symbols occurring in a compiled file does not depend on the `*package*` variable, in many cases, the `in-package` invocation in the file can be wrapped with `eval-only`:

```
(eval-only (in-package app))
```

The `in-package` form must be evaluated during compilation so that the remaining forms are read in the correct package. However the loading of the compiled versions of those forms doesn't require that package to be in effect; thus a compiled image of the `in-package` form need not appear in the compiled file.

Macros definitions may be treated with `eval-only` if the intent is only to make the expanded code available in the compiled file, and not to propagate compiled versions of the macros which produced it.

11.9.7 Structure `compile-opts`

Syntax:

```
(defstruct compile-opts ()
  shadow-fun shadow-var shadow-cross unused)
```

Description:

The `compile-opts` structure represents compiler options: its slots are variables which affect compiler behavior. The compiler expects the special variable `*compile-opts*` to hold a

`compile-opts` structure. It is recommended to manipulate options using the `with-compile-opts` macro.

Currently, all of the options are diagnostic. In the future, there may be other kinds of options.

Diagnostic options take on the values `nil`, `t`, `:warn` and `:error`. The `t` and `:warn` value are synonyms. A value of `nil` means that the option is disabled. The `t` and `:warn` values mean that the diagnostic controlled by the option will be emitted as a warning. The `:error` value indicates that the diagnostic will be an error.

The slots of `compile-opts` are as follows:

`shadow-fun`
: Diagnostic option, off by default. This option controls whether a diagnostic is emitted whenever a lexical function shadows another lexical function, a global function or a global macro. Note: shadowing of local macros is not diagnosed, because the compiler operates on code in which macros no longer exist.

`shadow-var`
: Diagnostic option, off by default. This option controls whether a diagnostic is emitted whenever a lexical variable shadows another lexical variable, a global variable, or a global macro. Note: shadowing of local macros is not diagnosed, because the compiler operates on code in which macros no longer exist. Note: special variables are not diagnosed for shadowing.

`shadow-cross`
: Diagnostic option, off by default. This option controls whether a diagnostic is emitted whenever a lexical function shadows a variable, or vice versa: whether a lexical variable shadows a function.

`unused`
: Diagnostic option, set to warn by default. This option controls whether a diagnostic is emitted whenever a lexical variable is defined whose value is not used. Variables whose names are uninterned symbols are exempt from this diagnostic. The rationale is that uninterned symbols are used for naming machine-generated variables, in generated code such as macro expansions. Situation in which a machine-generated variable is unused arise fairly often, and are and are not the result of a programming error. For the purpose of this diagnostic, what constitutes use of a variable is an access to its value, which isn't optimized away before being noted. Storing a value isn't use. An example of an access which is optimized away before being noted is an access which occurs in trivially dead code: for instance `(if nil a)` does not access `a` because the compiler discards the *`then`* expression of an `if` whose *`test`* expression is constantly false. The discarded expression is never traversed in a way that would cause it to be noted as accessing the `a` variable.

11.9.8 Special variable `*compile-opts*`

Description:

The special variable `*compile-opts*` holds a value of type `compile-opts` which is a structure type. It is recommended to manipulate options using the `with-compile-opts` macro.

11.9.9 Macro `with-compile-opts`

Syntax:

```
(with-compile-opts {(value option*) | form}*)
```

Description:

The `with-compile-opts` macro takes zero or more arguments. Each argument is either a clause which affects compiler options or else an ordinary *form* which is evaluated in a context in which the `*compile-options*` variable has been affected by all of the previous clauses.

The `with-compile-opts` macro arranges for `*compile-options*` to be freshly bound, as if by the `let` operator. The variable is then given a value which is a copy of the previous binding. The option-affecting clauses then operate destructively on this copy. When the macro terminates, the binding is undone and thus the prior value of `*compile-options*` is restored.

The clauses which operate on options have the syntax of a value followed by one or more symbols which must be the names of options which are compatible with that value. The clause indicates that all those options take on that value.

Example:

The following expression specifies that the file `"foo.tl"` is to be compiled with function and variable shadowing treated as error, but unused variable checking disabled. Then compile `"bar.tl"` with unused variable checking enabled.

```
(with-compile-opts
   (:error shadow-var shadow-fun)
   (nil unused)
   (compile-file "foo.tl")
   (:warn unused)
   (compile-file "bar.tl"))
```

11.9.10 Macro `load-time`

Syntax:

```
(load-time form)
```

Description:

The `load-time` macro makes it possible for a program to evaluate a form, such that, subsequently, the value of that form is then treated as if it were a literal object.

Literals are pieces of the program syntax which are not evaluated at all. On the other hand, the values of expressions are not literals.

From time to time, certain situations benefit from the program being able to perform an evaluation, and then have the result of that evaluation treated as a literal.

The `macro-time` macro makes this possible in its particular manner: that macro allows one or more expressions to be evaluated during macro expansion. The result of the `macro-time` is then quoted and substituted in place of the expression. That result then appears as a true literal to the executing code.

The `load-time` macro similarly arranges for the single form *form* to be evaluated. However, this evaluation doesn't take place at expansion time. It is delayed until the program executes.

What exactly "delayed until the program executes" means depends on whether `load-time` is used in compiled or interpreted code, and in what situation is it compiled.

If the `load-time` form appears in interpreted code, then the exact time when *form* is evaluated

is unspecified. The evaluator may identify all `load-time` forms which occur anywhere in a top-level expression, and perform their evaluations immediately, before evaluating the form itself. Then, when the `load-time` forms are encountered again during the evaluation of the form, they simply retrieve the previously evaluated values as if they were literal. Or else, the evaluation may be performed late: when the `load-time` form itself is encountered during normal evaluation. In that case, *form* will still be evaluated only once and then its value will be be inserted as a literal in subsequent reevaluations of that `load-time` form, if any.

If a `load-time` form appears in a non-top-level expression which is compiled, the compiler arranges for the compiled version of *form* to be executed when compiled version of the entire expression is executed. This execution occurs early, before the execution of forms that are not wrapped in `load-time`. The value produced by `form` is entered into the static data vector associated with the compiled top-level expression, which also holds ordinary literals. Whenever the value of that `load-time` form is required, the compiled code references it from the data vector as if it were a true literal.

When a `load-time` top-level form is processed by `compile-file`, it has no unusual semantics; the effect is that it is replaced by its argument *form*, which is in that case also considered a top-level form.

The implications of the translation scheme may be understood separately from the perspective of code processed with `compile-toplevel`, `compile` and `compile-file`.

A `load-time` form appearing in a form passed to `compile-toplevel` is translated such that its embedded *form* will be executed each time the virtual-machine description returned by `compile-toplevel` is executed, and the execution of all such forms is placed ahead of other code.

A `load-time` form appearing in an interpreted function which is processed by `compile` is evaluated immediately, and its value becomes a literal in the compiled version of the function.

A `load-time` form appearing as a non-top-level form inside a file that is processed by `compile-file` is compiled along with that form and deposited into the object file. When the object file is loaded, each compiled top-level form is executed. Each compiled top-level form's `load-time` calculations are executed first, and the corresponding *form* values become literals at that point. This execution order is individually ensured for each top-level form. Thus, the `load-time` forms in a given top-level form may rely on the side-effects of prior top-level forms having taken place. Note that, by default, `compile-file` also immediately executes each top-level form which it compiles and deposits into the output file. This execution is equivalent to a load; it causes `load-time` forms to be evaluated. The `compile-only` operator must be used around `load-time` forms which must be evaluated only when the compiled file is loaded, and not at compile time.

In all situations, the evaluation of *form* takes place in the global environment. Even if the `load-time` form is surrounded by constructs which establish lexical bindings, those lexical bindings aren't visible to *form*. Which dynamic bindings are visible to *form* depends on the exact situation. If a `load-time` form occurs in code that had been processed by `compile-file` and is now being loaded by `load`, then the dynamic environment in effect is the one in which the `load` occurred, with any modifications to that environment that were performed by previously executed forms. If a `load-time` form occurs in code that had been processed by `compile-toplevel`, then *form* is evaluated in the dynamic environment of the caller which invokes the execution of the resulting compiled object. When a `load-time` form occurs in the code of an function being processed by `compile`, then *form* is evaluated in the dynamic environment of the caller which invokes `compile`. If a `load-time` form occurs in a form processed processed by the evaluator, it is unspecified whether it takes place in the original dynamic environment in which the evaluator

was invoked, or whether it is in the dynamic environment of the immediately enclosing form which surrounds the `load-time` form.

A `load-time` form may be nested inside another `load-time` form. In this situation, two cases occur.

If the two forms are not embedded in a `lambda`, or else are embedded in the same `lambda`, then the inner `load-time` form is superfluous due to the presence of the outer `load-time`. That is to say, the inner `(load-time` *`form`*`)` expression is equivalent to *`form`*, because the outer form already establishes its evaluation to be in a load-time context.

If the inner `load-time` form occurs in a `lambda`, but the outer form occurs outside of that `lambda`, then the semantics of the inner `load-time` form is relevant and necessary. This is because expressions occurring in a `lambda` are evaluated when the `lambda` is called, which may take place from a non-load-time context, even if the `lambda` itself was produced in a load-time context.

An expression being embedded in a `lambda` means that it appears either in the `lambda` body, or else in the parameter list as the initializing expression for an optional parameter.

Notes:

When interpreted code containing `load-time` is evaluated, a mutating side effect may take place on the tree structure of that code itself as a result of the `load-time` evaluation. If that previously evaluated code is subsequently compiled, the compiled translation may be different from compiling the original unevaluated code. Specifically, the compiler may take advantage of the `load-time` evaluation which had already taken place in the interpreter, and simply take that value, and avoid compiling *`form`* entirely. This also has implications on the dynamic environment that is in effect when *`form`* is evaluated. If *`form`* is evaluated by the interpreter, then it interacts with the dynamic environment which as in effect in that situation; then when the compiler later just takes the result of that evaluation, the compiler's dynamic environment is irrelevant since *`form`* isn't being evaluated any more.

If *`form`*, when evaluated multiple times, potentially produces a different value on each evaluation, this has implications for the situation when an object produced by `compile-toplevel` is invoked multiple times. Each time such an object is invoked, the `load-time` forms are evaluated. If they produce different values, then it appears that the values of literals are changing. All lexical closures derived from the same compiled object share the same literal data. The `load` function never evaluates a compiled expression more than once. If the same compiled file is loaded more than once, a new compiled object instance is produced from each compiled expression, carrying its own storage area for literals. The `compile` function also never evaluates a compiled expression more than once; it produces a compiled object, and then executes it once in order to obtain a lexical closure which is returned. Invoking the closure doesn't cause the `load-time` expressions to be evaluated.

The `load-time` form is subject to compiler optimizations. A top-level expression is assumed to be evaluated at load time, so `load-time` does nothing in a top-level expression. It becomes active inside forms embedded in a `lambda` expressions. Since `load-time` may be used to hoist calculations outside of loops, `load-time` is also active in those parts of loops which are repeatedly evaluated.

The use of `load-time` is similar to defining a variable and then referring to the variable. For instance, a file containing this:

```
(defvarl a (list 1 2))
```

```
(defun f () (cons 0 a))
```

is similar to

```
(defun f () (cons 0 (load-time (list 1 2))))
```

When either file is loaded, in source or compiled form, `list` expression is evaluated at load time, and then when `f` is invoked, it retrieves the list.

Both approaches have advantages. The variable-based approach gives the value a name. The semantics of the variable is straightforward. The variable `a` can easily be assigned a new value. Using its name, the variable can be inspected from the interactive listener. The variable can be referenced from multiple top-level forms directly; it is not a static datum tied to a table of literal values that is tied to a single top-level form. Furthermore, the use of `defvar`/`defvarl` versus `defparm`/`defparml` controls whether the variable gets replaced with a new value when the file is reloaded.

The advantage of `load-time` is that it doesn't require a separate top-level form to achieve its load-time effect: the expression is simply nested at the point where it is needed. The `load-time` form can therefore be generated by macros, whose expansions cannot inject extra top-level forms into the site where they are invoked. If a macro writer would like some form to be evaluated at load time and its value accessible in a macro expansion that appears arbitrarily nested in code, then `load-time` may provide the path to a straightforward implementation strategy. Access to a `load-time` value is fast because it doesn't involve referencing through a variable binding; compiled code accesses the value directly via its fixed position in the static data table associated with that code. This advantage is insignificant, however, because access to lexical variables in compiled code is similarly fast, and a value can easily be propagated from a global variable to a lexical for the sake of speed. That said, `load-time` eliminates that copying step too.

A `load-time` is also useful when the value is not required, and instead the form produces a useful effect, which should be hoisted to load time. For instance, consider a macro which produces the following expansion:

```
(progn (load-time (defvar #:g0025)) (other-logic ... #:g0025))
```

no matter where this expansion is inserted, `compile-file` and `load` will ensure that the `defvar` is executed once, when the compiled file is loaded, as if that `defvar` appeared on its own as a top-level form. Then the `other-logic` form can refer to the variable, without the `defvar` being evaluated on each execution of the `progn`.

The author of a macro can use `load-time` to stage the evaluation of global effects that the macro expansion depends on simply by bundling these effects into the expansion, wrapped in `load-time`.

Dialect Note:

The `load-time` macro is similar to the ANSI Common Lisp `load-time-value` special operator. It doesn't support the *`read-only-p`* argument featured in the ANSI CL operator. The semantics of `load-time` is somewhat more precisely specified in terms of concrete implementation concepts. The ANSI CL `load-time-value` may evaluate *`form`* more than once in interpreted code; effectively, the ANSI CL implementation may treat `(load-time-value x)` as `(progn x)`. This is not true of **TXR Lisp**'s `load-time` which requires once-only evaluation even in interpreted code. The name `load-time` is used instead of `load-time-value` for several reasons. Firstly, `load-time` is useful for staging effects, like definitions, to load time, even when the resulting value is not used. Secondly, unlike **TXR Lisp**, ANSI CL features

multiple values: a form can yield zero or more values. The ANSI CL `load-time-value` operator, however, is restricted to yielding a single value, and its name may have been chosen to emphasize this aspect/restriction. That doesn't apply in the context of **TXR Lisp** in which all expressions which terminate normally yield exactly one value, making `"-value"` a suffix that adds no value. Lastly, `load-time` is shorter, and harmonizes with `macro-time`, which existed earlier.

11.9.11 Function `disassemble`

Syntax:

```
(disassemble function-name)
(disassemble function)
(disassemble compiled-expression)
```

Description:

The `disassemble` function presents a disassembly listing of the virtual-machine code of a compiled function or form. It also presents the literal data contained in that compiled object in a tabular form which is readily cross-referenced with the disassembly listing.

If the argument is a *`function-name`* then the function object is retrieved from the binding indicated by the name, in the global namespace. That object is then treated as if it were the *`function`* argument.

A *`function`* argument is one that is a function object. Only compiled virtual-machine functions can be disassembled; other kinds of functions are rejected by `disassemble`.

The `disassemble` function will also process the *`complied-expression`* object that is returned by the `compile-toplevel` function.

In the case of *`function`*, the entire compiled form containing *`function`* is disassembled. That form usually contains code which is external to the function, even possibly other functions. The disassembly listing indicates the entry point in the code block where the execution of *`function`* begins.

The `disassemble` function returns its argument.

11.9.12 Function `dump-compiled-objects`

Syntax:

```
(dump-compiled-objects stream object*)
```

Description:

The `dump-compiled-objects` function writes compiled objects into *`stream`* in the same format as the `compile-file` function.

Unlike under `compile-file`, the output is written into an arbitrary stream rather than a named file. The objects aren't specified by the to-be-compiled syntax processed from a source file, but rather as zero or more arguments which specify objects that are already compiled.

Each *`object`* must be be one of three kinds of values:

1. a virtual-machine-description object returned by `compile-toplevel` function; or
2. a compiled function object, satisfying the function `vm-fun-p`; or else
3. the name of a compiled function object, which may take any of the forms suitable as arguments to the `symbol-function` function.

First, `dump-compiled-objects` writes some preamble information into *stream*. Then, for each *object* that is not already a virtual-machine description, its corresponding virtual-machine description is retrieved. The virtual-machine description is converted into the externalized format required for the object format and that externalized format is written into *stream*. The `object` argument are thus processed in left-to-right order.

If exactly one call to `dump-compiled-objects` is used to populate an initially empty file, and no other data are written into the file, then that file is a valid compiled file. If that file is processed by `load-file` then each of the externalized forms is converted to a virtual-machine description and executed.

Note that virtual-machine descriptions are not functions. A function's virtual-machine description is the compiled version of the top-level form whose evaluation produced that function.

For example, if the following top-level form is compiled and executed, two functions are defined:

```
(let ()
  (defun a ())
  (defun b ()))
```

Then, the following two expressions all have the same effect on stream `s`:

```
(dump-compiled-objects s 'a)
(dump-compiled-objects s 'b)
```

Whether the `a` or `b` symbol is used to specify the object to be dumped, the same virtual-machine description is externalized and deposited into the stream. That machine description, when loaded and executed, defines two functions.

12 INTERACTIVE LISTENER

12.1 Overview

On some target platforms, **TXR** provides an interactive listener, which is invoked using the `-i` command-line option, or by executing `txr` with no arguments. The interactive listener provides features like visual editing of the command line, tab completion on **TXR Lisp** symbols, and history recall.

12.2 Basic Operation

The interactive listener prints a numbered prompt. The number in the prompt increments with every command. The first command line is numbered 1, the second one 2 and so forth.

The listener accepts input characters from the terminal. Characters are either interpreted as editing commands or other special characters, or else are inserted into the editing buffer. However, control characters which don't correspond to commands are silently rejected.

The carriage return character generated by the Enter key indicates that a complete line has been entered, and it is to be interpreted. The listener parses the line as a **TXR Lisp** expression, evaluates it, and prints the resulting value. If the evaluation of the line throws an exception, the listener intercepts the exception and prints information about it preceded by two asterisks and a space. These asterisks distinguish an exception from a result value.

If an empty line is entered, or a line containing only spaces, tabs or embedded carriage returns or linefeeds, the prompt is repeated without incrementing the number. Such a line is not entered into the history.

A line which only contains a **TXR Lisp** comment (optional spaces, tabs or embedded carriage returns or linefeeds, followed by a semicolon), also causes the prompt to be repeated without incrementing the number. However, such a line **is** entered into the history.

The listener does not allow lines containing certain bad syntax to be submitted with Enter. If the buffer contains an expression with unbalanced parentheses or brackets, or unterminated literals, then Enter generates a newline character which is inserted into the buffer. In that situation, if that newline character is being added at the very end of the buffer, the listener flashes the exclamation mark character (!) two times to warn the user that line has not been submitted: no computation is taking place, and the listener is waiting for more input. It is possible to force the submission of an unbalanced line using the sequence Ctrl-X Ctrl-F.

12.3 Limitations

The interactive listener can only accept up to 4095 abstract characters of input in a single command line.

Though the edit buffer is referred as the "command line", it may contain multiline input. The carriage return characters which separate multiple lines count as one abstract character each, and are understood to occupy two display positions.

The command line must contain exactly one complete **TXR Lisp** expression, or a comment. Multiple expressions will not be evaluated.

In multiline mode, if the number of lines exceeds the number of lines of the terminal display, the editing experience is adversely affected in unspecified ways.

The screen updating logic in the listener is based on the assumption that the display terminal uses ANSI emulation. No other terminal emulation is supported. The `TERM` environment variable is ignored.

12.4 Ways to Quit

Pressing Ctrl-D in a completely empty command line terminates the listener. Another way to quit is to enter the `:quit` keyword symbol. When the form input into the listener consists of this symbol, the listener will terminate:

```
1> (+ 2 2)
4
2> :quit
os-shell $
```

Another way to terminate is to evaluate a call to the `exit` function. This method allows a termination status to be specified:

```
1> (exit 1)
os-shell $
```

However, if a **TXR** interactive session is terminated this way, it will not save the listener history.

Raising a fatal signal with the `raise` function is another way to quit:

```
1> (raise sig-abrt)
Aborted (core dumped)
os-shell $
```

The previous remark about not saving the listener history applies here also.

12.5 Interrupting Evaluation

Ctrl-C typed while editing a command line is interpreted as an editing command which causes that command line to be canceled. The listener prints the string `"** intr"` and repeats the same prompt.

If a command line is submitted for evaluation, the evaluation might take a long time or block for input. In these situations, typing Ctrl-C will issue an interrupt signal. The listener has installed a handler for this signal which generates an exception of type `error` which is caught by the listener. The exception's message is the string `"intr"` so that the listener ends up printing `"intr **"` like in the case of the Ctrl-C editing command. In this situation, though, a new command-line prompt is issued with an incremented number, and the exception is recorded as a value.

12.6 Listener Variables

12.6.1 Variables `*0`, `*1`, `*2`, ..., `*99`

Description:

The listener provides useful variables which allow commands to reference the results of previous commands. As noted previously, the commands are enumerated with an incrementing number. Each command's number, modulo 100, corresponds to one of the variables `*0`, `*1`, `*2`, ..., `*99`. Thus, up to the previous hundred results can be referenced:

```
...
99> (+ 2 2)  ;; stored in *99
4
100> (* 3 2) ;; stored in *0
6
101> (+ *99 *0) ;; i.e. (+ 4 6)
10
```

12.6.2 Symbol macros `*-1`, `*-2`, ..., `*-20`

The listener provides small number of symbol macros for referencing the results of previous commands in a relative. The macro `*-1` refers to the value of the immediately previous command. The macro `*-2` refers to the value of the command before that one and so on.

Note: each of these macros expands to a reference to the `*r` vector, according to the following pattern:

```
*-1  -->  [*r (mod (- *v 1) 100)]
*-2  -->  [*r (mod (- *v 2) 100)]
...
*-20 -->  [*r (mod (- *v 20) 100)]
```

12.6.3 Variable `*n`

Description:

The listener variable `*n` evaluates to the current command-line number: the number of the command in which the variable occurs:

```
5> *n
5
6> (* 2 *n)
12
```

12.6.4 Variable `*v`

Description:

The listener variable `*v` evaluates to the current variable number: the command number modulo 100:

```
103> *v
3
104> *v
4
```

12.6.5 Variable `*r`

Description:

The listener variable `*r` evaluates to a hash table which associates variable numbers with command results:

```
213> 42
42
214> [*r 13]
42
```

The result hash allows relative addressing. For instance the expression `[*r (mod (pred *v) 100)]` refers to the result of the previous command.

12.7 Exceptions

The interactive listener catches all exceptions. Each caught exception is associated with the command's variable number, and stored as a value in the appropriate listener variable as well as the `*r` result hash. Exceptions are turned into values by creating a cons cell whose `car` is the exception symbol and whose `cdr` holds the exception's arguments.

For each caught exception, a message is printed beginning with the sequence `"** "`. Exactly how the message appears depends on the type and content of the exception.

12.8 Editing

The following sections describe the interactive editing commands available in the listener.

Terminals can often be configured with different choices of cursor shape: such as a block-shaped cursor, an underline cursor or a vertical line or "I-beam" cursor. In the following sections, the phrase "character under the cursor" refers to the character that is currently covered by a block cursor, underlined by an underline cursor, or that is immediately to the right of an I-beam cursor.

12.8.1 Move Left and Right

Moving within the line is achieved using the left and right arrow keys [←] and [→]. In addition, [Ctrl-B] ("back") and [Ctrl-F] ("forward") perform this movement.

12.8.2 Jump to Beginning and End of Line

The [Ctrl-A] command moves to the beginning of the line. ("A" is the beginning of the alphabet). The [Ctrl-E] ("end") command jumps to the end of the line, such that the last character of the line is to the left of the cursor position. On terminals which have the Home and End keys, these may also be used instead of

Ctrl-A and Ctrl-E.

In line mode, these commands move the cursor to the beginning or end of the edit buffer.

In multiline mode, if the cursor is not already at the beginning of a physical line, then Ctrl-A moves it to the first character of the physical line. Otherwise, Ctrl-A moves the cursor to the beginning of the edit buffer.

Similarly, in multiline mode, if the cursor not already at the end of a physical line, Ctrl-E moves it there. Otherwise, the cursor moves to the end of the edit buffer.

12.8.3 Jump to Matching Parenthesis

If the cursor is on an opening or closing parenthesis, brace or bracket, the Ctrl-] command tries to jump to the matching character. The logic for finding the matching character is identical to that of the Parenthesis Matching feature. If no matching character is found, then no movement takes place.

If the cursor is not on an opening or closing parenthesis, brace or bracket, then the closest such character is found. The cursor is moved to that character and then an attempt is made to jump to the matching one from that new position.

If the cursor is equidistant to two such characters, then one of them is chosen as follows. If the two characters are oriented in the same way (both are opening and closing), then that one is chosen whose convex side faces the cursor position. Thus, effectively, an inner enclosure is favored over an outer one. Otherwise, if the two characters have opposite orientation (one is opening and the other closing), then the one which is to the right of the cursor position is chosen.

Note: the Ctrl-] character can be produced on some terminals using Ctrl-5 (using the keyboard home row 5, not the numeric keypad 5). This the same key which produces the % character when Shift is used. The % character is used in the Vi editor for parenthesis matching.

12.8.4 Character Swap

The Ctrl-T (twiddle) command exchanges the character under the cursor with the previous character.

12.8.5 Delete Character Left

The Backspace key erases the character to the left of the cursor, and moves the cursor to the position which that character occupied.

It doesn't matter whether this key generates ASCII characters 8 (BS) or 127 (DEL): either one is acceptable. The Ctrl-H command also performs the same action, since it corresponds to ASCII BS.

12.8.6 Delete Character Right

The Ctrl-D command deletes the character under the cursor, if the cursor is block-shaped, or to the right of the cursor if the cursor is an I-beam. the cursor maintains its current character position relative to the start of the line. In multiline mode, if Ctrl-D is at the end of a line that is not the last line, it deletes the newline character, causing the following line to be joined to the end of the current line. If the cursor is at the end of the buffer, then Ctrl-D does nothing, except if the buffer is completely empty, in which case it is a quit indication. The Delete key, if available on the terminal, is a near synonym of Ctrl-D. It performs all the same functions, except that it does not act as a quit indication; Delete has no effect when the buffer is empty.

When a visual selection is in effect, then Ctrl-D and Del delete that selection, and copy it to the clipboard.

12.8.7 Delete Word Left

The Ctrl-W ("word") command deletes the word to the left of the cursor position. More precisely, this command first deletes any consecutive whitespace characters (spaces or tabs) to the left of the cursor. Then, it deletes consecutive non-whitespace characters. Material under the cursor or to the right remains. The deleted material is copied into the clipboard.

12.8.8 Delete to Beginning of Line

The Ctrl-U ("undo typing") command is a "super backspace" operation: it deletes all characters to the left of the cursor position. The cursor is moved to the leftmost position. In multiline mode, Ctrl-U deletes only to the beginning of the current physical line, not all the way to the first position of the buffer. Ctrl-U copies the deleted material into the clipboard.

12.8.9 Delete to End of Line

The Ctrl-K ("kill") command deletes the character under the cursor position and all subsequent characters. The cursor position doesn't change. In multiline mode, Ctrl-K deletes only until the end of the current physical line, not the entire buffer. The material deleted by Ctrl-K is copied into the clipboard.

12.8.10 Verbatim Character Insert

The Ctrl-V ("verbatim") command places the listener's input editor into a mode in which the next character is interpreted literally and inserted into the line, even if that character is a special character such as Enter, or a command character.

12.8.11 Verbatim Insert Mode

The two-character sequence Ctrl-X Ctrl-V ("extended verbatim", "super paste") enters into an verbatim insert mode useful for entry of free-form text. It is particularly useful in multiline mode. In this mode, almost every character is inserted verbatim, including Enter. The only commands recognized are: Ctrl-X, which terminates this mode, Backspace (whether that key generates ASCII BS or DEL) and arrow key navigation. Enter inserts a line break, which appears as such in multiline mode, or as `^M` in line mode.

12.8.12 Delete Current Line

The Ctrl-X Ctrl-K command sequence may be used in multiline mode to delete the entire physical line under the cursor. Any lines below that line move up to close the gap. In line mode, the command has no effect, other than canceling select mode. The deleted line, including the terminating newline character, if it has one, is copied into the clipboard.

12.8.13 History Recall

By default, the most recent 500 lines submitted to the interactive listener are remembered in a history. This history is available for recall, making it convenient to repair mistakes, or compose new lines which are based on previous lines. Note that the history suppresses consecutive, duplicate lines. The number of lines retained may be customized using the `*listener-hist-len*` variable.

If the ↑ key is used while editing a line, the contents of the line are placed into a temporary save area. The line display is then updated to show the most recent line of history. Using ↑ will recall successively less recent lines.

The ↓ key navigates in the opposite direction: from older lines to newer lines. When ↓ is invoked on the most recent history line, then the current line is restored from the temporary save area.

Instead of ↑ and ↓, the commands Ctrl-P ("previous") and Ctrl-N ("next") may be used.

If the Enter key is pressed while a recalled history line is showing, then that line will be submitted as if it were a newly composed line. The originally edited line which had been placed in the save area is discarded.

When a recalled line is showing, it may be edited. There are two important behaviors to note here. If a recalled history line is edited, and then ↑ or ↓ or a navigation command is used to show a different history line, or to restore the original current line, then the edit is made permanent: the edited line replaces its original version in the same position in the history. This feature allows corrections to be made to the history.

The edit is recorded in the line's undo history as a single change; if the edited line is visited again, then a single Ctrl-O command will revert all the edits that were made.

However, if a recalled line is edited and submitted without navigating to another line, then it is submitted as a newly composed line, without replacing the original in the history.

Each submitted line is entered into the history, if it is different from the most recent line already in history. This is true whether it is a freshly composed line, a recalled history line, or an edited history line.

12.8.14 History Search

It is possible to search backwards through the history interactively for a line containing a substring. The Ctrl-R command is used to initiate search. The command prompt is replaced with the prefix `search:` next to which a pair of empty square brackets appears, indicating that the listener is in search mode. The square brackets are the search box, enclosing the search text, which is initially empty.

In search mode, characters may be typed. They accumulate inside the search box, and constitute the string to search for. The listener instantly navigates to the most recent line which contains a substring match for the search string, and places the cursor on the first character of the match. Control characters entered directly are ignored. The Ctrl-V command be used to add a character verbatim, as in edit mode.

To remove characters from the search box, Backspace can be used. The search is not repeated with the shortened search text: the same line continues to show until a character is added, at which point a new search is issued.

Search mode has a "home position": a starting point for searches. The initial home position is whatever line of history is selected when search mode is initiated. Searches work backward in history from that line. If search text is edited by deleting characters and then adding new ones, the new search proceeds from the home position.

The Ctrl-R command can be used in search mode. It registers the currently showing line as the new home position, and then repeats the search using the existing search text backwards from the new position. If the search text is empty, Ctrl-R has no effect.

The Ctrl-C command leaves search mode at any time and causes the listener to resume editing the original input at the original character position. The Enter key accepts the result of a search and submits it as if it were a newly composed line.

Navigation and editing keys may be used in search mode. A navigation or editing key immediately cancels search mode, and is processed in edit mode, using whatever line was located by the search, at the matching character position.

The Ctrl-L (Clear Screen and Refresh), as well as Ctrl-Z (Suspend to Background) commands are available in search mode. Their effects takes place without leaving search mode.

Navigating to a history line manually using ↑ or ↓ (or Ctrl-P and Ctrl-N) has the same net effect same as locating that line using Ctrl-R search.

12.8.15 Submit and Stay in History

Normally when the Enter key is used on a recalled history line, the next time the listener is reentered, it jumps back to the newest history position where a new line is about to be composed.

The alternative command sequence Ctrl-X Enter provides a useful alternative behavior. After the submitted line is processed, the listener doesn't jump to the newest history position. Instead, it stays in the history, advancing forward by one position to the successor of the submitted line.

Ctrl-X Enter can be used to conveniently submit a range of lines from the history, one by one, in their original order.

12.8.16 Insert Previous Word

The equivalent command sequences Ctrl-X w and Ctrl-X Ctrl-W insert a word from the previous line at the cursor position. A word is defined as a sequence of non-whitespace characters, separated from other words by whitespace. By default, the last word of the previous line is inserted. Between the Ctrl-X and the following Ctrl-W or w, a decimal number can be entered. The number 1 specifies that the last word is to be inserted, 2 specifies the second last word, 3 the third word from the right and so on. Only the most recent three decimal digits are retained, so the number can range from 0 to 999. A value of 0, or a value which exceeds the number of words causes the Ctrl-W or w to do nothing. Note that "previous line" means relative to the current location in the history. If the 42nd most recent history line is currently recalled, this command takes material from the 43rd history line.

12.8.17 Insert Previous Atom

The equivalent command sequences Ctrl-X a and Ctrl-X Ctrl-A insert an atom from the previous line at the cursor position. A line only makes atoms available if it expresses a valid **TXR** form, free of syntax errors. A line containing only whitespace or a comment makes no atoms available. For the purposes of this editing feature, an atom is defined as the printed representation of a Lisp atom taken from the Lisp form specified in the previous line. The line is flattened into atoms as if by the `flatcar` function. By default, the last atom is extracted. A numeric argument typed between the Ctrl-X and Ctrl-A or a can be used to select a atoms by position from the end. The number 1 specifies the last atom, 2 the second last and so on. Only the most recent three decimal digits are retained, so the number can range from 0 to 999. A value of 0, or a value which exceeds the number of words causes the Ctrl-A or a to do nothing. Note that "previous line" has the same meaning as for the Ctrl-X Ctrl-W (insert previous word) command.

12.8.18 Insert Previous Line

The command sequences Ctrl-X Ctrl-R ("repeat") and Ctrl-X r, which are equivalent, insert an entire line of history into the current buffer. By default, the previous line is inserted. A less recent line can be selected by typing a numeric argument between the Ctrl-X and the Ctrl-R or r. The immediately previous history line is numbered 1, the one before it 2 and so on. If this command is used during history navigation, it references previous lines relative to the currently recalled history line.

12.8.19 Symbolic Completion

If the Tab key is pressed while editing a line, it is interpreted as a request for completion. There is a second completion command: the sequence Ctrl-X Tab.

When completion is invoked with Tab or Ctrl-X Tab, the listener looks at a few of the trailing characters to the left of the cursor position to determine the applicable list of completions. Completions are

determined from among the **TXR Lisp** symbols which have global variable, function, macro and symbolic macro bindings, as well as the static and instance slots of structures. Symbols which have operator bindings are also taken into consideration. If a package-qualified symbol is completed, then completion is restricted to that package. Keyword symbol completion is restricted to the contents of the keyword package. The namespaces which are searched for symbols are restricted according to preceding character syntax. For instance if the characters `.(` or `.[` immediately precede the prefix, then only those symbols are considered which are methods: that is, each is the static slot of at least one structure, in which that static slots holds a function.

The difference between Tab and Ctrl-X Tab is that Tab completion looks only for prefix matches among the eligible identifiers. Thus it is a pure completion in the sense that it suggests additional material that may follow what has been typed. If the buffer contains `(list` it will only suggest completions which can be endings for `list` such as `list*`, `listp`, and `list-str`. It will not suggest identifiers which rewrite the `list` prefix. By contrast, the Ctrl-X Tab completion suggests not only pure completions but also alternatives to the partial identifier, by looking for substring matches. For instance `copy-list` is a possible completion for `list`, as is `proper-list-p`.

If no completions are found, then the BEL character is sent to the terminal to generate a beep or a visual alert indication. The listener returns to editing mode.

If completions are found, listener enters into completion selection mode. The first available completion is placed into the line as if it had been typed in. The other completions may be viewed one by one using the Tab key. (Note that the Ctrl-X is not used, only Tab, even if completion mode had been entered via Ctrl-X Tab). When the completions are exhausted, the original uncompleted line is shown again, and Tab can continue to be used to cycle through the completions again. In completion mode, the Ctrl-C character acts as a command to cancel completion mode and return to editing the original uncompleted line. Any other input character causes the listener to keep the currently shown completion, and return to edit mode, where that character is immediately processed as if it had been typed in edit mode.

12.8.20 Edit with External Editor

The two character command Ctrl-X Ctrl-E launches an external editor to edit the current command line. The command line is stored in a temporary file first, and the editor is invoked on this file. When the editor terminates, the file is read into the editing buffer.

The editor is determined from the `EDITOR` environment variable. If this variable is unset or empty, the command does nothing.

The temporary file is created in the home directory, if that can be determined. Otherwise it is created in the current working directory. If the creation of the file fails, then the command silently returns to edit mode. The home directory is determined from the `HOME` environment variable in POSIX environments. On MS Windows, the `USERPROFILE` variable is probed for the user's directory.

If the command line contains embedded carriage returns (which denote line breaks in multiline mode) these are replaced with newline characters when written out to the file. Conversely, when the edited file is read back, its newlines are converted to carriage returns, so that multiline content is handled properly. (See the following section, Multiline Mode.)

12.8.21 Undo Editing

The listener provides an undo feature. The Ctrl-O command ("old", "oops") restores the edit buffer contents and cursor position to a previous state.

There is a single undo history which records up the 200 most recent edit states. However, the states are associated with history lines, so that it appears that each line has its own, independent undo history.

Undoing the edits in one line has no effect on the undo history of another line.

Undo also records edits for lines that have been canceled with Ctrl-C and are not entered into the history, making it possible to recall canceled lines.

The undo history is lost when **TXR** terminates.

Undo doesn't save and restore previous contents of the clipboard buffer.

There is no redo. When undo removes an edit to restore to a prior edit state, the removed edit is permanently discarded.

Note that if undo is invoked on a historic line, each undo step updates that history entry instantly to the restored state, not only the visible edit buffer. This is in contrast to the way new edits work. New edits are not committed to history until navigation takes place to a different history line.

Also note that when new edits are performed on a historic line and it is submitted with Enter without navigating to another line, the undo information for those edits is retained, and belongs to the newly submitted line. The historic line hasn't actually been modified, and so it has no new undo information. However, if a historic line is edited, and then navigation takes place to a different historic line, then the undo information is committed to that line, because the modifications to the line have been placed back in the history entry.

12.9 Visual Selection Mode

The interactive listener supports visual copy and paste operation. Text may be visually selected for copying into a clipboard or or for deletion. In visual selection mode, the actions of some editing commands are modified so that they act upon the selection instead of their usual target, or upon both the target and the selection.

12.9.1 Making a Selection

The Ctrl-S command enters into visual selection mode and marks the starting point of the selection, which is considered the position immediately to the left of the current character.

While in visual selection mode, it is possible to move around using the usual movement commands. The ending point of the selection tracks the movement.

The selected text is displayed in reverse video.

Typing Ctrl-S again while in visual selection mode cancels the mode.

Tab completion, history navigation, history search and editing in an external editor all cancel visual selection mode.

By default, the selection excludes the character which lies to the right of the rightmost endpoint. Thus, the selection simply consists of the text between these two positions, whether or not they are reversed. This style of selection pairs excellently with an I-beam style cursor, and has clear semantics. The endpoints are referenced to the positions between the characters, and everything between them is selected.

The selection behavior may be altered using the Boolean configuration variable `*listener-sel-inclusive-p*`. This variable is `nil` by default. If it is changed to true, then the selection includes the character to the right of the rightmost endpoint, if there is such a character within the current line. This style of selection pair well with a block-shaped cursor. It creates the apparent semantics that the endpoints of the selection are characters, rather than points between characters, and that these characters are included in the selection.

12.9.2 Selection Endpoint Toggle

In visual selection, the starting point of the selection remains fixed, while the ending point tracks the movement of the cursor. The Ctrl-^ command will exchange the two points. The effect is that the cursor jumps to the opposite end of the selection. That end is now the ending point which tracks the cursor movement.

12.9.3 Visual Copy

The Ctrl-Y command ("yank") copies the selected text into a clipboard buffer. The previous contents of the clipboard buffer, if any, are discarded.

Unlike the history, the clipboard buffer is not persisted. If **TXR** terminates, it is lost.

12.9.4 Visual Cut

If the Ctrl-D command is invoked while a selection is in effect, then instead of deleting the character under the cursor, it deletes the selection, and copies it to the clipboard. The Delete key has the same effect.

Ctrl-D and Del have no effect on the clipboard when visual selection is not in effect, and they operate on just one character.

12.9.5 Clipboard Paste

The Ctrl-Q command ("quote the clipboard") inserts text from the clipboard at the current cursor position. The cursor position is updated to be immediately after the inserted text. The clipboard text remains available for further pasting.

If nothing has been yet been copied to the clipboard in the current session, then this command has no effect.

12.9.6 Clipboard Swap Paste

The Ctrl-X Ctrl-Q command sequence ("exchange quote") exchanges the selected text with the contents of the clipboard. The selection is copied into the clipboard as if by Ctrl-Y and replaced by the previous contents of the clipboard.

If the clipboard has not yet been used in the current session,

If nothing has been yet been copied to the clipboard in the current session, then this command behaves like Ctrl-Y: text is yanked into the clipboard, but not deleted.

12.9.7 Visual Replace

In visual selection mode, an editing commands may be used which insert new text, or a character may be typed in order to insert it. When this happens, the selection is first deleted and visual mode is canceled. Then the insertion takes place and visual mode is canceled. The effect is that the newly inserted text replaces the selected text.

This applies to the Clipboard Paste Ctrl-Q command also. If a selection is effect when Ctrl-Q is invoked, the selected text is replaced with the clipboard buffer contents.

When a selection is replaced in this manner, the contents of the clipboard are unaffected.

12.9.8 Delete in Selection Mode

In visual mode, it is possible to issue commands which delete text.

One such command is Ctrl-D. Its special behavior in selection mode, Visual Cut, is described above.

The Backspace key and Ctrl-H also have a special behavior in select mode. If the cursor is at the rightmost endpoint of the selection, then these commands delete the selection and nothing else. If the cursor is at the leftmost endpoint of the selection, then these commands delete the selection, and take their usual effect of deleting a character also. In both cases, selection mode is canceled. The clipboard is not affected.

The Ctrl-W command for deleting the previous word, when used in visual selection mode, deletes the selection and cancels selection mode, and then deletes the word before the selection. Only the deleted selection is copied into the clipboard, not the deleted word.

All other deletion commands such as Ctrl-K simply cancel visual selection mode and take their usual effect.

12.10 Multiline Mode

The listener operates in one of two modes: line mode and multiline mode. This is determined by the special variable `*listener-multi-line-p*` whose default value is `t` (multiline mode). It is possible to toggle between line mode and multiline mode using the Ctrl-J command.

In line mode, all input given to a single prompt appears to be on a single line. When the line becomes longer than the screen width, it scrolls horizontally. In line mode, carriage return characters embedded in a line are displayed as `^M`.

In multiline mode, when the input exceeds the screen width, it simply wraps to take up additional lines rather than scrolling horizontally. Furthermore, multiline mode not only wraps long lines of input onto multiple lines of the display, but also supports true multiline input. In multiline mode, carriage return characters embedded in input are treated as line breaks rather than being rendered as `^M`.

Because carriage returns are not line terminators in text files, lines which contain embedded carriage returns are correctly saved into and retrieved from the persistent history file.

When Enter is typed in multiline mode, the listener tries to determine whether the current input, taken as a whole, is an incomplete expression which requires closing punctuation for elements like compound expressions and string literals.

If the input appears incomplete, then the Enter is inserted verbatim at the current cursor position, rather than signaling that the line is being submitted for evaluation. The Ctrl-X Enter command sequence also has this behavior.

12.11 Reading Forms Directly from the Terminal

In addition to multiline mode, the listener provides support for directly parsing input from the terminal, suitable for processing large amounts of pasted material.

If the `:read` keyword is entered into the listener, it will temporarily suspend interactive editing and allow the **TXR Lisp** parser to read directly from standard input. The reading stops when an error occurs, or EOF is indicated by entering Ctrl-D.

In direct parsing mode, each expression which is read is evaluated, but its value is not printed. However, the value of the last form evaluated is returned to the interactive listener, which prints the value and accepts it as if it as the result value of the `:read` command.

Note that none of the material read from the terminal is entered into the interactive history. Only the `:read` command which triggers this parsing mode appears in the history.

12.12 Clear Screen and Refresh

The Ctrl-L command clears the screen and redraws the line being edited. This is useful when the display is disturbed by the output of some background process, or serial line noise.

12.13 Suspend to Background

The Ctrl-Z ("Zzzz... (sleep)") command causes **TXR** to be placed into the background in a suspended, and control returned to the system shell.

Bringing the suspended **TXR** back into the foreground is achieved with a shell job-control command such as the `fg` command in GNU Bash.

When **TXR** is resumed, the interactive listener will redisplay the edited line and restore the previous cursor position.

Making full use of this feature requires a POSIX job control shell, in the sense that without job control support in the shell, there may not be a way to restore **TXR** into the terminal session's foreground, causing the user to lose interactive control over that **TXR** instance.

12.14 Editing Help

The Ctrl-X ? command shows a summary of commands, in a four-line display which temporarily replaces the editing area. The help text is divided into several pages. Ctrl-C dismisses the display, and returns to editing. The Ctrl-P, ← and ↑ keys return to the previous screen. The Ctrl-Z and Ctrl-L commands are available, having their usual meaning of suspending and refreshing the display. Any other key advances to the next screen. Advancing from the last screen, dismisses the display, and returns to editing. Navigating to the previous screen when the first screen is being shown also dismisses the display and returns to editing.

12.15 Print the Prompt

The `:prompt` command prints the current prompt, followed by a newline, without incrementing the prompt number. The `:p` command prints just the current prompt number, followed by a newline, without incrementing the number.

In plain mode, the `:prompt-on` command enables the printing of prompts. The full prompt is printed before reading each new expression. An abbreviated prompt is printed before reading the continuation lines of an incomplete expression. The printing of prompts is automatically enabled if the input device is an interactive terminal.

None of these prompt-related commands are entered into the history.

12.16 Plain Mode

When the input device isn't an interactive terminal, or if the `-n` or `--noninteractive` command-line operations are used when invoking **TXR**, the listener operates in *plain mode*. It reads input without providing any editing features: no completion, history recall, selection, or copy and paste. Only the line editing features provided by the operating system are available. Prompts appear if standard input is an interactive terminal, or if explicitly enabled. There is still an incrementing counter, and the numbered variables `*1`, `*2`, `...` for accessing evaluation results are established. Lines are still entered into the history, and the interactive profile is still processed, as usual.

Plain mode reads whole lines of input, yet recognizes multi-line expressions. Whenever a line of input is read which represents incomplete syntax, another line of input is read and appended to that line. This repeats until the accumulated input represents complete syntax, and is then processed as a unit.

Each unit of input is expected to represent a single expression, otherwise an error is diagnosed.

12.17 Interactive Profile File

Unless the `--noprofile` option has been used, when the listener starts up, it looks for file called `.txr_profile` in the user's home directory, as determined by the `HOME` environment variable in POSIX environments or the `USERPROFILE` environment variable on MS Windows. If that variable doesn't exist, no further attempt is made to locate this file.

If the file exists, it is subject to security checks. First, the `path-components-safe` is applied to its path name. The function validates that no component of the path name is a directory that is writable to another user, or a symbolic link that could be rewritten by another user. If that check passes, the file is then checked with the function `path-strictly-private-to-me-p` which requires that other users have no read or write permission. If the checks fail, then an error message is displayed and the file is not loaded.

If the file passes the security check, it is expected to be readable and to contain **TXR Lisp** forms, which are read and evaluated. Syntax errors encountered while reading the profile file are displayed on standard output, and any exceptions thrown that are derived from `error` are caught and displayed. The interactive listener starts in spite of these situations. Exceptions not derived from `error` will terminate the process.

The profile file is not read by noninteractive invocations of **TXR**: that is, when the `-i` option isn't present.

12.18 History Persistence

The history is maintained in a text file called `.txr_history` in the user's home directory. Whenever the interactive listener terminates, this file is updated with the history contents stored in the listener's memory. The next time the listener starts, it first reloads the history from this file, making the most recent `*listener-hist-len*` expressions of a previous session available for recall.

The history file is maintained in a way that is somewhat robust against the loss of history arising from the situation that a user manages multiple simultaneous **TXR** sessions. When a session terminates, it doesn't blindly overwrite the history file, which may have already been updated with new history produced by another session. Rather, it appends new entries to the history file. New entries are those that had not been previously read from the history file, but have been newly entered into the listener.

An effort is made to keep the history file trimmed to no more than twice the number of entries specified in `*listener-hist-len*`. The terminating session first makes a temporary copy of the existing history, which is trimmed to the most recent `*listener-hist-len*` entries. New entries are then appended to this temporary file. Finally, the actual history file is replaced with this temporary file by a `rename-path` a rename operation. This algorithm doesn't use locking, and is therefore not robust against the situation when a two or more multiple interactive **TXR** sessions belonging to the same user terminate at around the same time.

The home directory is determined from the contents of the `HOME` environment variable in POSIX environments or `USERPROFILE` on MS Windows. If this variable doesn't exist, or the user doesn't have permissions to write to this directory or to an existing history file in that directory, then the history isn't saved.

It is possible to save the history without terminating the interactive session, using the `:save` command. This saves the history in the manner described above. Each invocation of `:save` only adds to the history file new input since the most recent `:save` command.

When the history file is loaded, security checks take place, in exactly the same way that the `".txr_profile"` file is validated. First the path of the history file is checked using the function `path-components-safe`, which determines that no component of the path name can be subverted by another user, other than the superuser. If that check passes, then the file is checked using `path-strictly-`

`private-to-me-p` which requires that other users have no read or write permission. If the checks fail, then an error message is displayed and the history file is not loaded.

12.19 Parenthesis Matching

A feature of the listener is visual parenthesis matching in the form of a brief forward or backward jump of the cursor. This provides a hint to the programmer, helping to prevent avoid parenthesis balancing errors.

When any of the three closing characters `)`, `]` or `}` is inserted, the listener scans backward for the matching opening character. Likewise, if any of the three opening characters `(`, `[` or `{` is inserted in the middle of text, the listener scans forward for the matching closing character.

If the matching character is found, the cursor jumps to that character and then returns to the original position a brief moment later. If a new character is typed during the brief time delay, the delay is immediately canceled, so as not to hinder rapid typing.

This back-and-forth jump behavior also occurs when a character is erased using Backspace, and the cursor ends up immediately to the right of a parenthesis.

Note that the matching is unsophisticated; it doesn't observe the lexical conventions and syntax of the **TXR Lisp** programming language. For instance, a closing parenthesis outside a string literal may match match an opening one inside a string literal.

12.20 Listener Configuration Variables

The listener's behavior can be influenced through values of certain global variables. The settings can be made persistent by means of setting these variables in the interactive profile file.

12.20.1 Special variable `*listener-hist-len*`

Description:

> This special variable determines how many lines of history are retained by the listener. Changing this variable from within the listener has an instant effect. If the number is reduced from its current value, history lines are immediately discarded. The default value is 500.

12.20.2 Special variable `*listener-multi-line-p*`

Description:

> This is a Boolean variable which indicates whether the listener is in multiline mode. The default value is `nil`.
>
> Changing this variable from within the listener takes effect immediately for the next line of input.
>
> If multiline mode is toggled interactively from within the listener, the variable is updated to reflect the latest state. This happens when the command is submitted for evaluation.

12.20.3 Special variable `*listener-sel-inclusive-p*`

Description:

> This Boolean variable controls the behavior of visual selection. It is `nil` by default.
>
> A visual selection is determined by endpoints, which are abstract positions understood as being between characters. When a visual selection begins, it marks an endpoint immediately to the left

of a block-shaped cursor, or precisely at the in-between position of an I-beam cursor. The end of the visual selection is similarly determined from the ending cursor position. The selection consists of those characters which lie between these positions. This style of selection pairs well with an I-beam style cursor shape.

If the `*listener-sel-inclusive-p*` variable is set true, then the selection also includes one more character to the right of the rightmost endpoint, if there is such a character within the current line, giving rise to the appearance that the selection is determined by the starting and ending character, and includes them. This type of selection pairs well with a block-shaped cursor.

12.20.4 Special variable `*listener-pprint-p*`

Description:

This Boolean variable controls how the listener prints the results of evaluations. It is `nil` by default.

When the variable is `nil`, the evaluation result of each line entered into the listener is printed using the `prinl` function. Thus values are rendered in a machine-readable syntax, ensuing read/print consistency.

If the variable is set true, the evaluation result of each line is printed using the `pprinl` function.

12.20.5 Special variable `*listener-greedy-eval-p*`

Description:

The special variable `*listener-greedy-eval-p*` controls whether or not a "greedy evaluation" feature is enabled in the listener. The default value is `nil`, disabling the feature.

Greedy evaluation means that after the listener evaluates an expression successfully and prints its value, it then checks whether that value is an expression that may be further subject to nontrivial evaluation. If so, it evaluates that expression, and prints the resulting value. The process is then repeated with the resulting value. It keeps repeating until evaluation throws an error, or produces a self-evaluating object.

These additional evaluations are performed in such a way that all warnings are suppressed and all other exceptions are intercepted.

Greedy evaluation doesn't affect the state of the listener. Only the original expression is entered into the history. Only the value of the original expression is saved in the result hash or a numbered variable. The command-line number `*n` is incremented by one. The additional evaluations are only performed for the purpose of producing useful output. The evaluations may have side effects.

Example:

```
1> (set *listener-greedy-eval-p* t)
t
2> 'a
a
3> (defvar b 2)
b
2
4> (defvar c '(+ 2 2))
c
(+ 2 2)
```

```
4
5> (defvar d '(list '+ 2 2))
d
(list '+ 2 2)
(+ 2 2)
4
```

The `(defvar d ...)` form produces `d` symbol as its result value. That symbol has a variable binding as a result of that `defvar` and so evaluates; that evaluation produces `(list '+ 2 2)`, the contents of `d`. That object is a Lisp expression and is evaluated, producing `(+ 2 2)` and that is also an expression, which reduces to `4`. The object `4` is self-evaluating, and so the greedy evaluation process stops.

12.20.6 Special variable `*doc-url*`

Description:

The special variable `*doc-url*` holds a character string representing a web URL intended to point to the HTML version of this document. The initial value points to the publicly hosted document on the Internet. The user may change this to point to another location, such as a locally hosted copy of the document.

This variable is used by the `doc` function.

12.21 Listener-Related Functions

12.21.1 Function `doc`

Syntax:

```
(doc [symbol])
```

Description:

The `doc` function provides help for the library symbol *`symbol`*. If information about *`symbol`* is available in the HTML version of this document, and is indexed, then this function causes that document to be opened using a web browser, such that the browser navigates to the appropriate section of the manual.

If the *`symbol`* argument is omitted, then the document is opened without navigating to a particular section.

The base URL for the document is configured by the `*doc-url*` variable.

If *`symbol`* is successfully found, or else not specified, and `doc` successfully invokes the URL-opening mechanism, it returns `t`. Otherwise, it throws an error exception.

The web browser is invoked using a system-dependent strategy. On MS Windows, the `ShellExecuteW` function is relied upon to open the URL.

On other platforms, if the `BROWSER` environment variable exists and is nonempty, its value is assumed to indicate the name or path of the web-browsing program which can accept the URL as an argument. If this variable doesn't exist or is empty, then `doc` searches for a system-dependent URL-opening utility, such as `xdg-open`. If this utility is not found, then `doc` falls back to searching for a browser using one of several names. If no URL-opening mechanism is identified using the above strategies, an error exception is thrown. However, if the mechanism is identified, but does not successfully dispatch the URL to a browser, there is no requirement to throw an error

exception. It may appear that the `doc` function returns `t` but has no effect.

12.21.2 Function `quip`

Syntax:

```
(quip)
```

Description:

The `quip` function returns a randomly selected string containing a humorous quip, quote or witticism. The following code may be added to `.txr_profile` to produce a random quip on startup:

```
(put-line (quip))
```

The `quip` function was introduced in **TXR** 244. If the `.txr_profile` is used with installations of older **TXR** versions, it is recommended to use the following, to avoid calling the undefined function, as well as to prevent a warning:

```
(if (fboundp 'quip)
  (put-line (quip))
  (defun quip ()))
```

13 SETUID/SETGID OPERATION

On platforms with the Unix filesystem and process security model, **TXR** has support for executing setuid/setgid scripts, even on platforms whose operating system kernel does not honor the setuid/setgid bit on hash-bang scripts. On these systems, taking advantage of the feature requires **TXR** to be installed as a setuid/setgid executable. For this reason, **TXR** is aware when it is executed setuid and takes care to manage privileges. The following description about the handling of setuid applies to the parallel handling of setgid also.

When **TXR** starts, early in its execution it determines whether or not is is executing setuid. If so, it temporarily drops privileges, as a precaution. This is done before processing the command-line arguments. When **TXR** determines that it is executing a setuid script (a file marked executable to its owner and attributed with the set-user-ID bit), it then attempts to impersonate the owner of the script file by changing to effective user ID to that owner just before executing the file. It retains the real and saved user ID. If the attempt to assume that user ID is unsuccessful, then **TXR** permanently drops setuid privileges before executing the script. Likewise, before executing any code other than a setuid script, **TXR** also drops privileges.

TXR tries to honor and implement the setuid permissions on a script whether or not it is running setuid. When not running setuid, it nevertheless tries to change its effective user ID to that of the owner of the setuid script. This will succeed if it has sufficient permissions to do so.

To rephrase: in order for **TXR** to execute a file which is setuid root, it has to be running with a root effective user ID somehow. In order to execute a file which is setuid to a non-root user, **TXR** has to be running effectively as root or else as that user. It doesn't matter whether these privileges are achieved effectively using the setuid mechanism, or whether **TXR** is running with the required user ID as its real ID. However, if **TXR** is running setuid, it takes special care to temporarily drop the privileges as early as possible, and eventually to drop the privileges permanently before executing any code, other that the setuid script. If the setuid script cannot be executed with the privileges it calls for, **TXR** also drops privileges and executes it anyway, strictly as the real user who invoked the **TXR** executable.

What it means to drop privileges is to change the effective user ID and the saved user ID to be equal to the real user ID. On platforms where the `setresuid` function is available, **TXR** uses that function to drop

privileges. On platforms where `setresuid` is not available, **TXR** tries to drop privileges using the C language function call `setuid(r)`, where `r` is the previously noted real user ID obtained from `getuid()`. On some platforms, this only works for dropping root privileges: it overwrites the real and saved ID only if the caller is effectively root. On those platforms, this approach does not drop non-root privileges. **TXR** tries to detect whether this approach worked by evaluating the C language expression `seteuid(e)`, where `e` is the previously noted effective user ID. In other words, it attempts to regain the dropped privilege by recovering the previous effective ID. If this attempt succeeds, **TXR** immediately aborts. Dropping setgid privileges is similar. Where `setresgid` is available it is used, otherwise an attempt is made with `setegid(r)` where `r` is the previously noted real group ID. Then a test using `setegid(e)` is performed using the original effective group ID as `e`. This is done after dropping any setuid root user ID privilege which would allow such a test to succeed.

If **TXR** is running both setuid and setgid, and execute a script which is setuid only, it will still drop group privileges, and vice versa: if it executed a setgid script, it will drop user privileges. For instance, if a root-owned **TXR** runs a setgid script which is owned by user `10` and group-owned by group `20`, that script will run with an effective group ID of 20. The effective user ID will be that of the user who invoked the script: **TXR** will drop the root privilege to the original real ID of the user, and while for the setgid operation, it will change to the group ID of the script.

The setuid/setgid privilege machinery in **TXR** does not manipulate the list of supplementary ("ancillary", in the language of POSIX) group IDs. It is unnecessary for security because the list does not change while running with setuid privilege. No group IDs are added to the list which need to be retracted when privileges are dropped. The supplementary groups also persist across the execution of a setuid/setgid script.

14 STANDALONE APPLICATION SUPPORT

The **TXR** executable image supports a general mechanism by means of which a custom program can be packaged as an apparent standalone executable.

14.1 The Internal Argument String

The **TXR** executable contains a 128 byte data area preceded by the seven-byte ASCII character sequence `"@(txr):"`. The 128 byte data area which follows this identifying prefix represents a null-terminated UTF-8 string. In the stock executable, this area is filled with null bytes.

If the **TXR** executable is edited such that this area is replaced with a nonempty, null-terminated UTF-8 string, the program will, for the purposes of command-line-argument processing, treat this string as if it were the one and only command-line argument. (The original command-line arguments are still retained in the `*args*` and `*args-full*` variables).

The function `save-exe` creates a copy of the **TXR** executable with a custom internal argument.

Example:

> Suppose that **TXR** is copied to an executable in the same directory called `myapp` (or `myapp.exe` on an operating system which requires the `.exe` suffix). Also suppose that in the same directory, there exists a file called `myscript.tl`.
>
> This `myapp` executable can then be edited so that the data area which follows the `@(txr):` bytes contains the following string:
>
> ```
> --args|-e|(load (path-cat (dir-name txr-exe-path) "main.tl"))
> ```
>
> When the `myapp` executable is invoked, it will process the above string as a single command-line argument, causing the `main.tl` **TXR Lisp** source file to be loaded. Any arguments passed to

myapp are ignored and available to main.tl via the *args* variable.

14.2 Deployment Directory Structure

The **TXR** executable may require library files, depending on the functionality invoked by the program code. Library files are located relative to the installation directory, called the *sysroot*. The executable tries to dynamically determine the sysroot from its own location, according to the following directory structure. The executable may be renamed, it need not be called txr:

```
/path/to/sysroot/bin/txr
              .../share/txr/stdlib/cadr.tl
                        .../stdlib/cadr.tlo
                        .../stdlib/except.tl
                                  ...
              .../share/txr/lib/...
```

The above structure is assumed if the executable finds itself in a directory named "bin".

Otherwise, if the executable finds itself in a directory not named "bin", the following structure is expected:

```
/path/to/installation/txr
                  .../stdlib/cadr.tl
                  .../stdlib/cadr.tlo
                  .../stdlib/except.tl
                            ...
                  .../lib/...
```

The "lib/" directory shown above is for third-party libraries. This is the directory indicated in the default value of the *load-search-dirs* special variable. The directory is not required to exist.

Note that this structure had changed starting in **TXR** 264. Older versions of **TXR**, when the executable is not in a directory named "bin", expect the following structure:

```
/path/to/installation/txr
                  .../share/txr/stdlib/cadr.tl
                  .../share/txr/stdlib/cadr.tlo
                  .../share/txr/stdlib/except.tl
                                        ...
```

When a custom application is deployed using a possibly renamed txr executable, one of the above structures should be observed: either the sysroot with a bin subdirectory where the executable is located, on the same level with the share directory, or else the second structure in which the stdlib directory is a direct subdirectory of the executable directory. If one of these structures is not observed, the application may fail due to the failure of a library file to load.

If the executable discovers that its name ends in the suffix "lisp" (or else "lisp.exe" on the MS Windows platform) then the behavior is as if the --lisp command line option had been given. Similarly, if the executable finds that its name ends in "vm" (or "vm.exe" on MS Windows) it behaves as if the --compiled option had been given.

14.3 Function `save-exe`

Syntax:

```
(save-exe path arg-string)
```

Description:

The `save-exe` function produces an edited copy of the **TXR** executable at the specified *path*, inserting *arg-string* as the internal argument string.

In order for the copied executable to be useful, the required installation directory structure must be provided around it, as described in the previous section, Deployment Directory Structure.

The return value of `save-exe` is unspecified.

The `arg-string` should encode to 127 bytes of UTF-8 or less, or else it will be abruptly truncated, possibly in the middle of a UTF-8 sequence.

Example:

Create a copy of **TXR** called `myapp` which will load a file called `main.tl` that is located in the same directory.

```
(save-exe
  "myapp"
  "--args|-e|(load (path-cat (dir-name txr-exe-path) \
  \ \"main.tl\"))")
```

15 DEBUGGER

TXR had a simple, crude, built-in debugger, which was removed.

16 COMPATIBILITY

16.1 Overview

New **TXR** versions are usually intended to be backward-compatible with prior releases in the sense that documented features will continue to work in the same way. Due to new features, new versions of **TXR** will supply new behaviors where old versions of **TXR** would have produced an error, such as a syntax error. Though, strictly speaking, this means that something is working differently in a new version, replacing an error situation with functionality is usually not considered a deviation from backward-compatibility.

There is one notable deviation from this general requirement for backwards compatibility: the handling of compiled files. For pragmatic reasons, from time to time **TXR** may break backward compatibility, such that a newer version of **TXR** will not load compiled files produced by an older version. The files will have to be recompiled with the new **TXR**. More details are given in the section **Compiled File Compatibility** under the major section **LISP COMPILATION**. The rationale for not requiring backward compatibility support for older compiled files is that older files require the older implementation of the virtual machine which they target. In some cases the differences between the older virtual machine and new is so great that **TXR** would have to carry a whole separate virtual-machine implementation for the sake of the older files, which is a significant burden.

16.2 The `-C` compatibility option

When a change is introduced which is not backward compatible, **TXR**'s `-C` option can be used to request emulation of old behavior.

The option was introduced in **TXR** 98, and so the oldest **TXR** version which can be emulated is **TXR** 97.

Side effects occur in the processing of the option. If the option is specified multiple times, the behavior is unspecified.

16.3 Environment variable `TXR_COMPAT`

If the `TXR_COMPAT` environment variable exists, and its value is not en empty string, it must contain a decimal integer. Its value is taken by **TXR** as a request to emulate old behaviors, just like the value of the `-C` option.

If the variable has incorrect contents or an out-of-range value, **TXR** will print an error diagnostic and exit.

If both `-C` and the `TXR_COMPAT` environment variable are supplied, the behavior is unspecified.

16.4 Compatibility Version Values

The following version values which have a special meaning as arguments to the `-C` option, along with a description of what behaviors are affected. For each of these version values, the described behaviors are provided if `-C` is given an argument which is equal or lower. For instance `-C 103` selects the behaviors described below for version 105, but not those for 102.

283 In **TXR** 283 and older versions, the *`flags`* parameter of the `ftw` function defaults to zero, rather than `ftw-phys`.

275 In **TXR** 275 and older versions, the FFI type operator `align` can weaken the alignment of a type. The current behavior is that it can only increase the strictness of alignment, which mimics the `aligned` type attribute found in GNU C. For instance `(align 2 int)` will not have an effect, because 2 is lower than the alignment of `int`. The `pack` type operator must be used instead to specify any alignment, including lower. A compatibility value of 275 or lower restores the ability of `align` to specify weaker alignment.

273 In **TXR** 273 and older versions, `lazy-str-get-trailing-list` has a flaw, which causes it to produce an extra empty string. Because the `@(freeform)` directive in the pattern language is based on lazy strings, and depends on this function, it is affected by this issue. The extra empty string is produced because the materialized prefix of the lazy string is split on the terminator without regard for the fact that it ends in the terminator, producing an extra empty piece. For instance, if the terminator is `"\n"` the materialized prefix of the lazy string is `"foo\n"` and the remaining list of not-yet-materialized lazy string material is `("bar" "baz")`, then the returned list is `("foo" "" "bar" "baz")`, rather than `("foo" "bar" "baz")`. Whenever the lazy string's *`terminator`* is non-empty, this issue reproduces in almost all instances, because the materialized prefix, unless it is empty, is always terminated by the *`terminator`* and so the split always produces the extra empty string. This is not a rare edge case. Compatibility values of 273 and lower restore this behavior.

272 The compatibility version value 272 restores old behaviors in the pattern language with regard to the regex and function cases of positive match variables. **TXR** 273, several semantic improvements took place in this area, which can break existing code. Pattern variables of the form `@{`*`bident (fun [args ...])`*`}` can now invoke a vertical function against the full input, and the variable consequently to be bound to multiple lines. Previously this syntax invoked only horizontal functions or else vertical functions in a single-line horizontal mode. That behavior is restored by 272 or lower compatibility. Secondly, the function is now always invoked, whether or not the variable has a binding. The variable is then matched against the text spanned by the function to either give it a new binding or match the existing binding. The old behavior, restored by 272 or lower compatibility, is that the function is not invoked when the variable has a binding; the variable's value is instead used to match text. Lastly, a similar change took place in positive match regular expression variables of the `@{`*`bident /regex/`*`}` form. Prior to 273, when a variable

of this form has an existing binding, the regex is ignored, and the situation is treated as a match for the variable content. This old behavior is also restored.

265 Until **TXR** 265, the `with-resources` macro exhibited an undocumented behavior: the three-element binding expression (*var init cleanup*) immediately caused the `with-resources` form to terminate with a return value of `nil` if the *init* form returned `nil`. Neither the *cleanup* in the same expression, nor any subsequent binding expressions or the body of the construct, would be evaluated. Prior cleanup forms would be evaluated in reverse order, as documented. A compatibility value of 265 or less restores this behavior.

262 Selection 262 compatibility restores a wrong behavior which existed between versions 191 and 262 due to a regression. The wrong behavior is that the `defsymacro` operator macro-expanded the replacement form, instead of associating the macro symbol with the unexpanded form. This makes a crucial difference to symbol macros which rely on expansion-time effects, such as producing a different expansion each time they are used.

258 Selecting 258 or lower compatibility causes `abs-path-p` to behave like `portable-abs-path-p`.

257 Until **TXR** 257, the function `lexical-var-p` returned `t` for not only lexical variables, but also for locally bound special variables, which are not lexical. The behavior is restored if 257 or older compatibility is selected.

251 Until **TXR** 251, the syntax `obj.[fun arg]` was equivalent to `[obj.fun arg]`, providing little utility. A compatibility value of 251 or lower restores that behavior. The new behavior is that `obj.[fun arg]` is equivalent to `obj.[fun obj arg]`, with `obj` evaluated only once, performing method dispatch.

248 Until **TXR** 248, the `hash-revget` function defaulted to using `eql` equality for searching the hash table for matching values rather than the current `equal`. Also, until 248, the `@` token for denoting meta-expressions was treated with a low precedence relative to the range dot `..` token. This led to strange results, such as `@(a)..@(b)` parsing in a way equivalent to `@(rcons a @(b))` rather than `(rcons @(a) @(b))`. Not is that undesirable due to the lack of symmetry, it's also inconsistent with `@a..@b` denoting `(rcons @a @b)`. The latter is because in that case the `@` is handled as part of the symbol token as a token, and not as a separate operator. A compatibility value of 248 or lower restores the above old behaviors of `@` and `hash-revget`.

244 Until **TXR** 244, the `env-hash` function returned a new hash table each time it was called. The behavior is restored if 244 or older compatibility is selected.

243 Two mistakes in the pseudorandom number generator (PRNG) were discovered, affecting **TXR** 243 and older. Using this compatibility value, or lower, will restore the buggy behavior, allowing pseudorandom number sequences produced by those older versions can be reproduced. The PRNG is intended to be an implementation of the WELL512a PRNG described by Panneton and L'Ecuyer. The coding mistakes, however, resulted in the PRNG being an implementation of something other than WELL512a.

242 In **TXR** 242 and older, the instantiation of an object whose type inherits from the same supertype more than once resulted in duplicate execution of the supertype's initialization. This was a documented behavior. After 242, duplicate initialization is suppressed. For more information, see the section **Duplicate Supertypes**.**A**compatibility**value**of**242**or**lower**restores the duplicate initialization behavior.

237 Compatibility values of 237 or lower restore the destructive behavior of the `sort` and `shuffle` functions.

234 In **TXR** 234 and older versions, the exception throwing functions `throw` and `throwf` did not return, regardless of the exception type. All unhandled exceptions triggered internal handling leading to unwinding and termination. The current behavior is that only `error` exceptions lead to termination. When a non-error exception isn't intercepted by a catch or handler, the `throw` or `throwf` returns normally, yielding the value `nil`. If a compatibility value equal to or lower than

234 is requested, the old behavior occurs: all unhandled exceptions terminate.

231 Versions of **TXR** until 231 contained an undocumented feature: some library functions which are documented as having parameters that must be of string type were allowing the arguments to be symbols. For such symbolic arguments, the name of the symbol obtained from `symbol-name` was implicitly taken as the required string value. This behavior was removed: passing symbolic arguments to library function parameters documented as strings will cause an exception to the thrown. If a compatibility value of 231 or lower is specified, however, the tolerant behavior is restored.

227 In **TXR** 227 and older versions, the functions `carray-uint`, `carray-int`, `uint-carray` and `int-carray` had different names, namely `carray-unum`, `carray-num`, `unum-carray` and `num-carray`, respectively. If 227 or lower compatibility is selected, these functions become available under their old names in addition to their new names.

225 After **TXR** 225, the behavior of the `do` operator was adjusted. Previously, a form like `(do set x)` which contains no variable references like `@1`, `@2` or `@rest`, generated a function similar to `(lambda (. rest) (set x))`. This was contrary to documentation, which states that `(do set x)` should produce a variadic function which has one required argument, and which assigns that argument to the variable `x` when invoked. The current implementation is that `(do set x)` is equivalent to `(do set x @1)` which produces the documented behavior. If 225 or lower compatibility is selected, then the old behavior of `do` takes effect.

224 After **TXR** 224, the treatment of certain special structure functions has changed. Selecting 224 compatibility or lower restores that behavior. The specification given in the **Special Structure Functions** paragraph has always stated that special functions must be static slots, and that the behavior is unspecified if they are instance slots. The behavior of **TXR** 224 and earlier was that these functions worked anyway if they were instance slots; after **TXR** 224, they some special functions will no longer be recognized if bound to instance slots.

222 After **TXR** 222, the behavior of `:vars` in `@(collect)` was subject to an adjustment. Previously, if the collect body didn't bind any variables, and both required and optional variables were specified in `:vars`, it would still bind all of the optional ones to their default values. This was a poor behavior which violated the idea that `:vars` enforces an all-or-nothing binding discipline to keep the collected lists consistent. Selecting 222 compatibility or lower restores this behavior.

215 After **TXR** 215, the behavior of the `load` function changed with respect to its treatment of the `*load-path*` variable. In cases where `load` resolved the path by adding a suffix, `*load-path*` was bound to the unsuffixed name, which was a documented behavior. After **TXR** 215, also, the behavior of the `sub-str` function changed. When the arguments implicate the entire string, `sub-str` started just returning the original string, and not making a copy. The old behavior was to always make a copy. The above old behaviors of `load` and `sub-str` are restored if 215 or lower compatibility is requested. Note, however, that the restoration of the `sub-str` behavior in response to the compatibility option was only introduced in **TXR** 251. In **TXR** 249 and older, the compatibility value has no effect on the behavior of `sub-str`.

202 Up to **TXR** 202, the `logxor` function was incorrectly implemented, producing wrong results when both arguments are the same fixnum integer, or the same bignum object. The incorrect behavior is restored if 202 or earlier compatibility is requested. After 202, the behavior of the `print` function changed with regard to symbols in the keyword package. Regardless of the *`pretty-p`* flag, keywords are printed with the leading colon. Compatibility with 202 or earlier restores the behavior that when the *`pretty-p`* flag is true, symbols are printed without package prefixes.

199 After **TXR** 199, certain global variables that had been deprecated for a long time, and no longer documented, were removed. Requesting 199 or earlier compatibility restores those variables.

190 Until **TXR** 190, the `reset-struct` function neglected to perform `:postinit` initializations, and didn't invoke finalization on the structure object if an exception was thrown during reinitialization. Thus, contrary to documented requirements, reinitialization of a structure didn't behave

like fresh construction. Also, until **TXR** 190, macro parameter lists implemented the requirement that a `:` (colon keyword symbol) argument to an optional was treated as a missing argument, triggering argument-defaulting behavior. That requirement was removed; the colon symbol behaves as an ordinary value under destructuring with macro parameter lists. Moreover, until **TXR** 190, the `pub` symbol package didn't exist; the `*package*` variable was initialized to the user package and so symbols introduced by application code were interned in the same package as the **TXR Lisp** library. Until **TXR** 190, `defmacro` and `defsymacro` forms were evaluated immediately during macro expansion; in **TXR** 191 or later, this eager evaluation was abandoned. Unfortunately, this change introduced a regression, causing the replacement form of a `defsymacro` to be macro-expanded at the time that form is traversed by the expander, so that the macro is associated with the expanded version of that form. This is something which had been fixed in 137. It went unnoticed until much later, after the 262 release. All the above old behaviors are restored in compatibility with version 190 or earlier. Finally, one more change after **TXR** 190 that is controlled by the compatibility mechanism was a critical redesign of the requirements for the behavior of the `ldiff` function. Version 190 compatibility causes the `ldiff` symbol to refer to the old implementation of `ldiff`.

188 Until **TXR** 188, `equal`-based hash tables printed using the notation `#H((:equal-based ...) ...)` whereas `eql`-based hash tables simply omitted the `:equal-based` keyword. Changes were introduced in **TXR** 187 which gave rise to a read/print inconsistency with printing behavior. In **TXR** 189, further changes were introduced to fix this inconsistency: `equal`-based hash tables print without any keyword indicating equality, and `eql`-based hash tables print as `#H((:eql-based) ...)`. If 188 or compatibility is selected, hash tables are printed in the old way.

187 Until **TXR** 187, hash tables constructed by the `hash` function were based on `eql` equality by default; the `:equal-based` keyword argument had to be specified to override this default, and the `:eql-based` keyword didn't exist. Selecting 187 or lower compatibility restores the behavior of `eql` equality being default, and the `:eql-based` keyword being unrecognized. This affects all functions which implicitly rely on `hash`, those being : `uniq`, `unique`, and `group-by`. In spite of these changes, the printed representation of hash tables continues to use the `:equal-based` keyword to indicate hash tables based on `equal` and its absence to indicate `eql` equality. The new `:eql-based` keyword may be used in hash literals (unless 187 compatibility is in effect, in which case it is ignored).

184 A value of 184 or lower switches to the old implementation of the `op` and `do` macros which was replaced starting in **TXR** 185. Also, this has the effect of disabling the special recognition of meta-expressions and meta-symbols in the dot position of function calls, and the macro expansion of meta-symbols in quasiliterals. This is because the old `op` implementation implements these behaviors itself. The implication is that user code which binds custom macros to `sys:var` or `sys:expr` may be affected by 184 or lower compatibility.

185 A value of 185 or lower restores the old precedence of the double dot notation for expressing ranges, relative to the referencing dot. Until **TXR** 185, the expression `a.b..c.d` parsed as `(qref a (rcons b c) d)`. What is worse, it parsed this way even if written as `a.b .. c.d`. Starting in **TXR** 186, `..` has a lower precedence, producing the more useful and intuitive parse `(rcons (qref a b) (qref c d))`: in other words, the range with endpoints given by `a.b` and `c.d`.

183 A value of 183 or lower restores an inconsistent behavior in the `@(bind)` directive and other places in the **TXR** pattern language where binding takes place. Prior to version 184, a string-tree match was only tried in both directions when the left-hand side of a binding (the "pattern") was a variable. For non-variable pattern terms, such as Lisp expressions or atoms, the string-tree match was tried in one direction only: a string tree arising out of the pattern could match a string atom value on the right side. A string tree is a nested list structure whose leaves are strings: a list of

strings, a list of lists of strings, and so on, in any mixture. Concretely, before **TXR** 184, `@(bind "a" ("a" "b" "c"))` didn't match, but `@(bind ("a" "b" "c") "a")` did. However, if the variable `a` contained `"a"` then `@(bind a ("a" "b" "c"))` did match: an inconsistency.

177 A value of 177 or lower causes the emulation of a bug which was present in the `rng` awk macro. A range whose start and end condition matched on the same record failed to activate for that record, even though `rng` is inclusive. The behavior is incompatible with POSIX Awk.

174 A value of 174 or lower restores a previous behavior of variable substitution in the `output` directive and in quasiliterals in both the **TXR** pattern language and **TXR Lisp**. The behavior in question is the evaluation of the element indexing or range selection modifier, exemplified by `@{a[2]}`. The previous behavior was that if the variable is of any type other than list, it is converted to a string (unless it already is one). The indexing then applies to the string. If it is a list then the indexing or range selection applies to the original list value, prior to conversion to text. The current behavior is that indexing and range selection is applied to the original value if that value is any sequence type which satisfies the `seqp` function, otherwise to the string representation.

172 A value of 172 or lower restores a behavior of the **TXR** pattern matching language when matching a variable followed by a directive, such as `@a@(fun b)`. The old behavior is that the scan for a match for the directive takes place in an environment in which a binding for `a` has not yet been established. The new behavior is that the variable is always bound prior to the processing of the directive. During the search, it is bound to the range of text spanning between the starting position and the position being tried.

170 A value of 170 or lower disables the behavior that **TXR** scans standard input when no input sources are specified on the command line. Standard input must be requested explicitly using the `-` argument. This is how it was in all versions of **TXR** up to 170. Some programs may behave differently because of this. Specifically, programs which do not take any arguments, and do not select an input source using the `@(next)` directive, or suppress the use of an input source using `@(next nil)`, may now accidentally read from standard input. Until version 170, the functions `split`, `split*`, `partition` and `partition*` ignored negative indices in their *`index-list`* argument. The new behavior is that the length of the input sequence is added to any negative index values. The resulting values are then ignored if they are still negative.

165 A value of 165 restores the following behaviors, which changed starting in 166. There was a change in Lisp evaluation support of the **TXR** pattern language. Specifically, Lisp argument forms were not subject to expansion prior to evaluation in these directives: `output`, `mod`, `modlast`, `skip`, `fuzz`, `load`, `close`, `call`, `cat` and `next`.

161 Version 161 was the last version in which a bug existed in the `handle` macro. In spite of the documentation claiming that `handle` has the same syntax as `catch`, the clauses of `handle` were being passed the exception symbol as the leftmost argument, followed by the exception arguments. This convention is different from `catch` clauses which do not receive the exception symbol, only the arguments. The discrepancy was corrected by making `handle` behave like `catch`, as documented. Requesting compatibility with 161 or earlier restores the previous behavior of the `handle` macro.

156 After version 156, two behaviors changed in the in the macro expander for `caseq`, `caseql` and `casequal`: one outright bug was fixed, and one hitherto undocumented behavior was changed and specified in the documentation at the same time. Selecting a compatibility value of 156 or less restores the previous behaviors. The bug was that single-atom case keys were undergoing evaluation. For instance `(caseql x (a 0))` would arrange for the evaluation of `a` as a variable, rather than treating it as the symbol `a` itself. Though the compatibility mechanism restores the behavior, applications depending on the evaluating behavior should be changed to instead use `caseq*`, `caseql*` or `casequal`. A workaround for this bug for **TXR** versions 156 or older is to replace simple keys with a key list of length one, exemplified by a rewrite of the foregoing expression to `(caseql x ((a) 0))`. Here `a` is not evaluated. The undocumented behavior was that a matching clause which has no forms to be evaluated was producing a result value of `t`.

For example `(case 1 (1))` previously yielded `t`, but now yields `nil`, and this behavior is documented.

155 After version 155, the `tok-str` and `tok-where` functions changed semantics. Previously, these functions exhibited the flaw that under some conditions they extracted an empty token immediately following a nonempty token. This behavior was working as designed and documented, but the design was flawed, creating a major difficulty in simple tokenizing tasks when tokens may be empty strings. Requesting compatibility with version 155 or earlier restores the behavior.

154 After version 154, changes were introduced in the semantics of struct literals. Previously, the syntax `#S(abc x a y b)` denoted the construction of an instance of `abc` with `x a y b` as the constructor parameters, similarly to `(new abc x 'a y 'b)`. The new behavior is that `abc` is constructed using no parameters, as if by `(new abc)` and then the slot values are assigned. This means that the values specified in the literal override any manipulations of those slots by the type's user-defined `:postinit` handlers. Also, after 154, `print` methods are expected to take three arguments and are invoked for both pretty printing and regular machine-readable printing. Until 154, a struct's `print` methods was called only when that struct was being pretty-printed, and only with two arguments; ordinary printing side-stepped the method and rendered the standard `#S` syntax featuring all instance slots.

151 After version 151, changes were implemented to the way static slots work in **TXR Lisp** structs. Selecting compatibility with 151 restores most of the behaviors. Until 151, each structure type had its own instance of static slots whether they were newly defined or inherited. Under the new scheme, a derived struct shares one instance of each inherited static slot with its base type. Under the old scheme, a struct inherits the static initialization functions of its bases (the *`static-initfun`* argument passed in `make-struct-type`). These are invoked because they are relied upon by the `defstruct` macro to perform the initializations of all the inherited static slots. Under the new scheme, the static initialization functions are not inherited. Only the type's own *`static-initfun`* is invoked to initialize its newly defined static slots that it doesn't share with the parent. The inherited static slots simply preserve their current values they have in the base type; their values are untouched by the introduction of a derived type. The `static-slot-ensure` also changed semantics after version 151. The old behavior was problematic because it affected all static slots throughout the inheritance hierarchy matching the name passed in by argument. Since this function is the basis for redefining methods, its behavior broke the semantics of overriding. Selecting 151 compatibility only restores the behavior of this function and macros based on it like `defmeth`: in the situation when it introduces a new static slot into one or more struct types, in compatibility mode it introduces the slot separately into each type without sharing, and it recurses over the entire type hierarchy, storing *`new-val`* into all static slots which match *`name`*.

150 Until version 150, the `match-regex` function behaved in a different way from what was documented. Rather than returning the length of the match, it returned the index one past the last matching character. In the case when the starting position is zero, these values coincide; they are different if the match begins at some position inside the string. Compatibility with 150 restores the behavior. The `match-regst` function was also affected by this issue; however, since it returned nonsense result not corresponding to the matching text, it was repaired without backward compatibility. Also affected by version 150 compatibility are the `match-regex-right` and `match-regst-right` functions. These functions worked as documented; however, their specification changes after version 150 to a semantics which is more useful and less surprising to the programmer.

148 Up until version 148, the `:postinit` handlers specified in a `defstruct` were executed in derived-to-base order, opposite to the order of execution of `:init` handlers. Though described in terms of `defstruct` syntax and concepts, this is actually a change in how `make-struct-type` treats its *`postinitfun`* argument. Specifying 148 or earlier compatibility provides this old behavior. Also, until version 148, the `trim-str` function stripped leading and trailing white-space from a string consisting of not only spaces, tabs and newlines, but also carriage returns,

vertical tabs and form feeds.

145 In versions 144 and 145, **TXR** opened files in text mode on Cygwin, enabling conversion between CR-LF line endings and abstract newline characters. This behavior change was retracted, so that files on Cygwin are opened without specifying text mode, causing the streams to be effectively binary. The intended "Windows native" behavior of streams being text mode is instead provided in the Windows version of **TXR** by the Cygnal library.

143 Until version 143, the `stdlib` variable didn't include the trailing slash. The `makunbound` function semantics changed after version 143 to be more compatible with ANSI Common Lisp. Until 143, that function removed only the global binding, leaving the dynamic rebinding of a variable intact. The `defsymacro` operator neglected to remove the symbol's special variable mark, if the symbol was previously defined as a special variable. Also, until version 143 many more places in the **TXR** pattern language used bind expressions rather than Lisp expressions. The compatibility option restores these behaviors.

142 Until version 142, the **TXR** pattern language supported a prefix convention on data sources. Data sources beginning with the character `!` were treated as system command pipes, and data sources beginning with `$` indicated that a directory is to be scanned. This convention was recognized both for command-line arguments, the arguments of the `@(next)` directive, and of the `@(output)` directive, whether or not the argument was a literal or a computed value. This feature was dropped from the language after version 142. Also, until version 142, the `@(next)` directive recognized the name `"-"` as denoting standard input, and `@(output)` recognized it as standard output. These behaviors were also removed; versions after 142 recognize this convention only when it appears as a command-line argument. Lastly, until version 142, the `@(output)` directive evaluated the *`destination`* argument as an expression of the **TXR** pattern language, requiring `@` to be used to denote a Lisp expression. This is no longer required. All these old behaviors are provided if compatibility with 142 or earlier is requested.

139 After **TXR** 139, changes were implemented in the area of pseudorandom number generation. Compatibility with 139 brings back the previous seeding algorithm used by `make-random-state`, allowing the old pseudorandom sequences to be reproduced. This is only the case if the default value of 8 is used for the *`warmup-period`* argument of that function (which didn't exist in 139 or earlier versions).

138 After **TXR** 138, the variable name lookup rules in the **TXR** pattern language changed for greater utility and consistency. Compatibility with 138 or later restores the previous rules under which most accesses to a **TXR Lisp** variable from **TXR Lisp** require the `@` prefix denoting Lisp evaluation, but some do not.

137 Compatibility with **TXR** 137 restores the behavior of not expanding symbol macros in the dot position of a function call form. For instance if `x` is a symbol macro, in this compatibility mode it is not recognized in a form like `(list 1 2 . x)`. This preserves the behavior of code which depends on `x` in such a form to refer to a variable that is being otherwise shadowed by the symbol macro. **TXR** 137 compatibility also restores a particular behavior of the global and local macro defining operators `defsymacro` and `symacrolet`: in compatibility mode, these operators macro-expand the replacement forms of symbol macros at expansion time, and then bind the resulting expanded forms to their respective macro symbols. The forms are then potentially expanded again when the symbol macros are substituted. This wrong behavior was never implied by the documentation. The `with-slots` macro is also affected by this, because it is implemented in terms of `symacrolet`. Lastly, **TXR** 137 compatibility mode also restores another behavior of the dot position in function call forms: if the dot position of a function call form produces a sequence that is not a list, that sequence is converted to a list so that `(list . "abc")` produces `(#\a #\b #\c)`. After 137, no such treatment is applied to the value and the same form now yields `"abc"`.

136 A request for compatibility with **TXR** 136 or earlier restores the old behavior of the `if` directive, which in used to be a syntactic sugar for a `cases` directive with `require` at the top of each block. Though semantically well-defined and working as documented, the behavior was confusing,

since failed matching caused potential evaluation of multiple clauses, whereas programmers expect an if/elif/else ladder to select exactly one clause.

128 Compatibility with **TXR** 128 or earlier brings back the behavior that expressions in quasiliterals are evaluated according to **TXR** evaluation rules for quasiliterals which occur in the **TXR** pattern language. Similarly, expressions in `@(output)` blocks are treated **TXR** pattern language expressions.

127 In versions of **TXR** until 127, the functions `symbol-function`, `fboundp` and `fmakunbound` behaved similarly to their Common Lisp counterparts. See the Dialect Notes under these functions.

124 In **TXR** 124 and earlier versions, the `@(next)` directive didn't evaluate the *`source`* argument as a Lisp expression, but as a **TXR** pattern language expression. Lisp expressions thus had to be delimited by `@`. The current behavior is that the argument is treated as Lisp. If the compatibility option is set to 124 or lower, the old behavior is restored. However, even without the presence of the compatibility option, if the *`source`* argument is a meta-expression or meta-symbol (denotes by the `@` prefix in front of a compound expression or symbol, respectively) it is also treated in the old way. This latter behavior is obsolescent and will eventually disappear, and the compatibility option will be the only way to get the old behavior.

123 In **TXR** 123 and earlier, the variable initialization forms of a `for` or `for*` loop were evaluated outside of the scope of the implicit `nil` block. They are now inside the block. The compatibility option will restore the old behavior.

121 In **TXR** 121 and earlier versions, **TXR Lisp** expressions evaluated in the pattern language were placed in a lexical environment in which the pattern variables were visible as lexical variables. The meant that these variables could be directly captured in lexical closures. On the other hand, it meant that a Lisp function defined in a `@(do)` block could not access a variable established by a later `@(bind)`. It doesn't make sense for dynamically captured variables to be lexical, so the rule was changed. The backward compatibility switch will enable the old scoping behavior. Capturing the values of pattern variables in closures is possible indirectly under the new rule: simply bind new lexical variables with their values.

118 The `slot-p` function's name changed to `slotp` after 118. The compatibility option causes `slot-p` to be defined also.

117 The arguments of the `make-struct-type` acquired changed after version 117. 117 compatibility brings back the old interface.

114 **TXR** until version 114 reported parse errors in this format:

```
./txr: (file.txr:123): syntax error
```

The new format omits the program name prefix and parentheses.

Also, the `kill` function returned an integer, obtained from the return value of the underlying C function, rather than converting that value to a Boolean. The old behavior was not documented, and 114 compatibility restores it.

Lastly, prior to 115, random state objects were of type `*random-state*` (the same symbol as the special variable name) rather than of type `random-state`. This is a bug whose behavior is simulated by 114 compatibility.

113 Version 113 is the last version in which the `stat`, `lstat`, and `fstat` functions returned a property list rather than a structure. Requesting 113 compatibility restores the behavior of returning a

property list. However, the filesystem testing functions like `path-exists-p` will not work, because they rely on these functions returning a structure.

109 The optional trailing semicolon on hex and octal codes in the **TXR** pattern language was introduced in 110. The feature is disabled with 109 or lower compatibility, so that `@\x21;a` encodes `!;a` rather than the current behavior of encoding `!a`. Also, in 109 and earlier, newlines were allowed in word list literals and word list quasiliterals. They were treated as a word-separating space. A backslash-escaped newline, and all whitespace around it, was deleted just like in ordinary literals, and did not separate words. The old behavior is emulated.

107 Up through **TXR** 107, by accident, there was a function called `flip` as well as an operator by the same name. The function was renamed to `flipargs`. Version 107 compatibility or earlier provides the function under the original name also. Also, up until this version, **TXR** allowed functions and macros to be defined with the same names as built-in operators, and macros. Newer versions reject this as an error. Requesting compatibility to 107 or earlier suppresses the rejection, though without introducing any requirement that redefinition will work as expected.

105 Provides the behavior that the `open-file` function automatically marks a stream open on a TTY devices as a real-time stream (subject to the availability of the POSIX `isatty` function).

Also allows unrecognized backslash escape sequences in regular expression syntax to simply denote the escaped character literally, as was historically the case prior to **TXR** 106, so that `\z` for instance denotes `z`. As of **TXR** 106, these are diagnosed as errors.

102 Up to **TXR** 102, the `get-string` function did not close the stream. This old behavior is emulated.

101 Up to **TXR** 101, the `make-like` function incorrectly returned `nil` when converting the empty list `nil` to string type. This affects numerous generic sequence functions, causing their result to be `nil` instead of an empty string.

100 Up to **TXR** 100, the `split-str` function had an undocumented behavior. When the `sep` argument was an empty string, it split the string into individual characters as if by calling `list-str`. This behavior changed to the currently documented behavior starting in **TXR** 101. Also, the arguments of the `where` function, which introduced in **TXR** 91, were reversed starting in **TXR** 101.

99 Up to **TXR** 99, the substitution of TXR Lisp expressions in `@(output)` directives and in the quasistrings of the pattern language exhibited the buggy behavior that if the TXR Lisp expression produced a list, the list was rendered as a parenthesized representation, or the text `nil` in the empty list case. Moreover, in the `@(output)` case, the value of TXR Lisp expressions was not subject to filtering. Starting with **TXR** 100, these issues are fixed, making the behavior consistent with that of TXR Lisp quasiliterals.

97 Up to **TXR** 97, the error exception symbols such as `file-error` were named with underscores, as in `file_error`. These error symbols existed: `type_error`, `internal_error`, `numeric_error`, `range_error`, `query_error`, `file_error` and `process_error`.

16.5 Variables `txr-version` and `lib-version`

Description:

The `txr-version` variable gives the version of the **TXR** executable. Programs can express conditional variable based on detecting the version.

The `lib-version` variable gives the version of the installed library of **TXR** code accompanying the executable.

It is expected that these two variables have an identical value. Any discrepancy in their value indicates an installation whose library or **TXR** executable were upgraded independently. Should such a situation arise in any system and cause a problem, **TXR** programs can be defensively coded against it with the help of these variables.

Some features of the **TXR** library are built into the executable, whereas others are in the library directory. This aspect of library symbols isn't specified in this manual; knowing which of these two variables is relevant to a library feature requires familiarity with the implementation.

17 APPENDIX

17.1 A. NOTES ON EXOTIC REGULAR EXPRESSIONS

Users familiar with regular expressions may not be familiar with the complement and intersection operators, which are often absent from text processing tools that support regular expressions. The following remarks are offered in the hope that they may be of some use.

Equivalence to Sets

Regexp intersection is not essential; it may be obtained from complement and union as follows, since De Morgan's law applies to regular-expression algebra: `(R1)&(R2) = ~(~(R1)|~(R2))`. (The complement of the union of the complements of R1 and R2 constitutes the intersection.) This law works because the regular expression operators denote set operations in a straightforward way. A regular expression denotes a set of strings (a potentially infinite one) in a condensed way. The union of two regular expressions `R1|R2` denotes the union of the set of texts denoted by `R1` and that denoted by `R2`. Similarly `R1&R2` denotes a set intersection, and `~R` denotes a set complement. Thus algebraic laws that apply to set operations apply to regular expressions. It's useful to keep in mind this relationship between regular expressions and sets in understanding intersection and complement.

Given a finite set of strings, like the set `{ "abc", "def" }` which corresponds to the regular expression `(abc|def)`, the complement is the set which contains an infinite number of strings: it consists of all possible strings except `"abc"` and `"def"`. It includes the empty string, all strings of length 1, all strings of length 2, all strings of length 3 other than `"abc"` and `"def"`, all strings of length 4, etc. This means that a "harmless looking" expression like `~(abc|def)` can actually match arbitrarily long inputs.

Set Difference

How about matching only three-character-long strings other than `"abc"` or `"def"`? To express this, regex intersection can be used: these strings are the intersection of the set of all three-character strings, and the set of all strings which are not `"abc"` or `"def"`. The straightforward set-based reasoning leads us to this: `...&~(abc|def)`. This `A&~B` idiom is also called set difference, sometimes notated with a minus sign: `A-B` (which is not supported in **TXR** regular-expression syntax). Elements which are in the set `A`, but not `B`, are those elements which are in the intersection of `A` with the complement of `B`. This is similar to the arithmetic rule `A - B = A + -B`: subtraction is equivalent to addition of the additive inverse. Set difference is a useful tool: it enables us to write a positive match which captures a more general set than what is intended (but one whose regular expression is far simpler than a positive match for the exact set we want), then we can intersect this over-generalized set with the complemented set of another regular expression which matches the particulars that we wish excluded.

Expressiveness versus Power

It turns out that regular expressions which do not make use of the complement or intersection operators are just as powerful as expressions that do. That is to say, with or without these operators, regular expressions can match the same sets of strings (all regular languages). This means that for a given regular expression which uses intersection and complement, it is possible to find a regular expression which doesn't use these operators, yet matches the same set of strings. But, though they exist, such equivalent regular expressions are often much more complicated, which makes them difficult to design. Such expressions do not necessarily **express** what it is they match; they merely capture the equivalent set. They perform a job, without making it obvious what it is they do. The use of complement and intersection leads to natural ways of expressing many kinds of matching sets, which not only demonstrate the power to carry out an operation, but also easily express the concept.

Example: Matching C Language Comments

For instance, using complement, we can write a straightforward regular expression which matches C language comments. A C language comment is the digraph `/*`, followed by any string which does not contain the closing sequence `*/`, followed by that closing sequence. Examples of valid comments are `/**/`, `/* abc */` or `/***/`. But C comments do not nest (cannot contain comments), so that `/* /* nested */ */` actually consists of the comment `/* /* nested */`, which is followed by the trailing junk `*/`. Our simple characterization of the interior part of a C comment as a string which does not contain the terminating digraph makes use of the complement, and can be expressed using the complemented regular expression like this: `(~.*[*][/].*)`. That is to say, strings which contain `*/` are matched by the expression `.*[*][/].*`: zero or more arbitrary characters, followed by `*/`, followed by zero or more arbitrary characters. Therefore, the complement of this expression matches all other strings: those which do not contain `*/`. These strings make up the inside of a C comment between the `/*` and `*/`.

The equivalent simple regex is quite a bit more complicated. Without complement, we must somehow write a positive match for all strings such that we avoid matching `*/`. Obviously, sequences of characters other than `*` are included: `[^*]*`. Occurrences of `*` are also allowed, but only if followed by something other than a slash, so let's include this via union:

```
([^*]|[*][^/])*.
```

Alas, we already have a bug in this expression. The subexpression `[*][^/]` can match `**`, since a `*` is not a `/`. If the next character in the input is `/`, we missed a comment close. To fix the problem we revise to this:

```
([^*]|[*][^*/])*
```

(The interior of a C language comment is any mixture of zero or more non-asterisks, or digraphs consisting of an asterisk followed by something other than a slash or another asterisk). Oops, now we have a problem again. What if two asterisks occur in a comment? They are not matched by `[^*]`, and they are not matched by `[*][^*/]`. Actually, our regex must not simply match asterisk-non-asterisk digraphs, but rather sequences of one or more asterisks followed by a non-asterisk:

```
([^*]|[*]*[^*/])*
```

This is still not right, because, for instance, it fails to match the interior of a comment which is terminated by asterisks, including the simple test cases where the comment interior is nothing but asterisks. We have no provision in our expression for this case; the expression requires all runs of

asterisks to be followed by something which is not a slash or asterisk. The way to fix this is to add on a subexpression which optionally matches a run of zero or more interior asterisks before the comment close:

```
([^*]|[*]*[^*/])*[*]*
```

Thus the semi-final regular expression is

```
[/][*]([^*]|[*]*[^*/])*[*]*[*][/]
```

(Interpretation: a C comment is an interior string enclosed in `/* */`, where this interior part consists of a mixture of non-asterisk characters, as well as runs of asterisk characters which are terminated by a character other than a slash, except for possibly one rightmost run of asterisks which extends to the end of the interior, touching the comment close. Phew!) One final simplification is possible: the tail part `[*]*[*][/]` can be reduced to `[*]+[/]` such that the final run of asterisks is regarded as part of an extended comment terminator which consists of one or more asterisks followed by a slash. The regular expression works, but it's cryptic; to someone who has not developed it, it isn't obvious what it is intended to match. Working out complemented matching without complement support from the language is not impossible, but it may be difficult and error-prone, possibly requiring multiple iterations of trial-and-error development involving numerous test cases, resulting in an expression that doesn't have a straightforward relationship to the original idea.

The Non-Greedy Operator

The non-greedy operator `%` is actually defined in terms of a set difference, which is in turn based on intersection and complement. The uninteresting case `(R%)` where the right operand is empty reduces to `(R*)`: if there is no trailing context, the non-greedy operator matches `R` as far as possible, possibly to the end of the input, exactly like the greedy operator. The interesting case `(R%T)` is defined as a "syntactic sugar" which expands to the expression `((R*)&(~.*(T&.+).*))T` which means: match the longest string which is matched by `R*`, but which does not contain a non-empty match for `T`; then, match `T`. This is a useful and expressive notation. With it, we can write the regular expression for matching C language comments simply like this: `[/][*].%[*][/]` (match the opening sequence `/*`, then match a sequence of zero or more characters non-greedily, and then the closing sequence `*/`. With the non-greedy operator, we don't have to think about the interior of the comment as set of strings which excludes `*/`. Though the non-greedy operator appears expressive, its apparent simplicity may be deceptive. It looks as if it works "magically" by itself; "somehow" this `.%` part "knows" only to consume enough characters so that it doesn't swallow an occurrence of the trailing context. Care must be taken that the trailing context passed to the operator really is the correct text that should be excluded by the non-greedy match. For instance, take the expression `.%abc`. If you intend the trailing context to be merely `a`, you must be careful to write `(.%a)bc`. Otherwise, the trailing context is `abc`, and this means that the `.%` match will consume the longest string that does not contain `abc`, when in fact what was intended was to consume the longest string that does not contain `a`. The change in behavior of the `%` operator upon modifying the trailing context is not as intuitive as that of the * operator, because the trailing context is deeply involved in its logic.

On a related note, for single-character trailing contexts, it may be a good idea to use a complemented character class instead. That is to say, rather than `(.%a)bc`, consider `[^a]*abc`. The set of strings which don't contain the character a is adequately expressed by `[^a]*`.