The Nearly Complete Scheme48 1.3 Reference Manual

Taylor Campbell

First Edition

This manual is for Scheme48 version 1.3.

Table of Contents

1 Introduction

Scheme48 is an implementation of Scheme based on a byte-code virtual machine with design goals of simplicity and cleanliness. To briefly enumerate some interesting aspects of it, Scheme48 features:

- an advanced module system based on Jonathan Rees's W7 security kernel with well-integrated interaction between macros and modules;
- a virtual machine written in a dialect of Scheme itself, Pre-Scheme, for which a compiler is written with Scheme48;
- a sophisticated, user-level, preëmptive multithreading system with numerous high-level concurrency abstractions;
- a composable, lock-free shared-memory thread synchronization mechanism known as *optimistic concurrency*; and
- an advanced user environment that is well-integrated with the module and thread systems to facilitate very rapid development of software systems scaling from small to large and single-threaded to multi-threaded.

It was originally written by Jonathan Rees and Richard Kelsey in 1986 in response to the fact that so many Lisp implementations had started out simple and grown to be complex monsters of projects. It has been used in a number of research areas, including:

- mobile robots at Cornell [Donald 92];
- a multi-user collaboration system, sometimes known as a 'MUD' ('multi-user dungeon') or 'MUSE' ('multi-user simulation environment'), as well as general research in capability-based security [Museme; Rees 96]; and
- advanced distributed computing with higher-order mobile agents at NEC's Princeton research lab [Cejtin *et al.* 95].

The system is tied together in a modular fashion by a configuration language that permits quite easy mixing and matching of components, so much so that Scheme48 can be used essentially as its own OS, as it was in Cornell's mobile robots program, or just as easily within another, as the standard distribution is. The standard distribution is quite portable and needs only a 32-bit byte-addressed POSIX system.

The name 'Scheme48' commemorates the time it took Jonathan Rees and Richard Kelsey to originally write Scheme48 on August 6th & 7th, 1986: forty-eight hours. (It has been joked that the system has expanded to such a size now that it requires forty-eight hours to *read* the source.)

1.1 This manual

This manual begins in the form of an introduction to the usage of Scheme48, suitable for those new to the system, after which it is primarily a reference material, organized by subject. Included in the manual is also a complete reference manual for Pre-Scheme, a low-level dialect of Scheme for systems programming and in which the Scheme48 virtual machine is written; see Chapter 9 [Pre-Scheme], page 149.

This manual is, except for some sections pilfered and noted as such from the official but incomplete Scheme48 reference manual, solely the work of Taylor Campbell, on whom all

responsibility for the content of the manual lies. The authors of Scheme48 do not endorse this manual.

1.2 Acknowledgements

Thanks to Jonathan Rees and Richard Kelsey for having decided so many years ago to make a simple Scheme implementation with a clean design in the first place, and for having worked on it so hard for so many years (almost twenty!); to Martin Gasbichler and Mike Sperber, for having picked up Scheme48 in the past couple years when Richard and Jonathan were unable to work actively on it; to Jeremy Fincher for having asked numerous questions about Scheme48 as he gathered knowledge from which he intended to build an implementation of his own Lisp dialect, thereby inducing me to decide to write the manual in the first place; to Jorgen Schäfer, for having also asked so many questions, proofread various drafts, and made innumerable suggestions to the manual.

2 User environment

2.1 Running Scheme48

Scheme48 is run by invoking its virtual machine on a dumped heap image to resume a saved system state. The common case of invoking the default image, `scheme48.image`, which contains the usual command processor, run-time system, *&c.*, is what the `scheme48` script that is installed does. The actual virtual machine executable itself, `scheme48vm`, is typically not installed into an executable directory such as `/usr/local/bin/` on Unix, but in the Scheme48 library directory, which is, by default on Unix installations of Scheme48, `/usr/local/lib/`. However, both `scheme48` and `scheme48vm` share the following command-line options; the only difference is that `scheme48` has a default `-i` argument.

`-h` *heap-size*
: The size of Scheme48's heap, in cells. By default, the heap size is 3 megacells, or 12 megabytes, permitting 6 megabytes per semispace — Scheme48 uses a simple stop & copy garbage collector.[1] Since the current garbage collector cannot resize the heap dynamically if it becomes consistently too full, users on machines with much RAM may be more comfortable with liberally increasing this option.

`-s` *stack-size*
: The stack size, in cells. The default stack size is 10000 bytes, or 2500 cells. Note that this is only the size of the stack cache segment of memory for fast stack frame storage. When this overflows, there is no error; instead, Scheme48 simply copies the contents of the stack cache into the heap, until the frames it copied into the heap are needed later, at which point they are copied back into the stack cache. The `-s` option therefore affects only performance, not the probability of fatal stack overflow errors.

`-i` *image-filename*
: The filename of the suspended heap image to resume. When running the `scheme48` executable, the default is the regular Scheme48 image; when running the virtual machine directly, this option must be passed explicitly. For information on creating custom heap images, see Section 2.4.11 [Image-building commands], page 20, and also see Section 4.8 [Suspending and resuming heap images], page 75.

`-a` *argument ...*
: Command-line arguments to pass to the heap image's resumer, rather than being parsed by the virtual machine. In the usual Scheme48 command processor image, these arguments are put in a list of strings that will be the initial focus value (see Section 2.4.4 [Focus value], page 12).

`-u`
: Muffles warnings on startup about undefined imported foreign bindings.

The usual Scheme48 image may accept an argument of `batch`, using the `-a` switch to the virtual machine. This enters Scheme48 in batch mode, which displays no welcoming

[1] The Scheme48 team is also working on a new, generational garbage collector, but it is not in the standard distribution of Scheme48 yet.

banner, prints no prompt for inputs, and exits when an EOF is read. This may be used to run scripts from the command-line, often in the exec language (see Section 2.4.10 [Command programs], page 19), by sending text to Scheme48 through Unix pipes or shell heredocs. For example, this Unix shell command will load the command program in the file `foo.scm` into the exec language environment and exit Scheme48 when the program returns:

```
echo ,exec ,load foo.scm | scheme48 -a batch
```

This Unix shell command will load `packages.scm` into the module language environment, open the `tests` structure into the user environment, and call the procedure `run-tests` with zero arguments:

```
scheme48 -a batch <<END
,config ,load packages.scm
,open tests
(run-tests)
END
```

Scheme48 also supports [SRFI 22] and [SRFI 7] by providing R5RS and [SRFI 7] script interpreters in the location where Scheme48 binaries are kept as `scheme-r5rs` and `scheme-srfi-7`. See the [SRFI 22] and [SRFI 7] documents for more details. Scheme48's command processor also has commands for loading [SRFI 7] programs, with or without a [SRFI 22] script header; see Section 2.4.7 [SRFI 7], page 16.

2.1.1 Command processor introduction

The Scheme48 command processor is started up on resumption of the usual Scheme48 image. This is by default what the `scheme48` script installed by Scheme48 does. It will first print out a banner that contains some general information about the system, which will typically look something like this:

```
Welcome to Scheme 48 1.3 (made by root on Sun Jul 10 10:57:03 EDT 2005)
Copyright (c) 1993-2005 by Richard Kelsey and Jonathan Rees.
Please report bugs to scheme-48-bugs@s48.org.
Get more information at http://www.s48.org/.
Type ,? (comma question-mark) for help.
```

After the banner, it will initiate a REPL (read-eval-print loop). At first, there should be a simple '`>`' prompt. The command processor interprets Scheme code as well as *commands*. Commands operate the system at a level above or outside Scheme. They begin with a comma, and they continue until the end of the line, unless they expect a Scheme expression argument, which may continue as many lines as desired. Here is an example of a command invocation:

```
> ,set load-noisily on
```

This will set the `load-noisily` switch (see Section 2.4.2 [Command processor switches], page 11) on.

Note: If a command accepts a Scheme expression argument that is followed by more arguments, all of the arguments after the Scheme expression must be put on the same line as the last line of the Scheme expression.

Certain operations, such as breakpoints and errors, result in a recursive command processor to be invoked. This is known as *pushing a command level*. See Section 2.4.5 [Command

levels], page 13. Also, the command processor supports an *object inspector*, an interactive program for inspecting the components of objects, including continuation or stack frame objects; the debugger is little more than the inspector, working on continuations. See Section 2.4.9 [Inspector], page 18.

Evaluation of code takes place in the *interaction environment*. (This is what R5RS's `interaction-environment` returns.) Initially, this is the *user environment*, which by default is a normal R5RS Scheme environment. There are commands that set the interaction environment and evaluate code in other environments, too; see Section 2.4.6 [Module commands], page 15.

The command processor's prompt has a variety of forms. As above, it starts out with as a simple '`>`'. Several factors can affect the prompt. The complete form of the prompt is as follows:

- It begins with an optional command level (see Section 2.4.5 [Command levels], page 13) number: at the top level, there is no command level number; as command levels are pushed, the number is incremented, starting at 1.
- Optionally, the name of the interaction environment follows the command level number: if the interaction environment is the user environment, there is no name printed here; named environments are printed with their names; unnamed environments (usually created using the `,new-package` command; see Section 2.4.6 [Module commands], page 15) are printed with their numeric identifiers. If a command level number preceded an environment name, a space is printed between them.
- If the command processor is in the regular REPL mode, it ends with a '`>`' and a space before the user input area; if it is in inspector mode (see Section 2.4.9 [Inspector], page 18), it ends with a '`:`' and a space before the user input area.

For example, this prompt denotes that the user is in inspector mode at command level 3 and that the interaction environment is an environment named `frobozz`:

```
3 frobozz:
```

This prompt shows that the user is in the regular REPL mode at the top level, but in the environment for module descriptions (see Section 2.4.6 [Module commands], page 15):

```
config>
```

For a complete listing of all the commands in the command processor, see Section 2.4 [Command processor], page 10.

2.2 Emacs integration

Emacs is the canonical development environment for Scheme48. The `scheme.el` and `cmuscheme.el` packages provide support for editing Scheme code and running inferior Scheme processes, respectively. Also, the `scheme48.el` package provides more support for integrating directly with Scheme48.[2] `scheme.el` and `cmuscheme.el` come with GNU Emacs; `scheme48.el` is available separately from

`http://www.emacswiki.org/cgi-bin/wiki/download/scheme48.el`.

[2] `scheme48.el` is based on the older `cmuscheme48.el`, which is bundled with Scheme48 in the `emacs/` directory. Since `cmuscheme48.el` is older and less developed, it is not documented here.

To load `scheme48.el` if it is in the directory *emacs-dir*, add these lines to your `.emacs`:

```
(add-to-list 'load-path "emacs-dir/")
(autoload 'scheme48-mode "scheme48"
  "Major mode for improved Scheme48 integration."
  t)
(add-hook 'hack-local-variables-hook
          (lambda ()
            (if (and (boundp 'scheme48-package)
                     scheme48-package)
                (progn (scheme48-mode)
                       (hack-local-variables-prop-line)))))
```

The `add-hook` call sets Emacs up so that any file with a `scheme48-package` local variable specified in the file's `-*-` line or `Local Variables` section will be entered in Scheme48 mode. Files should use the `scheme48-package` variable to enable Scheme48 mode; they should not specify Scheme48 mode explicitly, since this would fail in Emacs instances without `scheme48.el`. That is, put this at the tops of files:

```
;;; -*- Mode: Scheme; scheme48-package: ... -*-
```

Avoid this at the tops of files:

```
;;; -*- Mode: Scheme48 -*-
```

There is also SLIME48, the Superior Lisp Interaction Mode for Emacs with Scheme48. It provides a considerably higher level of integration the other Emacs packages do, although it is less mature. It is at

`http://mumble.net/~campbell/scheme/slime48.tar.gz`;

there is also a Darcs repository[3] at

`http://mumble.net/~campbell/darcs/slime48/`.

Finally, `paredit.el` implements pseudo-structural editing facilities for S-expressions: it automatically balances parentheses and provides a number of high-level operations on S-expressions. `Paredit.el` is available on the web at

`http://mumble.net/~campbell/emacs/paredit.el`.

`cmuscheme.el` defines these:

`run-scheme` [*scheme-prog*] [Emacs command]
: Starts an inferior Scheme process or switches to a running one. With no argument, this uses the value of `scheme-program-name` to run the inferior Scheme system; with a prefix argument *scheme-prog*, this invokes *scheme-prog*.

`scheme-program-name` [Emacs variable]
: The Scheme program to invoke for inferior Scheme processes.

[3] Darcs is a revision control system; see

`http://www.darcs.net/`

for more details.

Under `scheme48-mode` with `scheme.el`, `cmuscheme.el`, and `scheme48.el`, these keys are defined:

C-M-f — `forward-sexp`
C-M-b — `backward-sexp`
C-M-k — `kill-sexp`
ESC C-DEL (*not* *C-M-DEL*) — `backward-kill-sexp`
C-M-q — `indent-sexp`
C-M-@ — `mark-sexp`
C-M-SPC — `mark-sexp`
S-expression manipulation commands. *C-M-f* moves forward by one S-expression; *C-M-b* moves backward by one. *C-M-k* kills the S-expression following the point; *ESC C-DEL* kills the S-expression preceding the point. *C-M-q* indents the S-expression following the point. *C-M-@* & *C-M-SPC*, equivalent to one another, mark the S-expression following the point.

C-c z — `switch-to-scheme`
Switches to the inferior Scheme process buffer.

C-c C-l — `scheme48-load-file`
Loads the file corresponding with the current buffer into Scheme48. If that file was not previously loaded into Scheme48 with *C-c C-l*, Scheme48 records the current interaction environment in place as it loads the file; if the file was previously recorded, it is loaded into the recorded environment. See Section 2.4.3 [Emacs integration commands], page 12.

C-c C-r — `scheme48-send-region`
C-c M-r — `scheme48-send-region-and-go`
C-c C-r sends the currently selected region to the current inferior Scheme process. The file of the current buffer is recorded as in the *C-c C-l* command, and code is evaluated in the recorded package. *C-c M-r* does similarly, but subsequently also switches to the inferior Scheme process buffer.

C-M-x — `scheme48-send-definition`
C-c C-e — `scheme48-send-definition`
C-c M-e — `scheme48-send-definition-and-go`
C-M-x (GNU convention) and *C-c C-e* send the top-level definition that the current point is within to the current inferior Scheme process. *C-c M-e* does similarly, but subsequently also switches to the inferior Scheme process buffer. *C-c C-e* and *C-c M-e* also respect Scheme48's file/environment mapping.

C-x C-e — `scheme48-send-last-sexp`
Sends the S-expression preceding the point to the inferior Scheme process. This also respects Scheme48's file/environment mapping.

2.3 Using the module system

Scheme48 is deeply integrated with an advanced module system. For complete detail of its module system, see Chapter 3 [Module system], page 23. Briefly, however:

- *Packages* are top-level environments suitable for evaluating expressions and definitions, either interactively, from files loaded on-the-fly, or as the bodies of modules. They can also access bindings exported by structures by *opening* the structures.
- *Structures* are libraries, or implementations of interfaces, exporting sets of bindings that packages can access. Underlying structures are usually packages, in which the user can, in some cases, interactively evaluate code during development.

Scheme48's usual development system, the command processor, provides a number of commands for working with the module system. For complete details, see Section 2.4.6 [Module commands], page 15. Chief among these commands are `,open` and `,in`. '`,open` *`struct`* `...`' makes all of the bindings from each of *struct* ... available in the interaction environment. Many of the sections in this manual describe one or more structures with the name they are given. For example, in order to use, or open, the multi-dimensional array library in the current interaction environment, one would enter

```
,open arrays
```

to the command processor. '`,in` *`struct`*' sets the interaction environment to be the package underlying *struct*. For instance, if, during development, the user decides that the package of the existing structure `foo` should open the structure `bar`, he might type

```
,in foo
,open bar
```

The initial interaction environment is known as the *user package*; the interaction environment may be reverted to the user package with the `,user` command.

Module descriptions, or code in the module configuration language (see Section 3.2 [Module configuration language], page 24) should be loaded into the special environment for that language with the `,config` command (see Section 2.4.6 [Module commands], page 15). *E.g.*, if `packages.scm` contains a set of module descriptions that the user wishes to load, among which is the definition of a structure `frobozz` which he wishes to open, he will typically send the following to the command processor prompt:

```
,config ,load packages.scm
,open frobozz
```

Note: These are commands for the interactive command processor, *not* special directives to store in files to work with the module system. The module language is disjoint from Scheme; for complete detail on it, see Chapter 3 [Module system], page 23.

2.3.1 Configuration mutation

(This section was derived from work copyrighted © 1993–2005 by Richard Kelsey, Jonathan Rees, and Mike Sperber.)

During program development, it is often desirable to make changes to packages and interfaces. In static languages, it is usually necessary to re-compile and re-link a program in order for such changes to be reflected in a running system. Even in interactive Common Lisp systems, a change to a package's exports often requires reloading clients that have already mentioned names whose bindings change. In those systems, once `read` resolves a use of a name to a symbol, that resolution is fixed, so a change in the way that a name resolves to a symbol can be reflected only by re-`read`ing all such references.

The Scheme48 development environment supports rapid turnaround in modular program development by allowing mutations to a program's configuration and giving a clear semantics to such mutation. The rule is that variable bindings in a running program are always resolved according to the current structure and interface bindings, even when these bindings change as a result of edits to the configuration. For example, consider the following:

```
(define-interface foo-interface (export a c))
(define-structure foo foo-interface
  (open scheme)
  (begin (define a 1)
         (define (b x) (+ a x))
         (define (c y) (* (b a) y))))
(define-structure bar (export d)
  (open scheme foo)
  (begin (define (d w) (+ (b w) a))))
```

This program has a bug. The variable named `b`, which is free in the definition of `d`, has no binding in `bar`'s package. Suppose that `b` was intended to be exported by `foo`, but was mistakenly omitted. It is not necessary to re-process `bar` or any of `foo`'s other clients at this point. One need only change `foo-interface` and inform the development system of that change (using, say, an appropriate Emacs command), and `foo`'s binding of `b` will be found when the procedure `d` is called and its reference to `b` actually evaluated.

Similarly, it is possible to replace a structure; clients of the old structure will be modified so that they see bindings from the new one. Shadowing is also supported in the same way. Suppose that a client package C opens a structure `mumble` that exports a name `x`, and `mumble`'s implementation obtains the binding of `x` from some other structure `frotz`. C will see the binding from `frotz`. If one then alters `mumble` so that it shadows `bar`'s binding of `x` with a definition of its own, procedures in C that refer to `x` will subsequently automatically see `mumble`'s definition instead of the one from `frotz` that they saw earlier.

This semantics might appear to require a large amount of computation on every variable reference: the specified behaviour appears to require scanning the package's list of opened structures and examining their interfaces — on every variable reference evaluated, not just at compile-time. However, the development environment uses caching with cache invalidation to make variable references fast, and most of the code is invoked only when the virtual machine traps due to a reference to an undefined variable.

2.3.2 Listing interfaces

The `list-interfaces` structure provides a utility for examining interfaces. It is usually opened into the config package with `,config ,open list-interfaces` in order to have access to the structures & interfaces easily.

`list-interface` *struct-or-interface* $\longrightarrow$ *unspecified* [procedure]

Lists all of the bindings exported by *struct-or-interface* along with their static types (see Section 3.4 [Static type system], page 30). For example,

```
> ,config ,open list-interfaces
> ,config (list-interface condvars)
condvar-has-value?        (proc (:condvar) :value)
condvar-value             (proc (:condvar) :value)
```

```
condvar?                    (proc (:value) :boolean)
make-condvar                (proc (&rest :value) :condvar)
maybe-commit-and-set-condvar! (proc (:condvar :value) :boolean)
maybe-commit-and-wait-for-condvar (proc (:condvar) :boolean)
set-condvar-has-value?!     (proc (:condvar :value) :unspecific)
set-condvar-value!          (proc (:condvar :value) :unspecific)
```

2.4 Command processor

The Scheme48 command processor is the main development environment. It incorporates a read-eval-print loop as well as an interactive inspector and debugger. It is well-integrated with the module system for rapid dynamic development, which is made even more convenient with the Emacs interface, `scheme48.el`; see Section 2.2 [Emacs integration], page 5.

2.4.1 Basic commands

There are several generally useful commands built-in, along with many others described in subsequent sections:

`,help` [command]
`,help` *command* [command]
`,?` [command]
`,?` *command* [command]

Requests help on commands. `,?` is an alias for `,help`. Plain '`,help`' lists a synopsis of all commands available, as well as all switches (see Section 2.4.2 [Command processor switches], page 11). '`,help` *command*' requests help on the particular command *command*.

`,exit` [command]
`,exit` *status* [command]
`,exit-when-done` [command]
`,exit-when-done` *status* [command]

Exits the command processor. '`,exit`' immediately exits with an exit status of 0. '`,exit` *status*' exits with the status that evaluating the expression *status* in the interaction environment produces. `,exit-when-done` is like `,exit`, but it waits until all threads complete before exiting.

`,go` *expression* [command]

`,go` is like `,exit`, except that it requires an argument, and it evaluates *expression* in the interaction environment in a *tail context* with respect to the command processor. This means that the command processor may no longer be reachable by the garbage collector, and may be collected as garbage during the evaluation of *expression*. For example, the full Scheme48 command processor is bootstrapped from a minimal one that supports the `,go` command. The full command processor is initiated in an argument to the command, but the minimal one is no longer reachable, so it may be collected as garbage, leaving only the full one.

`,run` *expression* [command]

Evaluates *expression* in the interaction environment. Alone, this command is not very useful, but it is required in situations such as the inspector (see Section 2.4.9 [In-

spector], page 18) and command programs (see Section 2.4.10 [Command programs], page 19).

`,undefine` *name* [command]
Removes the binding for *name* in the interaction environment.

`,load` *filename ...* [command]
Loads the contents each *filename* as Scheme source code into the interaction environment. Each *filename* is translated first (see Section 4.1.3 [Filenames], page 39). The given filenames may be surrounded or not by double-quotes; however, if a filename contains spaces, it must be surrounded by double-quotes. The differences between the `,load` command and Scheme's `load` procedure are that `,load` does not require its arguments to be quoted, allows arbitrarily many arguments while the `load` procedure accepts only one filename (and an optional environment), and works even in environments in which `load` is not bound.

`,translate` *from to* [command]
A convenience for registering a filename translation without needing to open the `filenames` structure. For more details on filename translations, see Section 4.1.3 [Filenames], page 39; this command corresponds with the `filename` structure's `set-translation!` procedure. As with `,load`, each of the filenames *from* and *to* may be surrounded or not by double-quotes, unless there is a space in the filenames, in which case it must be surrounded by double-quotes.

Note that in the exec language (see Section 2.4.10 [Command programs], page 19), `translate` is the same as the `filenames` structure's `set-translation!` procedure, *not* the procedure named `translate` from the `filenames` structure.

2.4.2 Switches

The Scheme48 command processor keeps track of a set of *switches*, user-settable configurations.

`,set` *switch* [command]
`,set` *switch* {*on*|*off*|*?*} [command]
`,unset` *switch* [command]
`,set` *?* [command]
'`,set` ***switch***' & '`,set` ***switch*** `on`' set the switch *switch* on. '`,unset` ***switch***' & '`,set` ***switch*** `off`' turn *switch* off. '`,set` ***switch*** `?`' gives a brief description of *switch*'s current status. '`,set ?`' gives information about all the available switches and their current state.

The following switches are defined. Each switch is listed with its name and its default status.

`ask-before-loading` *(off)*
If this is on, Scheme48 will prompt the user before loading modules' code. If it is off, it will quietly just load it.

`batch` *(off)*
: Batch mode is intended for automated uses of the command processor. With batch mode on, errors cause the command processor to exit, and the prompt is not printed.

`break-on-warnings` *(off)*
: If the `break-on-warnings` switch is on, warnings (see Section 4.2 [Condition system], page 51) signalled that reach the command processor's handler will cause a command level (see Section 2.4.5 [Command levels], page 13) to be pushed, similarly to breakpoints and errors.

`inline-values` *(off)*
: `Inline-values` tells whether or not certain procedures may be integrated inline.

`levels` *(on)*
: Errors will push a new command level (see Section 2.4.5 [Command levels], page 13) if this switch is on, or they will just reset back to the top level if `levels` is off.

`load-noisily` *(off)*
: Loading source files will cause messages to be printed if `load-noisily` is on; otherwise they will be suppressed.

2.4.3 Emacs integration commands

There are several commands that exist mostly for Emacs integration (see Section 2.2 [Emacs integration], page 5); although they may be used elsewhere, they are not very useful or convenient without `scheme48.el`.

`,from-file` *filename* [command]
`,end` [command]
: '`,from-file` *`filename`*' proclaims that the code following the command, until an `,end` command, comes from *filename* — for example, this may be due to an appropriate Emacs command, such as *`C-c C-l`* in `scheme48.el` —; if this is the first time the command processor has seen code from *filename*, it is registered to correspond with the interaction environment wherein the `,from-file` command was used. If it is not the first time, the code is evaluated within the package that was registered for *filename*.

`,forget` *filename* [command]
: Clears the command processor's memory of the package to which *filename* corresponds.

2.4.4 Focus value

The Scheme48 command processor maintains a current *focus value*. This is typically the value that the last expression evaluated to, or a list of values if it returned multiple values. If it evaluated to either zero values or Scheme48's 'unspecific' token (see Section 4.1 [System features], page 35), the focus value is unchanged. At the initial startup of Scheme48, the focus value is set to the arguments passed to Scheme48's virtual machine after the `-a`

argument on the command-line (see Section 2.1 [Running Scheme48], page 3). The focus value is accessed through the `##` syntax; the reader substitutes a special quotation (special so that the compiler will not generate warnings about a regular `quote` expression containing a weird value) for occurrences of `##`. Several commands, such as `,inspect` and `,dis`, either accept an argument or use the current focus value. Also, in the inspector (see Section 2.4.9 [Inspector], page 18), the focus object is the object that is currently being inspected.

```
> (cons 1 2)
'(1 . 2)
> ##
'(1 . 2)
> (begin (display "Hello, world!") (newline))
Hello, world!
> ##
'(1 . 2)
> (cdr ##)
2
> (define x 5)
; no values returned
> (+ ## x)
7
> (values 1 2 3)
; 3 values returned
1
2
3
> ##
'(1 2 3)
```

2.4.5 Command levels

The Scheme48 command processor maintains a stack of *command levels*, or recursive invocations of the command processor. Each command level retains information about the point from the previous command level at which it was pushed: the threads that were running — which the command processor suspends —, including the thread of that command level itself; the continuation of what pushed the level; and, if applicable, the condition (see Section 4.2 [Condition system], page 51) that caused the command level to be pushed. Each command level has its own thread scheduler, which controls all threads running at that level, including those threads' children.

Some beginning users may find command levels confusing, particularly those who are new to Scheme or who are familiar with the more simplistic interaction methods of other Scheme systems. These users may disable the command level system with the `levels` switch (see Section 2.4.2 [Command processor switches], page 11) by writing the command '`,set levels off`'.

`,push` [command]
`,pop` [command]
`,resume` [command]
`,resume` *level* [command]

`,reset` [command]
`,reset` *level* [command]

'`,push`' pushes a new command level. '`,pop`' pops the current command level. *C-d*/*^D*, or EOF, has the same effect as the `,pop` command. Popping the top command level inquires the user whether to exit or to return to the top level. '`,resume` *level*' pops all command levels down to *level* and resumes all threads that were running at *level* when it was suspended to push another command level. '`,reset` *level*' resets the command processor to *level*, terminating all threads at that level but the command reader thread. `,resume` & `,reset` with no argument use the top command level.

`,condition` [command]
`,threads` [command]

'`,condition`' sets the focus value to the condition that caused the command level to be pushed, or prints 'no condition' if there was no relevant condition. '`,threads`' invokes the inspector on the list of threads of the previous command level, or on nothing if the current command level is the top one.

```
> ,push
1> ,push
2> ,pop
1> ,reset

Top level
> ,open threads formats
> ,push
1> ,push
2> (spawn (lambda ()
            (let loop ()
              (sleep 10000)    ; Sleep for ten seconds.
              (format #t "~&foo~%")
              (loop)))
          'my-thread)
2>
foo
,push
3> ,threads
; 2 values returned
 [0] '#{Thread 4 my-thread}
 [1] '#{Thread 3 command-loop}
3: q
'(#{Thread 4 my-thread} #{Thread 3 command-loop})
3> ,resume 1

foo
2>
foo
,push
3> ,reset 1
```

```
Back to 1> ,pop
>
```

2.4.6 Module commands

Scheme48's command processor is well-integrated with its module system (see Chapter 3 [Module system], page 23). It has several dedicated environments, including the user package and the config package, and can be used to evaluate code within most packages in the Scheme48 image during program development. The config package includes bindings for Scheme48's configuration language; structure & interface definitions may be evaluated in it. The command processor also has provisions to support rapid development and module reloading by automatically updating references to redefined variables in compiled code without having to reload all of that code.

`,open` *struct ...* [command]

Opens each *struct* into the interaction environment, making all of its exported bindings available. This may have the consequence of loading code to implement those bindings. If there was code evaluated in the interaction environment that referred to a previously undefined variable for whose name a binding was exported by one of these structures, a message is printed to the effect that that binding is now available, and the code that referred to that undefined variable will be modified to subsequently refer to the newly available binding.

`,load-package` *struct* [command]
`,reload-package` *struct* [command]

`,load-package` and `,reload-package` both load the code associated with the package underlying *struct*, after ensuring that all of the other structures opened by that package are loaded as well. `,load-package` loads the code only if has not already been loaded; `,reload-package` unconditionally loads it.

`,user` [command]
`,user` *command-or-exp* [command]
`,config` [command]
`,config` *command-or-exp* [command]
`,for-syntax` [command]
`,for-syntax` *command-or-exp* [command]
`,new-package` [command]
`,in` *structure* [command]
`,in` *structure command-or-exp* [command]

These all operate on the interaction environment. '`,user`' sets it to the user package, which is the default at initial startup. '`,user` *`command-or-exp`*' temporarily sets the interaction environment to the user package, processes *command-or-exp*, and reverts the interaction environment to what it was before `,user` was invoked. The `,config` & `,for-syntax` commands are similar, except that they operate on the config package and the package used for the user package's macros (see Section 3.3 [Macros in concert with modules], page 28). '`,new-package`' creates a temporary, unnamed package with a vanilla R5RS environment and sets the interaction environment to it. That new package is not accessible in any way except to the user of the command processor, and it is destroyed after the user switches to another environment (unless the user uses the

`,structure` command; see below). '`,in` *`structure`*' sets the interaction environment to be *structure*'s package; *structure* is a name whose value is extracted from the config package. '`,in` *`structure command-or-exp`*' sets the interaction environment to *structure* temporarily to process *command-or-exp* and then reverts it to what it was before the use of `,in`. Note that, within a structure, the bindings available are exactly those bindings that would be available within the structure's static code, *i.e.* code in the structure's `begin` package clauses or code in files referred to by `files` package clauses.

`,user-package-is` *struct* [command]
`,config-package-is` *struct* [command]
`,user-package-is` & `,config-package-is` set the user & config packages, respectively, to be *struct*'s package. *Struct* is a name whose value is accessed from the current config package.

`,structure` *name interface* [command]
This defines a structure named *name* in the config package that is a view of *interface* on the current interaction environment.

2.4.7 SRFI 7

Scheme48 supports [SRFI 7] after loading the `srfi-7` structure by providing two commands for loading [SRFI 7] programs:

`,load-srfi-7-program` *name filename* [command]
`,load-srfi-7-script` *name filename* [command]
These load [SRFI 7] a program into a newly constructed structure, named *name*, which opens whatever other structures are needed by features specified in the program. `,load-srfi-7-program` loads a simple [SRFI 7] program; `,load-srfi-7-script` skips the first line, intended for [SRFI 22] Unix scripts.

2.4.8 Debugging commands

There are a number of commands useful for debugging, along with a continuation inspector, all of which composes a convenient debugger.

`,bound?` *name* [command]
`,where` [command]
`,where` *procedure* [command]
`,bound?` prints out binding information about *name*, if it is bound in the interaction environment, or 'Not bound' if *name* is unbound. `,where` prints out information about what file and package its procedure argument was created in. If *procedure* is not passed, `,where` uses the focus value. If `,where`'s argument is not a procedure, it informs the user of this fact. If `,where` cannot find the location of its argument's creation, it prints 'Source file not recorded.'

`,expand` [command]
`,expand` *exp* [command]
`,dis` [command]

`,dis` *proc* [command]

`,expand` prints out a macro-expansion of *exp*, or the focus value if *exp* is not provided. The expression to be expanded should be an ordinary S-expression. The expansion may contain 'generated names' and 'qualified names.' These merely contain lexical context information that allow one to differentiate between identifiers with the same name. Generated names look like `#{Generated` *name unique-numeric-id*`}`. Qualified names appear to be vectors; they look like `#(>>` *introducer-macro name unique-numeric-id*`)`, where *introducer-macro* is the macro that introduced the name.

`,dis` prints out a disassembly of its procedure, continuation, or template argument. If *proc* is passed, it is evaluated in the interaction environment; if not, `,dis` disassembles the focus value. The disassembly is of Scheme48's virtual machine's byte code.[4]

`,condition` [command]

`,threads` [command]

For the descriptions of these commands, see Section 2.4.5 [Command levels], page 13. These are mentioned here because they are relevant in the context of debugging.

`,trace` [command]

`,trace` *name ...* [command]

`,untrace` [command]

`,untrace` *name ...* [command]

Traced procedures will print out information about when they are entered and when they exit. '`,trace`' lists all of the traced procedures' bindings. '`,trace` *name ...*' sets each *name* in the interaction environment, which should be bound to a procedure, to be a traced procedure over the original procedure. '`,untrace`' resets all traced procedures to their original, untraced procedures. '`,untrace` *name ...*' untraces each individual traced procedure of *name ...* in the interaction environment.

`,preview` [command]

Prints a trace of the previous command level's suspended continuation. This is analogous with stack traces in many debuggers.

`,debug` [command]

Invokes the debugger: runs the inspector on the previous command level's saved continuation. For more details, see Section 2.4.9 [Inspector], page 18.

`,proceed` [command]

`,proceed` *exp* [command]

Returns to the continuation of the condition signalling of the previous command level. Only certain kinds of conditions will push a new command level, however — breakpoints, errors, and interrupts, and, if the `break-on-warnings` switch is on, warnings —; also, certain kinds of errors that do push new command levels do not permit being proceeded from. In particular, only with a few VM primitives may the `,proceed` command be used. If *exp* is passed, it is evaluated in the interaction

[4] A description of the byte code is forthcoming, although it does not have much priority to this manual's author. For now, users can read the rudimentary descriptions of the Scheme48 virtual machine's byte code instruction set in `vm/interp/arch.scm` of Scheme48's Scheme source.

environment to produce the values to return; if it is not passed, zero values are returned.

2.4.9 Inspector

Scheme48 provides a simple interactive object inspector. The command processor's prompt's end changes from '`>`' to '`:`' when in inspection mode. The inspector is the basis of the debugger, which is, for the most part, merely an inspector of continuations. In the debugger, the prompt is '`debug:`'. In the inspector, objects are printed followed by menus of their components. Entries in the menu are printed with the index, which optionally includes a symbolic name, and the value of the component. For example, a pair whose car is the symbol `a` and whose cdr is the symbol `b` would be printed by the inspector like this:

```
'(a . b)

 [0: car] 'a
 [1: cdr] 'b
```

The inspector maintains a stack of the focus objects it previously inspected. Selecting a new focus object pushes the current one onto the stack; the `u` command pops the stack.

`,inspect` [command]
`,inspect` *exp* [command]
Invokes the inspector. If *exp* is present, it is evaluated in the user package and its result is inspected (or a list of results, if it returned multiple values, is inspected). If *exp* is absent, the current focus value is inspected.

The inspector operates with its own set of commands, separate from the regular interaction commands, although regular commands may be invoked from the inspector as normal. Inspector commands are entered with or without a preceding comma at the inspector prompt. Multiple inspector commands may be entered on one line; an input may also consist of an expression to be evaluated. If an expression is evaluated, its value is selected as the focus object. Note, however, that, since inspector commands are symbols, variables cannot be evaluated just by entering their names; one must use either the `,run` command or wrap the variables in a `begin`.

These inspector commands are defined:

`menu` [inspector command]
`m` [inspector command]
`Menu` prints a menu for the focus object. `M` moves forward in the current menu if there are more than sixteen items to be displayed.

`u` [inspector command]
Pops the stack of focus objects, discarding the current one and setting the focus object to the current top of the stack.

`q` [inspector command]
Quits the inspector, going back into the read-eval-print loop.

`template` [inspector command]
Attempts to coerce the focus object into a template. If successful, this selects it as the new focus object; if not, this prints an error to that effect. Templates are the static

components of closures and continuations: they contain the code for the procedure, the top-level references made by the procedure, literal constants used in the code, and any inferior templates of closures that may be constructed by the code.

`d` [inspector command]
Goes down to the parent of the continuation being inspected. This command is valid only in the debugger mode, *i.e.* when the focus object is a continuation.

2.4.10 Command programs

The Scheme48 command processor can be controlled programmatically by *command programs*, programs written in the *exec language*. This language is essentially a mirror of the commands but in a syntax using S-expressions. The language also includes all of Scheme. The exec language is defined as part of the *exec package*.

`,exec` [command]
`,exec` *command* [command]
Sets the interaction environment to be the exec package. If an argument is passed, it is set temporarily, only to run the given command.

Commands in the exec language are invoked as procedures in Scheme. Arguments should be passed as follows:

- Identifiers, such as those of structure names in the config package, should be passed as literal symbols. For instance, the command '`,in frobbotz`' would become in the exec language `(in 'frobbotz)`.
- Filenames should be passed as strings; *e.g.*, '`,dump frob.image`' becomes `(dump "frob.image")`.
- Commands should be represented in list values with the car being the command name and the cdr being the arguments. Note that when applying a command an argument that is a command invocation is often quoted to produce a list, but the list should not include any quotation; for instance, '`,in mumble ,undefine frobnicate`' would become `(in 'mumble '(undefine frobnicate))`, even though simply '`,undefine frobnicate`' would become `(undefine 'frobnicate)`.

  The reason for this is that the command invocation in the exec language is different from a list that represents a command invocation passed as an argument to another command; since commands in the exec language are ordinary procedures, the arguments must be quoted, but the quoted arguments are not themselves evaluated: they are applied as commands.

  An argument to a command that expects a command invocation can also be a procedure, which would simply be called with zero arguments. For instance, `(config (lambda () (display (interaction-environment)) (newline)))` will call the given procedure with the interaction environment set to the config package.
- Expressions must be passed using the `run` command. For example, the equivalent of '`,user (+ 1 2)`' in the exec language would be `(user '(run (+ 1 2)))`.

Command programs can be loaded by running the `,load` command in the exec package. Scripts to load application bundles are usually written in the exec language and loaded into the exec package. For example, this command program, when loaded into the exec package,

will load `foo.scm` into the config package, ensure that the package `frobbotzim` is loaded, and open the `quuxim` structure in the user package:

```
(config '(load "foo.scm"))
(load-package 'frobbotzim)
(user '(open quuxim))
```

2.4.11 Image-building commands

Since Scheme48's operation revolves about an image-based model, these commands provide a way to save heap images on the file system, which may be resumed by invoking the Scheme48 virtual machine on them as in Section 2.1 [Running Scheme48], page 3.

`,build` *resumer filename* [command]
`,dump` *filename* [command]
`,dump` *filename message* [command]

`,build` evaluates *resumer*, whose value should be a unary procedure, and builds a heap image in *filename* that, when resumed by the virtual machine, will pass the resumer all of the command-line arguments after the `-a` argument to the virtual machine. The run-time system will have been initialized as with usual resumers (see Section 4.8 [Suspending and resuming heap images], page 75), and a basic condition handler will have been installed by the time that the resumer is called. On Unix, *resumer* must return an integer exit status for the process. `,dump` dumps the Scheme48 command processor, including all of the current settings, to *filename*. If *message* is passed, it should be a string delimited by double-quotes, and it will be printed as part of the welcome banner on startup; its default value, if it is not present, is `"(suspended image)"`.

2.4.12 Resource statistics and control

Scheme48 provides several devices for querying statistics about various resources and controlling resources, both in the command processor and programmatically.

`,collect` [command]

Forces a garbage collection and prints the amount of space in the heap before and after the collection.

`,time` *expression* [command]

Evaluates *expression* and prints how long it took. Three numbers are printed: run time, GC time, and real time. The run time is the amount of time in Scheme code; the GC time is the amount of time spent in the garbage collector; and the real time is the actual amount of time that passed during the expression's evaluation.

`,keep` [command]
`,keep` *kind ...* [command]
`,flush` [command]
`,flush` *kind ...* [command]

Scheme48 maintains several different kinds of information used for debugging information. '`,keep`' with no arguments shows what kinds of debugging data are preserved and what kinds are not. '`,keep` *kind ...*' requests that the debugging data of the given kinds should be kept; the `,flush` command requests the opposite. '`,flush`' with

no arguments flushes location names and resets the debug data table. The following are the kinds of debugging data:

`names` procedure names

`maps` environment maps used by the debugger to show local variable names

`files` filenames where procedures were defined

`source` source code surrounding continuations, printed by the debugger

`tabulate` if true, will store debug data records in a global table that can be easily flushed; if false, will store directly in compiled code

`,flush` can also accept `location-names`, which will flush the table of top-level variables' names (printed, for example, by the `,bound?` command); `file-packages`, which will flush the table that maps filenames to packages in which code from those files should be evaluated; or `table`, in which case the table of debug data is flushed.

Removing much debug data can significantly reduce the size of Scheme48 heap images, but it can also make error messages and debugging much more difficult. Usually, all debug data is retained; only for images that must be small and that do not need to be debuggable should the debugging data flags be turned off.

The `spatial` structure exports these utilities for displaying various statistics about the heap:

`space` ⟶ *unspecified* [procedure]
`vector-space` [*predicate*] ⟶ *unspecified* [procedure]
`record-space` [*predicate*] ⟶ *unspecified* [procedure]

`Space` prints out a list of the numbers of all objects and the number of bytes allocated for those objects on the heap, partitioned by the objects' primitive types and whether or not they are immutable (pure) or mutable (impure). `Vector-space` prints the number of vectors and the number of bytes used to store those vectors of several different varieties, based on certain heuristics about their form. If the predicate argument is passed, it gathers only vectors that satisfy that predicate. `Record-space` prints out, for each record type in the heap, both the number of all instances of that record type and the number of bytes used to store all of those instances. Like `vector-space`, if the predicate argument is passed, `record-space` will consider only those records that satisfy the predicate.

All of these three procedures first invoke the garbage collector before gathering statistics.

The `traverse` structure provides a simple utility for finding paths by which objects refer to one another.

`traverse-breadth-first` *object* ⟶ *unspecified* [procedure]
`traverse-depth-first` *object* ⟶ *unspecified* [procedure]

These traverse the heap, starting at *object*, recording all objects transitively referred to. `Traverse-breadth-first` uses a FIFO-queue-directed breadth-first graph traversal, while `traverse-depth-first` uses a LIFO-stack-directed depth-first graph traversal. The traversal halts at any leaves in the graph, which are distinguished by

an internal *leaf predicate* in the module. See below on `set-leaf-predicate!` on how to customize this and what the default is.

The traversal information is recorded in a global resource; it is not thread-safe, and intended only for interactive usage. The record can be reset by passing some simple object with no references to either `traverse-breadth-first` or `traverse-depth-first`; *e.g.*, `(traverse-depth-first #f)`.

`trail` *object* ⟶ *unspecified* [procedure]

After traversing the heap from an initial object, `(trail` *`object`*`)` prints the path of references and intermediate objects by which the initial object holds a transitive reference to *object*.

`set-leaf-predicate!` *predicate* ⟶ *unspecified* [procedure]
`usual-leaf-predicate` *object* ⟶ *boolean* [procedure]

`Set-leaf-predicate!` sets the current leaf predicate to be *predicate*. `Usual-leaf-predicate` is the default leaf predicate; it considers simple numbers (integers and flonums), strings, byte vectors, characters, and immediate objects (true, false, nil, and the unspecific object) to be leaves, and everything else to be branches.

3 Module system

Scheme48 has an advanced module system that is designed to interact well with macros, incremental compilation, and the interactive development environment's (see Chapter 2 [User environment], page 3) code reloading facilities for rapid program development. For details on the integration of the module system and the user environment for rapid code reloading, see Section 2.3 [Using the module system], page 7.

3.1 Module system architecture

The fundamental mechanism by which Scheme code is evaluated is the lexical environment. Scheme48's module system revolves around this fundamental concept. Its purpose is to control the denotation of names in code[1] in a structured, modular manner. The module system is manipulated by a static *configuration language*, described in the next section; this section describes the concepts in the architecture of the module system.

The *package* is the entity internal to the module system that maps a set of names to denotations. For example, the package that represents the Scheme language maps `lambda` to a descriptor for the special form that the compiler interprets to construct a procedure, `car` to the procedure that accesses the car of a pair, *&c*. Packages are not explicitly manipulated by the configuration language, but they lie underneath structures, which are described below. A package also contains the code of a module and controls the visibility of names within that code. It also includes some further information, such as optimizer switches. A *structure* is a view on a package; that is, it contains a package and an *interface* that lists all of the names it exports to the outside. Multiple structures may be constructed atop a single package; this mechanism is often used to offer multiple abstraction levels to the outside. A *module* is an abstract entity: it consists of some code, the namespace visible to the code, and the set of abstractions or views upon that code.

A package contains a list of the structures whose bindings should be available in the code of that package. If a structure is referred to in a such a list of a package, the package is said to *open* that structure. It is illegal for a package to open two structures whose interfaces contain the same name.[2] Packages may also modify the names of the bindings that they import. They may import only selected bindings, exclude certain bindings from structures, rename imported bindings, create alias bindings, and add prefixes to names.

Most packages will open the standard `scheme` structure, although it is not implicitly opened, and the module system allows not opening `scheme`. It may seem to be not very useful to not open it, but this is necessary if some bindings from it are intended to be shadowed by another structure, and it allows for entirely different languages from Scheme to be used in a package's code. For example, Scheme48's byte code interpreter virtual machine is implemented in a subset of Scheme called Pre-Scheme, which is described in a later chapter in this manual. The modules that compose the VM all open not the `scheme` structure but the `prescheme` structure. The configuration language itself is controlled by the module system, too. In another example, from Scsh, the Scheme shell, there is a structure

[1] This is in contrast to, for example, Common Lisp's package system, which controls the mapping from strings to names.

[2] The current implementation, however, does not detect this. Instead it uses the left-most structure in the list of a package's `open` clause; see the next section for details on this.

`scsh` that contains all of the Unix shell programming facilities. However, the `scsh` structure necessarily modifies some of the bindings related to I/O that the `scheme` structure exports. Modules could not open both `scheme` and `scsh`, because they both provide several bindings with the same names, so Scsh defines a more convenient `scheme-with-scsh` structure that opens both `scheme`, but with all of the shadowed bindings excluded, and `scsh`; modules that use Scsh would open neither `scsh` nor `scheme`: they instead open just `scheme-with-scsh`.

Interfaces are separated from structures in order that they may be reüsed and combined. For example, several different modules may implement the same abstractions differently. The structures that they include would, in such cases, reüse the same interfaces. Also, it is sometimes desirable to combine several interfaces into a *compound interface*; see the `compound-interface` form in the next section. Furthermore, during interactive development, interface definitions may be reloaded, and the structures that use them will automatically begin using the new interfaces; see Section 2.3 [Using the module system], page 7.

Scheme48's module system also supports *parameterized modules*. Parameterized modules, sometimes known as *generic modules*, *higher-order modules* or *functors*, are essentially functions at the module system level that map structures to structures. They may be instantiated or applied arbitrarily many times, and they may accept and return arbitrarily many structures. Parameterized modules may also accept and return other parameterized modules.

3.2 Module configuration language

Scheme48's module system is used through a *module configuration language*. *The configuration language is entirely separate from Scheme.* Typically, in one configuration, or set of components that compose a program, there is an `interfaces.scm` file that defines all of the interfaces used by the configuration, and there is also a `packages.scm` file that defines all of the packages & structures that compose it. Note that modules are not necessarily divided into files or restricted to one file: modules may include arbitrarily many files, and modules' code may also be written in-line to structure expressions (see the `begin` package clause below), although that is usually only for expository purposes and trivial modules.

Structures are always created with corresponding *package clauses*. Each clause specifies an attribute of the package that underlies the structure or structures created using the clauses. There are several different types of clauses:

`open` *structure* ... [package clause]
`access` *structure* ... [package clause]

> `Open` specifies that the package should open each of the listed structures, whose packages will be loaded if necessary. `Access` specifies that each listed structure should be accessible using the (`structure-ref` *structure identifier*) special form, which evaluates to the value of *identifier* exported by the accessed structure *structure*. `Structure-ref` is available from the `structure-refs` structure. Each *structure* passed to `access` is not opened, however; the bindings exported thereby are available only using `structure-ref`. While the qualified `structure-ref` mechanism is no longer useful in the presence of modified structures (see below on `modify`, `subset`, &

`with-prefix`), some old code still uses it, and `access` is also useful to force that the listed structures' packages be loaded without cluttering the namespace of the package whose clauses the `access` clause is among.

`for-syntax` *package-clause ...* [package clause]
Specifies a set of package clauses for the next floor of the reflective tower; see Section 3.3 [Macros in concert with modules], page 28.

`files` *file-specifier ...* [package clause]
`begin` *code ...* [package clause]
`Files` and `begin` specify the package's code. `Files` takes a sequence of namelists for the filenames of files that contain code; see Section 4.1.3 [Filenames], page 39. `Begin` accepts in-line program code.

`optimize` *optimizer-specifier ...* [package clause]
`integrate` [*on?*] [package clause]
`Optimize` clauses request that specified compiler optimizers be applied to the code. (Actually, 'optimizer' is a misnomer. The `optimize` clause may specify arbitrary passes that the compiler can be extended with.) `Integrate` clauses specify whether or not integrable procedures from other modules, most notably Scheme primitives such as `car` or `vector-ref`, should actually be integrated in this package. This is by default on. Most modules should leave it on for any reasonable performance; only a select few, into which code is intended to be dynamically loaded frequently and in which redefinition of imported procedures is common, need turn this off. The value of the argument to `integrate` clauses should be a literal boolean, *i.e.* `#t` or `#f`; if no argument is supplied, integration is enabled by default.

Currently, the only optimizer built-in to Scheme48 is the automatic procedure integrator, or `auto-integrate`, which attempts stronger type reconstruction than is attempted with most code (see Section 3.4 [Static type system], page 30) and selects procedures below a certain size to be made integrable (so that the body will be compiled in-line in all known call sites). Older versions of Scheme48 also provided another optimizer, `flat-environments`, which would flatten certain lexical closure environments, rather than using a nested environment structure. Now, however, Scheme48's byte code compiler always flattens environments; specifying `flat-environments` in an `optimize` clause does nothing.

A configuration is a sequence of definitions. There are definition forms for only structures and interfaces.

`define-structure` *name interface package-clause ...* [configuration form]
`define-structures` ((*name interface*) ...) *package-clause ...* [configuration form]
`Define-structure` creates a package with the given package clauses and defines *name* to be the single view atop it, with the interface *interface*. `Define-structures` also creates a package with the given package clauses; upon that package, it defines each *name* to be a view on it with the corresponding interface.

`define-module` (*name parameter ...*) *definition ... result* [configuration form]
`def` *name ...* (*parameterized-module argument ...*) [configuration form]
`Define-module` defines *name* to be a parameterized module that accepts the given parameters.

`define-interface` *name interface* [configuration form]
Defines *name* to be the interface that *interface* evaluates to. *Interface* may either be an interface constructor application or simply a name defined to be an interface by some prior `define-interface` form.

`export` *export-specifier ...* [interface constructor]
`Export` constructs a simple interface with the given export specifiers. The export specifiers specify names to export and their corresponding static types. Each *export-specifier* should have one of the following forms:

symbol in which case *symbol* is exported with the most general value type;

(*symbol type*)
in which case *symbol* is exported with the given type; or

((*symbol ...*) *type*)
in which case each *symbol* is exported with the same given type

For details on the valid forms of *type*, see Section 3.4 [Static type system], page 30.
Note: All macros listed in interfaces *must* be explicitly annotated with the type `:syntax`; otherwise they would be exported with a Scheme value type, which would confuse the compiler, because it would not realize that they are macros: it would instead treat them as ordinary variables that have regular run-time values.

`compound-interface` *interface ...* [interface constructor]
This constructs an interface that contains all of the export specifiers from each *interface*.

Structures may also be constructed anonymously; this is typically most useful in passing them to or returning them from parameterized modules.

`structure` *interface package-clauses* [structure constructor]
`structures` (*interface ...*) *package-clauses* [structure constructor]
`Structure` creates a package with the given clauses and evaluates to a structure over it with the given interface. `Structures` does similarly, but it evaluates to a number of structures, each with the corresponding *interface*.

`subset` *structure* (*name ...*) [structure constructor]
`with-prefix` *structure name* [structure constructor]
`modify` *structure modifier ...* [structure constructor]
These modify the interface of *structure*. `Subset` evaluates to a structure that exports only *name ...*, excluding any other names that *structure* exported. `With-prefix` adds a prefix *name* to every name listed in *structure*'s interface. Both `subset` and `with-prefix` are syntactic sugar for the more general `modify`, which applies the modifier commands in a strictly right-to-left or last-to-first order. **Note:** These all *denote new structures with new interfaces*; they do not destructively modify existing structures' interfaces.

`prefix` *name* [modifier command]
`expose` *name* ... [modifier command]
`hide` *name* ... [modifier command]
`alias` (*from to*) ... [modifier command]
`rename` (*from to*) ... [modifier command]

`Prefix` adds the prefix *name* to every exported name in the structure's interface. `Expose` exposes only *name* ...; any other names are hidden. `Hide` hides *name* .... `Alias` exports each *to* as though it were the corresponding *from*, as well as each *from*. `Rename` exports each *to* as if it were the corresponding *from*, but it also hides the corresponding *from*.

Examples:

```
(modify structure
        (prefix foo:)
        (expose bar baz quux))
```

makes only `foo:bar`, `foo:baz`, and `foo:quux`, available.

```
(modify structure
        (hide baz:quux)
        (prefix baz:)
        (rename (foo bar)
                (mumble frotz))
        (alias (gargle mumph)))
```

exports `baz:gargle` as what was originally `mumble`, `baz:mumph` as an alias for what was originally `gargle`, `baz:frotz` as what was originally `mumble`, `baz:bar` as what was originally `foo`, *not* `baz:quux` — what was originally simply `quux` —, and everything else that *structure* exported, but with a prefix of `baz:`.

There are several simple utilities for binding variables to structures locally and returning multiple structures not necessarily over the same package (*i.e.* not with `structures`). These are all valid in the bodies of `define-module` and `def` forms, and in the arguments to parameterized modules and `open` package clauses.

`begin` *body* [syntax]
`let` ((*name value*) ...) *body* [syntax]
`receive` (*name* ...) *producer body* [syntax]
`values` *value* ... [syntax]

These are all as in ordinary Scheme. Note, however, that there is no reasonable way by which to use `values` except to call it, so it is considered a syntax; also note that `receive` may not receive a variable number of values — *i.e.* there are no 'rest lists' —, because list values in the configuration language are nonsensical.

Finally, the configuration language also supports syntactic extensions, or macros, as in Scheme.

`define-syntax` *name transformer-specifier* [configuration form]

Defines the syntax transformer *name* to be the transformer specified by *transformer-specifier*. *Transformer-specifier* is exactly the same as in Scheme code; it is evaluated as ordinary Scheme.

3.3 Macros in concert with modules

One reason that the standard Scheme language does not support a module system yet is the issue of macros and modularity. There are several issues to deal with:

- that compilation of code that uses macros requires presence of those macros' definitions, which prevents true separate compilation, because those macros may be from other modules;
- that a macro's expansion must preserve referential transparency and hygiene, for example in cases where it refers to names from within the module in which it was defined, even if those names weren't exported; and
- that a macro's code may be arbitrary Scheme code, which in turn can use other modules, so one module's compile-time, when macros are expanded, is another's run-time, when the code used in macros is executed by the expander: this makes a tower of phases of code evaluation over which some coherent control must be provided.

Scheme48's module system tries to address all of these issues coherently and comprehensively. Although it cannot offer *total* separate compilation, it can offer incremental compilation, and compiled modules can be dumped to the file system & restored in the process of incremental compilation.[3]

Scheme48's module system is also very careful to preserve non-local module references from a macro's expansion. Macros in Scheme48 are required to perform hygienic renaming in order for this preservation, however; see Section 4.1.12 [Explicit renaming macros], page 48. For a brief example, consider the `delay` syntax for lazy evaluation. It expands to a simple procedure call:

```
(delay expression)
  ↦ (make-promise (lambda () expression))
```

However, `make-promise` is not exported from the `scheme` structure. The expansion works correctly due to the hygienic renaming performed by the `delay` macro transformer: when it hygienically renames `make-promise`, the output contains not the symbol but a special token that refers exactly to the binding of `make-promise` from the environment in which the `delay` macro transformer was defined. Special care is taken to preserve this information. Had `delay` expanded to a simple S-expression with simple symbols, it would have generated a free reference to `make-promise`, which would cause run-time undefined variable errors, or, if the module in which `delay` was used had its *own* binding of or imported a binding of the name `make-promise`, `delay`'s expansion would refer to the wrong binding, and there could potentially be drastic and entirely unintended impact upon its semantics.

Finally, Scheme48's module system has a special design for the tower of phases, called a *reflective tower*.[4] Every storey represents the environment available at successive macro levels. That is, when the right-hand side of a macro definition or binding is evaluated in an environment, the next storey in that environment's reflective tower is used to evaluate that macro binding. For example, in this code, there are two storeys used in the tower:

```
(define (foo ...bar...)
  (let-syntax ((baz ...quux...))
```

[3] While such facilities are not built-in to Scheme48, there is a package to do this, which will probably be integrated at some point soon into Scheme48.

[4] This would be more accurately named 'syntactic tower,' as it has nothing to do with reflection.

```
...zot...))
```

In order to evaluate code in one storey of the reflective tower, it is necessary to expand all macros first. Most of the code in this example will eventually be evaluated in the first storey of the reflective tower (assuming it is an ordinary top-level definition), but, in order to expand macros in that code, the `let-syntax` must be expanded. This causes `...quux...` to be evaluated in the *second* storey of the tower, after which macro expansion can proceed, and long after which the enclosing program can be evaluated.

The module system provides a simple way to manipulate the reflective tower. There is a package clause, `for-syntax`, that simply contains package clauses for the next storey in the tower. For example, a package with the following clauses:

```
(open scheme foo bar)
(for-syntax (open scheme baz quux))
```

has all the bindings of `scheme`, `foo`, & `bar`, at the ground storey; and the environment in which macros' definitions are evaluated provides everything from `scheme`, `baz`, & `quux`.

With no `for-syntax` clauses, the `scheme` structure is implicitly opened; however, if there are `for-syntax` clauses, `scheme` must be explicitly opened.[5] Also, `for-syntax` clauses may be arbitrarily nested: reflective towers are theoretically infinite in height. (They are internally implemented lazily, so they grow exactly as high as they need to be.)

Here is a simple, though contrived, example of using `for-syntax`. The `while-loops` structure exports `while`, a macro similar to C's `while` loop. `While`'s transformer unhygienically binds the name `exit` to a procedure that exits from the loop. It necessarily, therefore, uses explicit renaming macros (see Section 4.1.12 [Explicit renaming macros], page 48) in order to break hygiene; it also, in the macro transformer, uses the `destructure` macro to destructure the input form (see Section 6.8 [Library utilities], page 109, in particular, the structure `destructuring` for destructuring S-expressions).

```
(define-structure while-loops (export while)
  (open scheme)
  (for-syntax (open scheme destructuring))
  (begin
    (define-syntax while
      (lambda (form r compare)
        (destructure (((WHILE test . body) form))
          `(,(r 'CALL-WITH-CURRENT-CONTINUATION)
             (,(r 'LAMBDA) (EXIT)
               (,(r 'LET) (r 'LOOP) ()
                 (,(r 'IF) ,test
                   (,(r 'BEGIN)
                     ,@body
                     (,(r 'LOOP)))))))))
      (CALL-WITH-CURRENT-CONTINUATION LAMBDA LET IF BEGIN))))
```

This next `while-example` structure defines an example procedure `foo` that uses `while`. Since `while-example` has no macro definitions, there is no need for any `for-syntax` clauses;

[5] This is actually only in the default config package of the default development environment. The full mechanism is very general.

it imports `while` from the `while-loops` structure only at the ground storey, because it has no macro bindings to evaluate the transformer expressions of:

```
(define-structure while-example (export foo)
  (open scheme while-loops)
  (begin
    (define (foo x)
      (while (> x 9)
        (if (integer? (sqrt x))
            (exit (expt x 2))
            (set! x (- x 1)))))))
```

3.4 Static type system

Scheme48 supports a rudimentary static type system. It is intended mainly to catch some classes of type and arity mismatch errors early, at compile-time. By default, there is only *extremely* basic analysis, which is typically only good enough to catch arity errors and the really egregious type errors. The full reconstructor, which is still not very sophisticated, is enabled by specifying an optimizer pass that invokes the code usage analyzer. The only optimizer pass built-in to Scheme48, the automatic procedure integrator, named `auto-integrate`, does so.

The type reconstructor attempts to assign the most specific type it can to program terms, signalling warnings for terms that are certain to be invalid by Scheme's dynamic semantics. Since the reconstructor is not very sophisticated, it frequently gives up and assigns very general types to many terms. Note, however, that it is very lenient in that it only assigns more general types: it will *never* signal a warning because it could not reconstruct a very specific type. For example, the following program will produce no warnings:

```
(define (foo x y) (if x (+ y 1) (car y)))
```

Calls to `foo` that are clearly invalid, such as `(foo #t 'a)`, could cause the type analyzer to signal warnings, but it is not sophisticated enough to determine that `foo`'s second argument must be either a number or a pair; it simply assigns a general value type (see below).

There are some tricky cases that depend on the order by which arguments are evaluated in a combination, because that order is not specified in Scheme. In these cases, the relevant types are narrowed to the most specific ones that could not possibly cause errors at run-time for any order. For example,

```
(lambda (x) (+ (begin (set! x '(3)) 5) (car x)))
```

will be assigned the type `(proc (:pair) :number)`, because, if the arguments are evaluated right-to-left, and `x` is not a pair, there will be a run-time type error.

The type reconstructor presumes that all code is potentially reachable, so it may signal warnings for code that the most trivial control flow analyzer could decide unreachable. For example, it would signal a warning for `(if #t 3 (car 7))`. Furthermore, it does not account for continuation throws; for example, though it is a perfectly valid Scheme program, the type analyzer might signal a warning for this code:

```
(call-with-current-continuation
  (lambda (k) (0 (k))))
```

The type system is based on a type lattice. There are several maximum or 'top' elements, such as `:values`, `:syntax`, and `:structure`; and one minimum or 'bottom' element, `:error`. This description of the type system makes use of the following notations: E `:` T means that the term E has the type, or some compatible subtype of, T; and $T_a \sqsubseteq T_b$ means that T_a is a compatible subtype of T_b — that is, any term whose static type is T_a is valid in any context that expects the type T_b —.

Note that the previous text has used the word 'term,' not 'expression,' because static types are assigned to not only Scheme expressions. For example, `cond` macro has the type `:syntax`. Structures in the configuration language also have static types: their interfaces. (Actually, they really have the type `:structure`, but this is a deficiency in the current implementation's design.) Types, in fact, have their own type: `:type`. Here are some examples of values, first-class or otherwise, and their types:

```
cond : :syntax

(values 1 'foo '(x . y))
  : (some-values :exact-integer :symbol :pair)

:syntax : :type

3 : :exact-integer

(define-structure foo (export a b) ...)
foo : (export a b)
```

One notable deficiency of the type system is the absence of any sort of parametric polymorphism.

`join` *type* ... [type constructor]
`meet` *type* ... [type constructor]

`Join` and `meet` construct the supremum and infimum elements in the type lattice of the given types. That is, for any two disjoint types T_a and T_b, let T_j be (`join` T_a T_b) and T_m be (`meet` T_a T_b):

- $T_j \sqsubseteq T_a$ and $T_j \sqsubseteq T_b$
- $T_a \sqsubseteq T_m$ and $T_b \sqsubseteq T_m$

For example, (`join :pair :null`) allows either pairs or nil, *i.e.* lists, and (`meet :integer :exact`) accepts only integers that are also exact.

(More complete definitions of supremum, infimum, and other elements of lattice theory, may be found elsewhere.)

`:error` [type]

This is the minimal, or 'bottom,' element in the type lattice. It is the type of, for example, calls to `error`.

`:values` [type]
`:arguments` [type]

All Scheme *expressions* have the type `:values`. They may have more specific types as well, but all expressions' types are compatible subtypes of `:values`. `:Values` is a maximal element of the type lattice. `:Arguments` is synonymous with `:values`.

`:value` [type]

Scheme expressions that have a single result have the type `:value`, or some compatible subtype thereof; it is itself a compatible subtype of `:values`.

`some-values` *type ...* [type constructor]

`Some-values` is used to denote the types of expressions that have multiple results: if E_1 ... E_n have the types T_1 ... T_n, then the Scheme expression `(values` E_1 ... E_n`)` has the type `(some-values` T_1 ... T_n`)`.

`Some-values`-constructed types are compatible subtypes of `:values`.

`Some-values` also accepts 'optional' and 'rest' types, similarly to Common Lisp's 'optional' and 'rest' formal parameters. The sequence of types may contain a `&opt` token, followed by which is any number of further types, which are considered to be optional. For example, `make-vector`'s domain is `(some-values :exact-integer &opt :value)`. There may also be a `&rest` token, which must follow the `&opt` token if there is one. Following the `&rest` token is one more type, which the rest of the sequents in a sequence after the required or optional sequents must satisfy. For example, `map`'s domain is `(some-values :procedure (join :pair :null) &rest (join :pair :null))`: it accepts one procedure and at least one list (pair or null) argument.

`procedure` *domain codomain* [type constructor]
`proc` (*arg-type ...*) *result-type* [type constructor]

Procedure type constructors. Procedure types are always compatible subtypes of `:value`. `Procedure` is a simple constructor from a specific domain and codomain; *domain* and *codomain* must be compatible subtypes of `:values`. `Proc` is a more convenient constructor. It is equivalent to `(procedure (some-values` *arg-type ...*`)` *result-type*`)`.

`:boolean` [type]
`:char` [type]
`:null` [type]
`:unspecific` [type]
`:pair` [type]
`:string` [type]
`:symbol` [type]
`:vector` [type]
`:procedure` [type]
`:input-port` [type]
`:output-port` [type]

Types that represent standard Scheme data. These are all compatible subtypes of `:value`. `:Procedure` is the general type for all procedures; see `proc` and `procedure` for procedure types with specific domains and codomains.

`:number` [type]
`:complex` [type]
`:real` [type]
`:rational` [type]

`:integer` [type]
: Types of the Scheme numeric tower. `:integer` ⊑ `:rational` ⊑ `:real` ⊑ `:complex` ⊑ `:number`

`:exact` [type]
`:inexact` [type]
`:exact-integer` [type]
`:inexact-real` [type]
: `:Exact` and `:inexact` are the types of exact and inexact numbers, respectively. They are typically met with one of the types in the numeric tower above; `:exact-integer` and `:inexact-real` are two conveniences for the most common meets.

`:other` [type]
: `:Other` is for types that do not fall into any of the previous value categories. (`:other` ⊑ `:value`) All new types introduced, for example by `loophole` (see Section 4.1.11 [Type annotations], page 48), are compatible subtypes of `:other`.

`variable` *type* [type constructor]
: This is the type of all assignable variables, where ***type*** ⊑ `:value`. Assignment to variables whose types are value types, not assignable variable types, is invalid.

`:syntax` [type]
`:structure` [type]
: `:Syntax` and `:structure` are two other maximal elements of the type lattice, along with `:values`. `:Syntax` is the type of macros or syntax transformers. `:Structure` is the general type of all structures.

3.4.1 Types in the configuration language

Scheme48's configuration language has several places in which to write types. However, due to the definitions of certain elements of the configuration language, notably the `export` syntax, the allowable type syntax is far more limited than the above. Only the following are provided:

`:values` [type]
`:value` [type]
`:arguments` [type]
`:syntax` [type]
`:structure` [type]
: All of the built-in maximal elements of the type lattice are provided, as well as the simple compatible subtype `:values`, `:value`.

`:boolean` [type]
`:char` [type]
`:null` [type]
`:unspecific` [type]
`:pair` [type]
`:string` [type]
`:symbol` [type]
`:vector` [type]

`:procedure` [type]
`:input-port` [type]
`:output-port` [type]
`:number` [type]
`:complex` [type]
`:real` [type]
`:rational` [type]
`:integer` [type]
`:exact-integer` [type]

These are the only value types provided in the configuration language. Note the conspicuous absence of `:exact`, `:inexact`, and `:inexact-real`.

`procedure` *domain codomain* [type constructor]
`proc` (*arg-type ...*) *result-type* [type constructor]

These two are the only type constructors available. Note here the conspicuous absence of `some-values`, so procedure types that are constructed by `procedure` can accept only one argument (or use the overly general `:values` type) & return only one result (or, again, use `:values` for the codomain), and procedure types that are constructed by `proc` are similar in the result type.

4 System facilities

This chapter details many facilities that the Scheme48 run-time system provides.

4.1 System features

Scheme48 provides a variety of miscellaneous features built-in to the system.

4.1.1 Miscellaneous features

The structure `features` provides some very miscellaneous features in Scheme48.

`immutable?` *object* ⟶ *boolean* [procedure]
`make-immutable!` *object* ⟶ *object* [procedure]

All Scheme objects in Scheme48 have a flag determining whether or not they may be mutated. All immediate Scheme objects (`()`, `#f`, *&c.*) are immutable; all fixnums (small integers) are immutable; and all stored objects — vectors, pairs, *&c.* — may be mutable. `Immutable?` returns `#t` if *object* may not be mutated, and `make-immutable!`, a bit ironically, modifies *object* so that it may not be mutated, if it was not already immutable, and returns it.

```
(immutable? #t)                          ⇒ #t
(define p (cons 1 2))
(immutable? p)                           ⇒ #f
(car p)                                  ⇒ 1
(set-car! p 5)
(car p)                                  ⇒ 5
(define q (make-immutable! p))
(eq? p q)                                ⇒ #t
(car p)                                  ⇒ 5
(immutable? q)                           ⇒ #t
(set-car! p 6)                           error  immutable pair
```

`string-hash` *string* ⟶ *integer-hash-code* [procedure]

Computes a basic but fast hash of *string*.

```
(string-hash "Hello, world!")            ⇒ 1161
```

`force-output` *port* ⟶ *unspecified* [procedure]

Forces all buffered output to be sent out of *port*.

This is identical to the binding of the same name exported by the `i/o` structure (see Section 4.5.1 [Ports], page 58).

`current-noise-port` ⟶ *output-port* [procedure]

The current noise port is a port for sending noise messages that are inessential to the operation of a program.

The `silly` structure exports a single procedure, implemented as a VM primitive for the silly reason of efficiency, hence the name of the structure.[1] It is used in an inner loop of the reader.

[1] The author of this manual is not at fault for this nomenclature.

`reverse-list->string` *char-list count* ⟶ *string* [procedure]
Returns a string of the first *count* characters in *char-list*, in reverse. It is a serious error if *char-list* is not a list whose length is at least *count*; the error is not detected by the VM, so bogus pointers may be involved as a result. Use this routine with care in inner loops.

The `debug-messages` structure exports a procedure for emitting very basic debugging messages for low-level problems.

`debug-message` *item ...* ⟶ *unspecified* [procedure]
Prints *item ...* directly to an error port,[2] eliding buffering and thread synchronization on the Scheme side. Objects are printed as follows:

- Fixnums (small integers) are written in decimal.
- Characters are written literally with a `#\` prefix. No naming translation is performed, so the space and newline characters are written literally, not as `#\space` or `#\newline`.
- Records are written as #{*type-name*}, where *type-name* is the name of the record's type.
- Strings and symbols are written literally.
- Booleans and the empty list are written normally, *i.e.* as `#t`, `#f`, or `()`.
- Pairs are written as `(...)`.
- Vectors are written as `#(...)`.
- Objects of certain primitive types are written as #{*type*}: procedures, templates, locations, code (byte) vectors, and continuations.[3]
- Everything else is printed as #{???}.

The `code-quote` structure exports a variant of `quote` that is useful in some sophisticated macros.

`code-quote` *object* ⟶ *object* [special form]
Evaluates to the literal value of *object*. This is semantically identical to `quote`, but *object* may be anything, and the compiler will not signal any warnings regarding its value, while such warnings would be signalled for `quote` expressions that do not wrap readable S-expressions: arbitrary, compound, unreadable data may be stored in `code-quote`. Values computed at compile-time may thus be transmitted to run-time code. However, care should be taken in doing this.

4.1.2 Various utilities

The `util` structure contains some miscellaneous utility routines extensively used internally in the run-time system. While they are not meant to compose a comprehensive library (such as, for example, [SRFI 1]), they were found useful in building the run-time system without introducing massive libraries into the core of the system.

---

[2] On Unix, this is `stderr`, the standard I/O error output file.

[3] Continuations here are in the sense of VM stack frames, not escape procedures as obtained using `call-with-current-continuation`.

`unspecific` ⟶ *unspecific* [procedure]

Returns Scheme48's *unspecific* token, which is used wherever R5RS uses the term 'unspecific' or 'unspecified.' In this manual, the term 'unspecified' is used to mean that the values returned by a particular procedure are not specified and may be anything, including a varying number of values, whereas 'unspecific' refers to Scheme48's specific 'unspecific' value that the `unspecific` procedure returns.

`reduce` *kons knil list* ⟶ *final-knil* [procedure]

Reduces *list* by repeatedly applying *kons* to elements of *list* and the current *knil* value. This is the fundamental list recursion operator.

```
(reduce kons knil
        (cons elt1
              (cons elt2
                    (...(cons eltN '())...))))
    ≡
(kons elt1
      (kons elt2
            (...(kons eltN knil)...)))
```

Example:

```
(reduce append '() '((1 2 3) (4 5 6) (7 8 9)))
    ⇒ (1 2 3 4 5 6 7 8 9)
(append '(1 2 3)
        (append '(4 5 6)
                (append '(7 8 9) '())))
    ⇒ (1 2 3 4 5 6 7 8 9)
```

`fold` *combiner list accumulator* ⟶ *final-accumulator* [procedure]

Folds *list* into an accumulator by repeatedly combining each element into an accumulator with *combiner*. This is the fundamental list iteration operator.

```
(fold combiner
      (list elt1 elt2 ... eltN)
      accumulator)
    ≡
(let* ((accum1 (combiner elt1 accumulator))
       (accum2 (combiner elt2 accum1))
       ...
       (accumN (combiner eltN accumN-1)))
  accumN)
```

Example:

```
(fold cons '() '(a b c d))
    ⇒ (d c b a)
(cons 'd (cons 'c (cons 'b (cons 'a '()))))
    ⇒ (d c b a)
```

`fold->2` *combiner list accumulator*$_1$ *accumulator*$_2$ $\longrightarrow$ [procedure]
[*final-accumulator*$_1$ *final-accumulator*$_2$]
`fold->3` *combiner list accumulator*$_1$ *accumulator*$_2$ *accumulator*$_3$ $\longrightarrow$ [procedure]
[*final-accumulator*$_1$ *final-accumulator*$_2$ *final-accumulator*$_3$]

Variants of `fold` for two and three accumulators, respectively.

```
;;; Partition list by elements that satisfy pred? and those
;;; that do not.

(fold->2 (lambda (elt satisfied unsatisfied)
           (if (pred? elt)
               (values (cons elt satisfied) unsatisfied)
               (values satisfied (cons elt unsatisfied))))
         list
         '() '())
```

`filter` *predicate list* $\longrightarrow$ *filtered-list* [procedure]

Returns a list of all elements in *list* that satisfy *predicate*.

```
(filter odd? '(3 1 4 1 5 9 2 6 5 3 5))
    ⇒ (3 1 1 5 9 5 3 5)
```

`posq` *object list* $\longrightarrow$ *integer or* `#f` [procedure]
`posv` *object list* $\longrightarrow$ *integer or* `#f` [procedure]
`position` *object list* $\longrightarrow$ *integer or* `#f` [procedure]

These find the position of the first element equal to *object* in *list*. `Posq` compares elements by `eq?`; `posv` compares by `eqv?`; `position` compares by `equal?`.

```
(posq 'c '(a b c d e f))
    ⇒ 2
(posv 1/2 '(1 1/2 2 3/2))
    ⇒ 1
(position '(d . e) '((a . b) (b . c) (c . d) (d . e) (e . f)))
    ⇒ 3
```

`any` *predicate list* $\longrightarrow$ *value or* `#f` [procedure]
`every` *predicate list* $\longrightarrow$ *boolean* [procedure]

`Any` returns the value that *predicate* returns for the first element in *list* for which *predicate* returns a true value; if no element of *list* satisfied *predicate*, `any` returns `#f`. `Every` returns `#t` if every element of *list* satisfies *predicate*, or `#f` if there exist any that do not.

```
(any (lambda (x) (and (even? x) (sqrt x)))
     '(0 1 4 9 16))
    ⇒ 2
(every odd? '(1 3 5 7 9))
    ⇒ #t
```

`sublist` *list start end* $\longrightarrow$ *list* [procedure]

Returns a list of the elements in *list* including & after that at the index *start* and before the index *end*.

```
(sublist '(a b c d e f g h i) 3 6)      ⇒ (d e f)
```

`last` *list* $\longrightarrow$ *value* [procedure]

Returns the last element in *list*. `Last`'s effect is undefined if *list* is empty.

```
(last '(a b c))                    ⇒ c
```

`insert` *object list elt<* $\longrightarrow$ *list* [procedure]

Inserts *object* into the sorted list *list*, comparing the order of *object* and each element by *elt<*.

```
(insert 3 '(0 1 2 4 5) <)          ⇒ (0 1 2 3 4 5)
```

4.1.3 Filenames

There are some basic filename manipulation facilities exported by the `filenames` structure.[4]

`*scheme-file-type*` $\longrightarrow$ *symbol* [constant]
`*load-file-type*` $\longrightarrow$ *symbol* [constant]

`*Scheme-file-type*` is a symbol denoting the file extension that Scheme48 assumes for Scheme source files; any other extension, for instance in the filename list of a structure definition, must be written explicitly. `*Load-file-type*` is a symbol denoting the preferable file extension to load files from. (`*Load-file-type*` was used mostly in bootstrapping Scheme48 from Pseudoscheme or T long ago and is no longer very useful.)

`file-name-directory` *filename* $\longrightarrow$ *string* [procedure]
`file-name-nondirectory` *filename* $\longrightarrow$ *string* [procedure]

`File-name-directory` returns the directory component of the filename denoted by the string *filename*, including a trailing separator (on Unix, `/`). `File-name-nondirectory` returns everything but the directory component of the filename denoted by the string *filename*, including the extension.

```
(file-name-directory "/usr/local/lib/scheme48/scheme48.image")
    ⇒ "/usr/local/lib/scheme48/"
(file-name-nondirectory "/usr/local/lib/scheme48/scheme48.image")
    ⇒ "scheme48.image"
(file-name-directory "scheme48.image")
    ⇒ ""
(file-name-nondirectory "scheme48.image")
    ⇒ "scheme48.image"
```

Namelists are platform-independent means by which to name files. They are represented as readable S-expressions of any of the following forms:

basename represents a filename with only a basename and no directory or file type/extension;

(*directory basename* [*type*])
represents a filename with a single preceding directory component and an optional file type/extension; and

[4] The facilities Scheme48 provides are very rudimentary, and they are not intended to act as a coherent and comprehensive pathname or logical name facility such as that of Common Lisp. However, they served the basic needs of Scheme48's build process when they were originally created.

`((`*directory* `...)` *basename* `[`*type*`])`
: represents a filename with a sequence of directory components, a basename, and an optional file type/extension.

Each atomic component — that is, the basename, the type/extension, and each individual directory component — may be either a string or a symbol. Symbols are converted to the canonical case of the host operating system by `namestring` (on Unix, lowercase); the case of string components is not touched.

`namestring` *namelist directory default-type* ⟶ *string* [procedure]
: Converts *namelist* to a string in the format required by the host operating system.[5] If *namelist* did not have a directory component, *directory*, a string in the underlying operating system's format for directory prefixes, is added to the resulting namestring; and, if *namelist* did not have a type/extension, *default-type*, which may be a string or a symbol and which should *not* already contain the host operating system's delimiter (usually a dot), is appended to the resulting namestring.

  Directory or *default-type* may be `#f`, in which case they are not prefixed or appended to the resulting filename.

```
(namestring 'foo #f #f)                 ⇒ "foo"
(namestring 'foo "bar" 'baz)            ⇒ "bar/foo.baz"
(namestring '(rts defenum) "scheme" 'scm)
    ⇒ "scheme/rts/defenum.scm"
(namestring '((foo bar) baz quux) "zot" #f)
    ⇒ "zot/foo/bar/baz.quux"
(namestring "zot/foo/bar/baz.quux" #f "mumble")
    ⇒ "zot/foo/bar/baz.quux.mumble"
```

4.1.3.1 Filename translations

Scheme48 keeps a registry of *filename translations*, translations from filename prefixes to the real prefixes. This allows abstraction of actual directory prefixes without necessitating running Scheme code to construct directory pathnames (for example, in configuration files). Interactively, in the usual command processor, users can set filename translations with the `,translate`; see Section 2.4.1 [Basic commands], page 10.

`translations` ⟶ *string/string-alist* [procedure]
: Returns the alist of filename translations.

`set-translation!` *from to* ⟶ *unspecified* [procedure]
: Adds a filename prefix translation, overwriting an existing one if one already existed.

`translate` *filename* ⟶ *translated-filename* [procedure]
: Translates the first prefix of *filename* found in the registry of translations and returns the translated filename.

```
(set-translation! "s48" "/home/me/scheme/scheme48/scheme")
(translate (namestring '(bcomp frame) "s48" 'scm))
```

[5] However, the current standard distribution of Scheme48 is specific to Unix: the current code implements only Unix filename facilities.

```
    ⇒ "/home/me/scheme/scheme48/scheme/bcomp/frame.scm"
(translate (namestring "comp-packages" "s48" 'scm))
    ⇒ "/home/me/scheme/scheme48/scheme/comp-packages.scm"
(translate "s48/frobozz")
    ⇒ "/home/me/scheme/scheme48/scheme/frobozz"
(set-translation! "scheme48" "s48")
(translate (namestring '((scheme48 big) filename) #f 'scm))
    ⇒ scheme48/big/filename.scm
(translate (translate (namestring '((scheme48 big) filename) #f 'scm)))
    ⇒ "/home/me/scheme/scheme48/scheme/big/filename.scm"
```

One filename translation is built-in, mapping `=scheme48/` to the directory of system files in a Scheme48 installation, which on Unix is typically a directory in `/usr/local/lib`.

```
(translate "=scheme48/scheme48.image")
    ⇒ /usr/local/scheme48/scheme48.image
```

4.1.4 Fluid/dynamic bindings

The `fluids` structure provides a facility for dynamically bound resources, like special variables in Common Lisp, but with first-class, unforgeable objects.

Every thread (see Chapter 5 [Multithreading], page 77) in Scheme48 maintains a *fluid or dynamic environment*. It maps *fluid descriptors* to their values, much like a lexical environment maps names to their values. The dynamic environment is implemented by deep binding and dynamically scoped. Fluid variables are represented as first-class objects for which there is a top-level value and possibly a binding in the current dynamic environment. Escape procedures, as created with Scheme's `call-with-current-continuation`, also store & preserve the dynamic environment at the time of their continuation's capture and restore it when invoked.

The convention for naming variables that are bound to fluid objects is to add a prefix of `$` (dollar sign); *e.g.*, `$foo`.

`make-fluid` *top-level-value* ⟶ *fluid* [procedure]
: Fluid constructor.

`fluid` *fl* ⟶ *value* [procedure]
`set-fluid!` *fl value* ⟶ *unspecified* [procedure]
`fluid-cell-ref` *fluid-cell* ⟶ *value* [procedure]
`fluid-cell-set!` *fluid-cell value* ⟶ *unspecified* [procedure]
: `Fluid` returns the value that the current dynamic environment associates with *fl*, if it has an association; if not, it returns *fl*'s top-level value, as passed to `make-fluid` to create *fl*. `Set-fluid!` assigns the value of the association in the current dynamic environment for *fl* to *value*, or, if there is no such association, it assigns the top-level value of *fl* to *value*. Direct assignment of fluids is deprecated, however, and may be removed in a later release; instead, programmers should use fluids that are bound to mutable cells (see Section 4.1.7 [Cells], page 44). `Fluid-cell-ref` and `fluid-cell-set!` are conveniences for this; they simply call the corresponding cell operations after fetching the cell that the fluid refers to by using `fluid`.

`let-fluid` *fluid value thunk* $\longrightarrow$ *values* [procedure]
`let-fluids` $fluid_0$ $value_0$ $fluid_1$ $value_1$ ... *thunk* $\longrightarrow$ *values* [procedure]

These dynamically bind their fluid arguments to the corresponding value arguments and apply *thunk* with the new dynamic environment, restoring the old one after *thunk* returns and returning the value it returns.

```
(define $mumble (make-fluid 0))

(let ((a (fluid $mumble))
      (b (let-fluid $mumble 1
           (lambda () (fluid $mumble))))
      (c (fluid $mumble))
      (d (let-fluid $mumble 2
           (lambda ()
             (let-fluid $mumble 3
               (lambda () (fluid $mumble)))))))
  (list a b c d))
    ⇒ (0 1 0 3)

(let ((note (lambda (when)
              (display when)
              (display ": ")
              (write (fluid $mumble))
              (newline))))
  (note 'initial)
  (let-fluid $mumble 1 (lambda () (note 'let-fluid)))
  (note 'after-let-fluid)
  (let-fluid $mumble 1
    (lambda ()
      (note 'outer-let-fluid)
      (let-fluid $mumble 2 (lambda () (note 'inner-let-fluid)))))
  (note 'after-inner-let-fluid)
  ((call-with-current-continuation
     (lambda (k)
       (lambda ()
         (let-fluid $mumble 1
           (lambda ()
             (note 'let-fluid-within-cont)
             (let-fluid $mumble 2
               (lambda () (note 'inner-let-fluid-within-cont)))
             (k (lambda () (note 'let-fluid-thrown)))))))))
  (note 'after-throw))
    ⊣ initial: 0
    ⊣ let-fluid: 1
    ⊣ after-let-fluid: 0
    ⊣ outer-let-fluid: 1
    ⊣ inner-let-fluid: 2
```

```
⊣ let-fluid-within-cont: 1
⊣ inner-let-fluid-within-cont: 2
⊣ let-fluid-thrown: 0
⊣ after-throw: 0
```

4.1.5 ASCII character encoding

These names are exported by the `ascii` structure.

`char->ascii` *char* ⟶ *ascii-integer* [procedure]
`ascii->char` *ascii-integer* ⟶ *character* [procedure]

These convert characters to and from their integer ASCII encodings. `Char->ascii` and `ascii->char` are similar to R5RS's `char->integer` and `integer->char`, but they are guaranteed to use the ASCII encoding. Scheme48's `integer->char` and `char->integer` deliberately do not use the ASCII encoding to encourage programmers to make use of only what R5RS guarantees.

```
(char->ascii #\a)                ⇒ 97
(ascii->char 97)                 ⇒ #\a
```

`ascii-limit` ⟶ *integer* [constant]
`ascii-whitespaces` ⟶ *ascii-integer-list* [constant]

`Ascii-limit` is an integer that is one greater than the highest number that `char->ascii` may return or `ascii->char` will accept. `Ascii-whitespaces` is a list of the integer encodings of all characters that are considered whitespace: space (32), horizontal tab (9), line-feed/newline (10), vertical tab (11), form-feed/page (12), and carriage return (13).

4.1.6 Integer enumerations

Scheme48 provides a facility for *integer enumerations*, somewhat akin to C enums. The names described in this section are exported by the `enumerated` structure.

Note: These enumerations are *not* compatible with the enumerated/finite type facility (see Section 6.2 [Enumerated/finite types and sets], page 96).

`define-enumeration` *enumeration-name* (*enumerand-name* ...) [syntax]

Defines *enumeration-name* to be a static enumeration. (Note that it is *not* a regular variable. It is actually a macro, though its exact syntax is not exposed; it must be exported with the `:syntax` type (see Section 3.4 [Static type system], page 30).) *Enumeration-name* thereafter may be used with the enumeration operators described below.

`enum` *enumeration-name enumerand-name* ⟶ *enumerand-integer* [syntax]
`components` *enumeration-name* ⟶ *component-vector* [syntax]

`Enum` expands to the integer value represented symbolically by *enumerand-name* in the enumeration *enumeration-name* as defined by `define-enumeration`. `Components` expands to a literal vector of the components in *enumeration-name* as defined by `define-enumeration`. In both cases, *enumerand-name* must be written literally as the name of the enumerand; see `name->enumerand` for extracting an enumerand's integer given a run-time symbol naming an enumerand.

`enumerand->name` *enumerand-integer enumeration-name* ⟶ *symbol* [syntax]
`name->enumerand` *enumerand-name enumeration-name* ⟶ [syntax]
integer-enumerand

`Enumerand->name` expands to a form that evaluates to the symbolic name that the integer value of the expression *enumerand-integer* is mapped to by *enumeration-name* as defined by `define-enumeration`. `Name->enumerand` expands to a form that evaluates to the integer value of the enumerand in *enumeration-name* that is represented symbolically by the value of the expression *enumerand-name*.

The `enum-case` structure provides a handy utility of the same name for dispatching on enumerands.

`enum-case` [syntax]

```
(enum-case enumeration-name key
  ((enumerand-name ...) body)
  ...
  [(else else-body)])
```

Matches *key* with the clause one of whose names maps in *enumeration-name* to the integer value of *key*. *Key* must be an exact, non-negative integer. If no matching clause is found, and *else-body* is present, `enum-case` will evaluate *else-body*; if *else-body* is not present, `enum-case` will return an unspecific value.

Examples:

```
(define-enumeration foo
  (bar
   baz))

(enum foo bar)                          ⇒ 0
(enum foo baz)                          ⇒ 1

(enum-case foo (enum foo bar)
  ((baz) 'x)
  (else  'y))
    ⇒ y

(enum-case foo (enum foo baz)
  ((bar) 'a)
  ((baz) 'b))
    ⇒ b

(enumerand->name 1 foo)                 ⇒ baz
(name->enumerand 'bar foo)              ⇒ 0
(components foo)                        ⇒ #(bar baz)
```

4.1.7 Cells

Scheme48 also provides a simple mutable cell data type from the `cells` structure. It uses them internally for local, lexical variables that are assigned, but cells are available still to the rest of the system for general use.

`make-cell` *contents* ⟶ *cell* [procedure]
`cell?` *object* ⟶ *boolean* [procedure]
`cell-ref` *cell* ⟶ *value* [procedure]
`cell-set!` *cell value* ⟶ *unspecified* [procedure]

`Make-cell` creates a new cell with the given contents. `Cell?` is the disjoint type predicate for cells. `Cell-ref` returns the current contents of *cell*. `Cell-set!` assigns the contents of *cell* to *value*.

Examples:

```
(define cell (make-cell 42))
(cell-ref cell)                         ⇒ 42
(cell? cell)                            ⇒ #t
(cell-set! cell 'frobozz)
(cell-ref cell)                         ⇒ frobozz
```

4.1.8 Queues

The `queues` structure exports names for procedures that operate on simple first-in, first-out queues.

`make-queue` ⟶ *queue* [procedure]
`queue?` *object* ⟶ *boolean* [procedure]

`Make-queue` constructs an empty queue. `Queue?` is the disjoint type predicate for queues.

`queue-empty?` *queue* ⟶ *boolean* [procedure]
`empty-queue!` *queue* ⟶ *unspecified* [procedure]

`Queue-empty?` returns `#t` if *queue* contains zero elements or `#f` if it contains some. `Empty-queue!` removes all elements from *queue*.

`enqueue!` *queue object* ⟶ *unspecified* [procedure]
`dequeue!` *queue* ⟶ *value* [procedure]
`maybe-dequeue!` *queue* ⟶ *value or* `#f` [procedure]
`queue-head` *queue* ⟶ *value* [procedure]

`Enqueue!` adds *object* to *queue*. `Dequeue!` removes & returns the next object available from *queue*; if *queue* is empty, `dequeue!` signals an error. `Maybe-dequeue!` is like `dequeue!`, but it returns `#f` in the case of an absence of any element, rather than signalling an error. `Queue-head` returns the next element available from *queue* without removing it, or it signals an error if *queue* is empty.

`queue-length` *queue* ⟶ *integer* [procedure]

Returns the number of objects in *queue*.

`on-queue?` *queue object* ⟶ *boolean* [procedure]
`delete-from-queue!` *queue object* ⟶ *unspecified* [procedure]

`On-queue?` returns true if *queue* contains *object* or `#f` if not. `Delete-from-queue!` removes the first occurrence of *object* from *queue* that would be dequeued.

`queue->list` *queue* ⟶ *list* [procedure]
`list->queue` *list* ⟶ *queue* [procedure]

These convert queues to and from lists of their elements. `Queue->list` returns a list in the order in which its elements were added to the queue. `List->queue` returns a queue that will produce elements starting at the head of the list.

Examples:

```
(define q (make-queue))
(enqueue! q 'foo)
(enqueue! q 'bar)
(queue->list q)                          ⇒ (foo bar)
(on-queue? q 'bar)                       ⇒ #t
(dequeue! q)                             ⇒ 'foo
(queue-empty? q)                         ⇒ #f
(delete-from-queue! queue 'bar)
(queue-empty? q)                         ⇒ #t
(enqueue! q 'frobozz)
(empty-queue! q)
(queue-empty? q)                         ⇒ #t
(dequeue! q)                             error  empty queue
```

Queues are integrated with Scheme48's optimistic concurrency (see Section 5.2 [Optimistic concurrency], page 77) facilities, in that every procedure exported except for `queue->list` ensures fusible atomicity in operation — that is, every operation except for `queue->list` ensures that the transaction it performs is atomic, and that it may be fused within larger atomic transactions, as transactions wrapped within `call-ensuring-atomicity` *&c.* may be.

4.1.9 Hash tables

Scheme48 provides a simple hash table facility in the structure `tables`.

`make-table` [*hasher*] ⟶ *table* [procedure]
`make-string-table` ⟶ *string-table* [procedure]
`make-symbol-table` ⟶ *symbol-table* [procedure]
`make-integer-table` ⟶ *integer-table* [procedure]

Hash table constructors. `Make-table` creates a table that hashes keys either with *hasher*, if it is passed to `make-table`, or `default-hash-function`, and it compares keys for equality with `eq?`, unless they are numbers, in which case it compares with `eqv?`. `Make-string-table` makes a table whose hash function is `string-hash` and that compares the equality of keys with `string=?`. `Make-symbol-table` constructs a table that hashes symbol keys by converting them to strings and hashing them with `string-hash`; it compares keys' equality by `eq?`. Tables made by `make-integer-table` hash keys by taking their absolute value, and test for key equality with the `=` procedure.

`make-table-maker` *comparator hasher* ⟶ *table-maker* [procedure]

Customized table constructor constructor: this returns a nullary procedure that creates a new table that uses *comparator* to compare keys for equality and *hasher* to hash keys.

`table?` *object* ⟶ *boolean* [procedure]
Hash table disjoint type predicate.

`table-ref` *table key* ⟶ *value or* `#f` [procedure]
`table-set!` *table key value* ⟶ *unspecified* [procedure]
`Table-ref` returns the value associated with *key* in *table*, or `#f` if there is no such association. If *value* is `#f`, `table-set!` ensures that there is no longer an association with *key* in *table*; if *value* is any other value, `table-set!` creates a new association or assigns an existing one in *table* whose key is *key* and whose associated value is *value*.

`table-walk` *proc table* ⟶ *unspecified* [procedure]
`Table-walk` applies *proc* to the key & value, in that order of arguments, of every association in *table*.

`make-table-immutable!` *table* ⟶ *table* [procedure]
This makes the structure of *table* immutable, though not its contents. `Table-set!` may not be used with tables that have been made immutable.

`default-hash-function` *value* ⟶ *integer-hash-code* [procedure]
`string-hash` *string* ⟶ *integer-hash-code* [procedure]
Two built-in hashing functions. `Default-hash-function` can hash any Scheme value that could usefully be used in a `case` clause. `String-hash` is likely to be fast, as it is implemented as a VM primitive. `String-hash` is the same as what the `features` structure exports under the same name.

4.1.10 Weak references

Scheme48 provides an interface to weakly held references in basic weak pointers and *populations*, or sets whose elements are weakly held. The facility is in the structure `weak`.

4.1.10.1 Weak pointers

`make-weak-pointer` *contents* ⟶ *weak-pointer* [procedure]
`weak-pointer?` *object* ⟶ *boolean* [procedure]
`weak-pointer-ref` *weak-pointer* ⟶ *value or* `#f` [procedure]
`Make-weak-pointer` creates a weak pointer that points to *contents*. `Weak-pointer?` is the weak pointer disjoint type predicate. `Weak-pointer-ref` accesses the value contained within `weak-pointer`, or returns `#f` if there were no strong references to the contents and a garbage collection occurred. Weak pointers resemble cells (see Section 4.1.7 [Cells], page 44), except that they are immutable and hold their contents weakly, not strongly.

4.1.10.2 Populations (weak sets)

`make-population` ⟶ *population* [procedure]
`add-to-population!` *object population* ⟶ *unspecified* [procedure]
`population->list` *population* ⟶ *list* [procedure]
`walk-population` *proc population* ⟶ *unspecified* [procedure]
`Make-population` constructs an empty population. `Add-to-population!` adds *object* to the population *population*. `Population->list` returns a list of the elements

of *population*. Note, though, that this can be dangerous in that it can create strong references to the population's contents and potentially leak space because of this. `Walk-population` applies *proc* to every element in *population*.

4.1.11 Type annotations

Scheme48 allows optional type annotations with the `loophole` special form from the `loopholes` structure.

`loophole` *type expression* $\longrightarrow$ *values* [syntax]
This is exactly equivalent in semantics to *expression*, except the static type analyzer is informed that the whole expression has the type *type*. For details on the form of *type*, see Section 3.4 [Static type system], page 30.

Type annotations can be used for several different purposes:

- simply to give more information to the static type analyzer;
- to work as a simple abstract data type facility: passing a type name that does not already exist creates a new disjoint value type; and
- to prevent the type system from generating warnings in the rare cases where it would do so incorrectly, such as in the `primitive-cwcc`, `primitive-catch`, and `with-continuation` devices (to be documented in a later edition of this manual).

To see an example of the second use, see `rts/jar-defrecord.scm` in Scheme48's source tree.

Note: Type annotations do *not* damage the safety of Scheme's type system. They affect only the static type analyzer, which does not change run-time object representations; it only checks type soundness of code and generates warnings for programs that would cause run-time type errors.

4.1.12 Explicit renaming macros

Scheme48 supports a simple low-level macro system based on explicitly renaming identifiers to preserve hygiene. The macro system is well-integrated with the module system; see Section 3.3 [Macros in concert with modules], page 28.

Explicit renaming macro transformers operate on simple S-expressions extended with *identifiers*, which are like symbols but contain more information about lexical context. In order to preserve that lexical context, transformers must explicitly call a *renamer* procedure to produce an identifier with the proper scope. To test whether identifiers have the same denotation, transformers are also given an identifier comparator.

The facility provided by Scheme48 is almost identical to the explicit renaming macro facility described in [Clinger 91].[6] It differs only by the `transformer` keyword, which is described in the paper but not used by Scheme48, and in the annotation of auxiliary names.

`define-syntax` *name transformer* [*aux-names*] [syntax]
Introduces a derived syntax *name* with the given transformer, which may be an explicit renaming transformer procedure, a pair whose car is such a procedure and

[6] For the sake of avoiding any potential copyright issues, the paper is not duplicated here, and instead the author of this manual has written the entirety of this section.

whose cdr is a list of auxiliary identifiers, or the value of a `syntax-rules` expression. In the first case, the added operand *aux-names* may, and usually should except in the case of local (non-exported) syntactic bindings, be a list of all of the auxiliary top-level identifiers used by the macro.

Explicit renaming transformer procedures are procedures of three arguments: an input form, an identifier renamer procedure, and an identifier comparator procedure. The input form is the whole form of the macro's invocation (including, at the car, the identifier whose denotation was the syntactic binding). The identifier renamer accepts an identifier as an argument and returns an identifier that is hygienically renamed to refer absolutely to the identifier's denotation in the environment of the macro's definition, not in the environment of the macro's usage. In order to preserve hygiene of syntactic transformations, macro transformers must call this renamer procedure for any literal identifiers in the output. The renamer procedure is referentially transparent; that is, two invocations of it with the same arguments in terms of `eq?` will produce the same results in the sense of `eq?`.

For example, this simple transformer for a `swap!` macro is incorrect:

```
(define-syntax swap!
  (lambda (form rename compare)
    (let ((a (cadr  form))
          (b (caddr form)))
      `(LET ((TEMP ,a))
         (SET! ,a ,b)
         (SET! ,b TEMP)))))
```

The introduction of the literal identifier `temp` into the output may conflict with one of the input variables if it were to also be named `temp`: `(swap! temp foo)` or `(swap! bar temp)` would produce the wrong result. Also, the macro would fail in another very strange way if the user were to have a local variable named `let` or `set!`, or it would simply produce invalid output if there were no binding of `let` or `set!` in the environment in which the macro was used. These are basic problems of abstraction: the user of the macro should not need to know how the macro is internally implemented, notably with a `temp` variable and using the `let` and `set!` special forms.

Instead, the macro must hygienically rename these identifiers using the renamer procedure it is given, and it should list the top-level identifiers it renames (which cannot otherwise be extracted automatically from the macro's definition):

```
(define-syntax swap!
  (lambda (form rename compare)
    (let ((a (cadr  form))
          (b (caddr form)))
      `(,(rename 'LET) ((,(rename 'TEMP) ,a))
         (,(rename 'SET!) ,a ,b)
         (,(rename 'SET!) ,b ,(rename 'TEMP)))))
  (LET SET!))
```

However, some macros are unhygienic by design, *i.e.* they insert identifiers into the output intended to be used in the environment of the macro's usage. For example, consider a `loop` macro that loops endlessly, but binds a variable named `exit` to an escape procedure

to the continuation of the `loop` expression, with which the user of the macro can escape the loop:

```
(define-syntax loop
  (lambda (form rename compare)
    (let ((body (cdr form)))
      `(,(rename 'CALL-WITH-CURRENT-CONTINUATION)
        (,(rename 'LAMBDA) (EXIT)      ; Literal, unrenamed EXIT.
          (,(rename 'LET) ,(rename 'LOOP) ()
            ,@body
            (,(rename 'LOOP)))))))
  (CALL-WITH-CURRENT-CONTINUATION LAMBDA LET))
```

Note that macros that expand to `loop` must also be unhygienic; for instance, this naïve definition of a `loop-while` macro is incorrect, because it hygienically renames `exit` automatically by of the definition of `syntax-rules`, so the identifier it refers to is not the one introduced unhygienically by `loop`:

```
(define-syntax loop-while
  (syntax-rules ()
    ((LOOP-WHILE test body ...)
     (LOOP (IF (NOT test)
               (EXIT))                  ; Hygienically renamed.
           body ...))))
```

Instead, a transformer must be written to not hygienically rename `exit` in the output:

```
(define-syntax loop-while
  (lambda (form rename compare)
    (let ((test (cadr form))
          (body (cddr form)))
      `(,(rename 'LOOP)
        (,(rename 'IF) (,(rename 'NOT) ,test)
          (EXIT))                       ; Not hygienically renamed.
        ,@body)))
  (LOOP IF NOT))
```

To understand the necessity of annotating macros with the list of auxiliary names they use, consider the following definition of the `delay` form, which transforms `(delay` *exp*`)` into `(make-promise (lambda ()` *exp*`))`, where `make-promise` is some non-exported procedure defined in the same module as the `delay` macro:

```
(define-syntax delay
  (lambda (form rename compare)
    (let ((exp (cadr form)))
      `(,(rename 'MAKE-PROMISE) (,(rename 'LAMBDA) () ,exp)))))
```

This preserves hygiene as necessary, but, while the compiler can know whether `make-promise` is *exported* or not, it cannot in general determine whether `make-promise` is *local*, *i.e.* not accessible in any way whatsoever, even in macro output, from any other modules. In this case, `make-promise` is *not* local, but the compiler cannot in general know this, and it would be an unnecessarily heavy burden on the compiler, the linker, and related code-processing systems to assume that all bindings are not local. It is

therefore better[7] to annotate such definitions with the list of auxiliary names used by the transformer:

```
(define-syntax delay
  (lambda (form rename compare)
    (let ((exp (cadr form)))
      `(,(rename 'MAKE-PROMISE) (,(rename 'LAMBDA) () ,exp))))
  (MAKE-PROMISE LAMBDA))
```

4.2 Condition system

As of version 1.3 (different from all older versions), Scheme48 supports two different condition systems. One of them, the original one, is a simple system where conditions are represented as tagged lists. This section documents the original one. The new condition system is [SRFI 34, 35], and there is a complicated translation layer between the old one, employed by the run-time system, and the new one, which is implemented in a layer high above that as a library, but a library which is always loaded in the usual development environment. See the [SRFI 34, 35] documents for documentation of the new condition system. [SRFI 34] is available from the `exceptions` structure; SRFI 35, from the `conditions` structure.

Note: The condition system changed in Scheme48 version 1.3. While the old one is still available, the names of the structures that implement it changed. `Signals` is now `simple-signals`, and `conditions` is now `simple-conditions`. The structure that `signals` *now* names implements the same interface, but with [SRFI 34, 35] underlying it. The structure that the name `conditions` *now* identifies [SRFI 35]. You will have to update all old code that relied on the old `signals` and `conditions` structure either by using those structures' new names or by invasively modifying all code to use [SRFI 34, 35]. Also, the only way to completely elide the use of the SRFIs is to evaluate this in an environment with the `exceptions-internal` and `vm-exceptions` structure open:

```
(begin (initialize-vm-exceptions! really-signal-condition)
       ;; INITIALIZE-VM-EXCEPTIONS! returns a very large object,
       ;; which we probably don't want printed at the REPL.
       #t)
```

4.2.1 Signalling, handling, and representing conditions

Scheme48 provides a simple condition system.[8] *Conditions* are objects that describe exceptional situations. Scheme48 keeps a registry of *condition types*, which just have references to their supertypes. Conditions are simple objects that contain only two fields, the type and the type-specific data (the *stuff*). Accessor procedures should be defined for particular condition types to extract the data contained within the 'stuff' fields of instances of of those condition types. Condition types are represented as symbols. *Condition handlers* are part of the system's dynamic context; they are used to handle exceptional situations when conditions are signalled that describe such exceptional situations. *Signalling* a condition

[7] However, the current compiler in Scheme48 does not require this, though the static linker does.

[8] Note, however, that Scheme48's condition system is likely to be superseded in the near future by [SRFI 34, SRFI 35].

signals that an exceptional situation occurred and invokes the current condition handler on the condition.

Scheme48's condition system is split up into three structures:

`simple-signals`
: Exports procedures to signal conditions and construct conditions, as well as some utilities for common kinds of conditions.

`handle`
: Exports facilities for handling signalled conditions.

`simple-conditions`
: The system of representing conditions as objects.

The `simple-signals` structure exports these procedures:

`make-condition` *type-name stuff* $\longrightarrow$ *condition* [procedure]
: The condition object constructor.

`signal-condition` *condition* $\longrightarrow$ *values* (*may not return*) [procedure]
`signal` *type-name stuff* ... $\longrightarrow$ *values* (*may not return*) [procedure]
: `Signal-condition` signals the given condition. `Signal` is a convenience atop the common conjunction of `signal-condition` and `make-condition`: it constructs a condition with the given type name and stuff, whereafter it signals that condition with `signal-condition`.

`error` *message irritant* ... $\longrightarrow$ *values* (*may not return*) [procedure]
`warn` *message irritant* ... $\longrightarrow$ *values* (*may not return*) [procedure]
`syntax-error` *message irritant* ... $\longrightarrow$ *expression* (*may not return*) [procedure]
`call-error` *message irritant* ... $\longrightarrow$ *values* (*may not return*) [procedure]
`note` *message irritant* ... $\longrightarrow$ *values* (*may not return*) [procedure]
: Conveniences for signalling standard condition types. These procedures generally either do not return or return an unspecified value, unless specified to by a user of the debugger. `Syntax-error` returns the expression `(quote syntax-error)`, if the condition handler returns to `syntax-error` in the first place.

  By convention, the message should be lowercased (*i.e.* the first word should not be capitalized), and it should not end with punctuation. The message is typically not a complete sentence. For example, these all follow Scheme48's convention:

  argument type error

  wrong number of arguments

  invalid syntax

  ill-typed right-hand side

  out of memory, unable to continue

  These, on the other hand, do not follow the convention and should be avoided:

  Argument type error:

  An argument of the wrong type was passed.

  possible type mismatch:

  Luser is an idiot!

Elaboration on a message is performed usually by wrapping an irritant in a descriptive list. For example, one might write:

```
(error "invalid argument"
       '(not a pair)
       `(while calling ,frobbotz)
       `(received ,object))
```

This might be printed as:

```
Error: invalid argument
       (not a pair)
       (while calling #{Procedure 123 (frobbotz in ...)})
       (received #(a b c d))
```

The `handle` structure exports the following procedures:

`with-handler` *handler thunk* $\longrightarrow$ *values* [procedure]

Sets up *handler* as the condition handler for the dynamic extent of *thunk*. *Handler* should be a procedure of two arguments: the condition that was signalled and a procedure of zero arguments that propagates the condition up to the next dynamically enclosing handler. When a condition is signalled, *handler* is tail-called from the point that the condition was signalled at. Note that, because *handler* is tail-called at that point, it will *return* to that point also.

Warning: `With-handler` is potentially very dangerous. If an exception occurs and a condition is raised in the handler, the handler itself will be called with that new condition! Furthermore, the handler may accidentally return to an unexpecting signaller, which can cause very confusing errors. Be careful with `with-handler`; to be perfectly safe, it might be a good idea to throw back out to where the handler was initially installed before doing anything:

```
((call-with-current-continuation
   (lambda (k)
     (lambda ()
       (with-handler (lambda (c propagate)
                       (k (lambda () handler body)))
         (lambda () body))))))
```

`ignore-errors` *thunk* $\longrightarrow$ *values or condition* [procedure]
`report-errors-as-warnings` *thunk message irritant ...* $\longrightarrow$ *values* [procedure]

`Ignore-errors` sets up a condition handler that will return error conditions to the point where `ignore-errors` was called, and propagate all other conditions. If no condition is signalled during the dynamic extent of *thunk*, `ignore-errors` simply returns whatever *thunk* returned. `Report-errors-as-warnings` downgrades errors to warnings while executing *thunk*. If an error occurs, a warning is signalled with the given message, and a list of irritants constructed by adding the error condition to the end of the list *irritant* ....

Finally, the `simple-conditions` structure defines the condition type system. (Note that conditions themselves are constructed only by `make-condition` (and `signal`) from the `simple-signals` structure.) Conditions are very basic values that have only two universally

defined fields: the type and the stuff. The type is a symbol denoting a condition type. The type is specified in the first argument to `make-condition` or `signal`. The stuff field contains whatever a particular condition type stores in conditions of that type. The stuff field is always a list; it is created from the arguments after the first to `make-condition` or `signal`. Condition types are denoted by symbols, kept in a global registry that maps condition type names to their supertype names.

`define-condition-type` *name supertype-names* ⟶ *unspecified* [procedure]
Registers the symbol *name* as a condition type. Its supertypes are named in the list *supertype-names*.

`condition-predicate` *ctype-name* ⟶ *predicate* [procedure]
Returns a procedure of one argument that returns `#t` if that argument is a condition whose type's name is *ctype-name* or `#f` if not.

`condition-type` *condition* ⟶ *type-name* [procedure]
`condition-stuff` *condition* ⟶ *list* [procedure]
Accessors for the two immutable fields of conditions.

`error?` *condition* ⟶ *boolean* [procedure]
`warning?` *condition* ⟶ *boolean* [procedure]
`note?` *condition* ⟶ *boolean* [procedure]
`syntax-error?` *condition* ⟶ *boolean* [procedure]
`call-error?` *condition* ⟶ *boolean* [procedure]
`read-error?` *condition* ⟶ *boolean* [procedure]
`interrupt?` *condition* ⟶ *boolean* [procedure]
Condition predicates for built-in condition types.

`make-exception` *opcode reason arguments* ⟶ *exception* [procedure]
`exception?` *condition* ⟶ *boolean* [procedure]
`exception-opcode` *exception* ⟶ *integer-opcode* [procedure]
`exception-reason` *exception* ⟶ *symbol* [procedure]
`exception-arguments` *exception* ⟶ *list* [procedure]
Exceptions represent run-time errors in the Scheme48 VM. They contain information about what opcode the VM was executing when it happened, what the reason for the exception occurring was, and the relevant arguments.

4.2.2 Displaying conditions

The `display-conditions` structure is also relevant in this section.

`display-condition` *condition port* ⟶ *unspecified* [procedure]
Prints *condition* to *port* for a user to read. For example:

```
(display-condition (make-condition 'error
                                   "Foo bar baz"
                                   'quux
                                   '(zot mumble: frotz))
                   (current-output-port))
⊣ Error: Foo bar baz
⊣        quux
⊣        (zot mumble: frotz)
```

`&disclose-condition` *condition* ⟶ *disclosed* [method table]
Method table (see Section 4.4 [Generic dispatch system], page 56) for a generic procedure (not exposed) used to translate a condition object into a more readable format. See Section 4.6.2 [Writer], page 70.

`limited-write` *object port max-depth max-length* ⟶ *unspecified* [procedure]
A utility for avoiding excessive output: prints *object* to *port*, but will never print more than *max-length* of a subobject's components, leaving a `---` after the last component, and won't recur further down the object graph from the vertex *object* beyond *max-depth*, instead printing an octothorpe (`#`).

```
(let ((x (cons #f #f)))
  (set-car! x x)
  (set-cdr! x x)
  (limited-write x (current-output-port) 2 2))
    ⊣ ((# # ---) (# # ---) ---)
```

4.3 Bitwise manipulation

Scheme48 provides two structures for bit manipulation: bitwise integer operations, the `bitwise` structure, and homogeneous vectors of bytes (integers between 0 and 255, inclusive), the `byte-vectors` structure.

4.3.1 Bitwise integer operations

The `bitwise` structure exports these procedures:

`bitwise-and` *integer ...* ⟶ *integer* [procedure]
`bitwise-ior` *integer ...* ⟶ *integer* [procedure]
`bitwise-xor` *integer ...* ⟶ *integer* [procedure]
`bitwise-not` *integer* ⟶ *integer* [procedure]
Basic twos-complement bitwise boolean logic operations.

`arithmetic-shift` *integer count* ⟶ *integer* [procedure]
Shifts *integer* by the given bit count. If *count* is positive, the shift is a left shift; otherwise, it is a right shift. `Arithmetic-shift` preserves *integer*'s sign.

`bit-count` *integer* ⟶ *integer* [procedure]
Returns the number of bits that are set in *integer*. If *integer* is negative, it is flipped by the bitwise NOT operation before counting.

```
(bit-count #b11010010)                    ⇒ 4
```

4.3.2 Byte vectors

The structure `byte-vectors` exports analogues of regular vector procedures for *byte vectors*, homogeneous vectors of bytes:

`make-byte-vector` *length fill* ⟶ *byte-vector* [procedure]
`byte-vector` *byte ...* ⟶ *byte-vector* [procedure]
`byte-vector?` *object* ⟶ *boolean* [procedure]
`byte-vector-length` *byte-vector* ⟶ *integer* [procedure]

`byte-vector-ref` *byte-vector index* ⟶ *byte* [procedure]
`byte-vector-set!` *byte-vector index byte* ⟶ *unspecified* [procedure]
Fill and each *byte* must be bytes, *i.e.* integers within the inclusive range 0 to 255. Note that `make-byte-vector` is not an exact analogue of `make-vector`, because the *fill* parameter is required.

Old versions of Scheme48 referred to byte vectors as 'code vectors' (since they were used to denote byte code). The `code-vectors` structure exports `make-code-vector`, `code-vector?`, `code-vector-length`, `code-vector-ref`, and `code-vector-set!`, identical to the analogously named byte vector operations.

4.4 Generic dispatch system

Scheme48 supports a CLOS-style generic procedure dispatch system, based on type predicates. The main interface is exported by `methods`. The internals of the system are exposed by the `meta-methods` structure, but they are not documented here. The generic dispatch system is used in Scheme48's writer (see Section 4.6.2 [Writer], page 70) and numeric system.

Types in Scheme48's generic dispatch system are represented using type predicates, rather than having every object have a single, well-defined 'class.' The naming convention for simple types is to prefix the type name with a colon. The types support multiple inheritance. Method specificity is determined based on descending order of argument importance. That is, given two methods, M & N, such that they are both applicable to a given sequence of arguments, and an index i into that sequence, such that i is the first index in M's & N's lists of argument type specifiers, from left to right, where the type differs: if the type for M's argument at i is more specific than the corresponding type in N's specifiers, M is considered to be more specific than N, even if the remaining argument type specifiers in N are more specific.

`define-simple-type` *name* (*supertype* ...) *predicate* [syntax]
Defines *name* to be a *simple type* with the given predicate and the given supertypes.

`singleton` *value* ⟶ *simple-type* [procedure]
Creates a *singleton type* that matches only *value*.

`define-generic` *proc-name method-table-name* [*prototype*] [syntax]
Defines *proc-name* to be a *generic procedure* that, when invoked, will dispatch on its arguments via the *method table* that *method-table-name* is defined to be and apply the most specific method it can determine defined in the *method-table-name* method table to its arguments. The convention for naming variables that will be bound to method tables is to add an ampersand to the front of the name. *Prototype* is a suggestion for what method prototypes should follow the shape of, but it is currently ignored.

`define-method` *method-table prototype body* [syntax]
Adds a *method* to *method-table*, which is usually one defined by `define-generic`.[9] *Prototype* should be a list whose elements may be either identifiers, in which case

[9] There is an internal interface, a sort of meta-object protocol, to the method dispatch system, but it is not yet documented.

that parameter is not used for dispatching, or lists of two elements, the `car` of which is the parameter name and the `cadr` of which should evaluate to the type on which to dispatch. As in many generic dispatch systems of similar designs, methods may invoke the next-most-specific method. By default, the name `next-method` is bound in *body* to a nullary procedure that calls the next-most-specific method. The name of this procedure may be specified by the user by putting the sequence `"next"` *`next-method-name`* in *prototype*, in which case it will be *next-method-name* that is bound to that procedure. For example:

```
(define-method &frob ((foo :bar) "next" frobozz)
  (if (mumble? foo)
      (frobozz)     ; Invoke the next method.
      (yargh blargle foo)))
```

A number of simple types are already defined & exported by the `methods` structure. Entries are listed as *`type-name`* `<-` (*`supertype ...`*), *`predicate`*

- `:values <- (), (lambda (x) #t)` — Abstract supertype of all run-time values
- `:value <- (:values), (lambda (x) #t)` — Abstract supertype of all first-class values
- `:zero <- (:values), (lambda (x) #f)` — Type that no objects satisfy
- `:number <- (:value), number?`
- `:complex <- (:number), complex?` — (This happens to be equivalent to `:number`.)
- `:real <- (:complex), real?`
- `:rational <- (:real), rational?`
- `:integer <- (:rational), integer?`
- `:exact-integer <- (:integer), (lambda (x) (and (integer? x) (exact? x)))`
- `:boolean <- (:value), boolean?`
- `:symbol <- (:value), symbol?`
- `:char <- (:value), char?`
- `:null <- (:value), null?`
- `:pair <- (:value), pair?`
- `:vector <- (:value), vector?`
- `:string <- (:value), string?`
- `:procedure <- (:value), procedure?`
- `:input-port <- (:value), input-port?`
- `:output-port <- (:value), output-port?`
- `:eof-object <- (:value), eof-object?`
- `:record <- (:value), record?`

4.5 I/O system

Scheme48 supports a sophisticated, non-blocking, user-extensible I/O system untied to any particular operating system's I/O facilities. It is based in three levels: channels, ports, and the facilities already built with both ports and channels in Scheme48, such as buffering.

4.5.1 Ports

While channels provide the low-level interface directly to the OS's I/O facilities, *ports* provide a more abstract & generalized mechanism for I/O transmission. Rather than being specific to channels or being themselves primitive I/O devices, ports are functionally parameterized. This section describes the usual I/O operations on ports. The next section describes the programmatic port parameterization mechanism, and the section following that describes the most commonly used built-in port abstraction, ports atop channels.

4.5.1.1 Port operations

The following names are exported by the `i/o` structure.

`input-port?` *value* ⟶ *boolean* [procedure]
`output-port?` *value* ⟶ *boolean* [procedure]
These return `#t` if their argument is both a port and either an input port or output port, respectively, or `#f` if neither condition is true.

`close-input-port` *port* ⟶ *unspecified* [procedure]
`close-output-port` *port* ⟶ *unspecified* [procedure]
Closes *port*, which must be an input port or an output port, respectively.

`char-ready?` [*port*] ⟶ *boolean* [procedure]
`output-port-ready?` *port* ⟶ *boolean* [procedure]
`Char-ready?` returns a true value if there is a character ready to be read from *port* and `#f` if there is no character ready. *Port* defaults to the current input port if absent; see below on current ports. `Output-port-ready?` returns a true value if *port* is ready to receive a single written character and `#f` if not.

`read-block` *block start count port* [*wait?*] ⟶ *count-read or EOF* [procedure]
`write-block` *block start count port* ⟶ *count-written* [procedure]
`write-string` *string port* ⟶ *char-count-written* [procedure]
`Read-block` attempts to read *count* elements from *port* into *block*, which may be a string or a byte vector, starting at *start*. If fewer than *count* characters or bytes are available to read from *port*, and *wait?* is a true value or absent, `read-block` will wait until *count* characters are available and read into *block*; if *wait?* is `#f`, `read-block` immediately returns. `Read-block` returns the number of elements read into *block*, or an end of file object if the stream's end is immediately encountered. `Write-block` writes *count* elements from *block*, which may be a string or a byte vector, starting at *start* to *port*. `Write-string` is a convenience atop `write-block` for writing the entirety of a string to a port.

`newline` [*port*] ⟶ *unspecified* [procedure]
Writes a newline character or character sequence to the output port *port*. *Port* defaults to the current output port; see below on current ports.

`disclose-port` *port* ⟶ *disclosed* [procedure]
Returns a disclosed representation of *port*; see Section 4.6.2 [Writer], page 70.

`force-output` *port* ⟶ *unspecified* [procedure]
Forces all buffered output in the output port *port* to be sent.

`make-null-output-port` ⟶ *output-port* [procedure]
Returns an output port that will ignore any output it receives.

4.5.1.2 Current ports

Scheme48 keeps in its dynamic environment (see Section 4.1.4 [Fluid/dynamic bindings], page 41) a set of 'current' ports. These include R5RS's current input and output ports, as well as ports for general noise produced by the system, and ports for where error messages are printed. These procedures are exported by the `i/o` structure.

`current-input-port` ⟶ *input-port* [procedure]
`current-output-port` ⟶ *output-port* [procedure]
`current-noise-port` ⟶ *output-port* [procedure]
`current-error-port` ⟶ *output-port* [procedure]
These return the values in the current dynamic environment of the respective ports. `Current-input-port` and `current-output-port` are also exported by the `scheme` structure.

`input-port-option` *arguments* ⟶ *input-port* [procedure]
`output-port-option` *arguments* ⟶ *output-port* [procedure]
These are utilities for retrieving optional input and output port arguments from rest argument lists, defaulting to the current input or output ports. For example, assuming the newline character sequence is simply `#\newline`, `newline` might be written as:

```
(define (newline . maybe-port)
  (write-char #\newline (output-port-option maybe-port)))
```

`silently` *thunk* ⟶ *values* [procedure]
This stifles output from the current noise port in the dynamic extent of *thunk*, which is applied to zero arguments. `Silently` returns the values that *thunk* returns.

`with-current-ports` *input output error thunk* ⟶ *values* [procedure]
`With-current-ports` dynamically binds the current input, output, and error ports to *input*, *output*, and *error*, respectively, in the dynamic extent of *thunk*, which is applied to zero arguments. The current noise port is also bound to *error*. `With-current-ports` returns the values that *thunk* returns.

Similarly to `with-current-ports`, the `i/o-internal` structure also exports these procedures:

`call-with-current-input-port` *port thunk* ⟶ *values* [procedure]
`call-with-current-output-port` *port thunk* ⟶ *values* [procedure]
`call-with-current-noise-port` *port thunk* ⟶ *values* [procedure]
These bind individual current ports for the dynamic extent of each *thunk*, which is applied to zero arguments. These all return the values that *thunk* returns.

4.5.2 Programmatic ports

Ports are user-extensible; all primitive port operations on them — `read-char`, `write-block`, *&c.* — are completely generalized. Abstractions for buffered ports are also available.

4.5.2.1 Port data type

The `ports` structure defines the basis of the port data type and exports the following procedures.

`make-port` *handler status lock data buffer index limit pending-eof?* [procedure]
⟶ *port*

Port constructor. The arguments are all the fields of ports, which are described below. Note that `make-port` is rarely called directly; usually one will use one of the buffered port constructors instead.

`port-handler` *port* ⟶ *port-handler* [procedure]
`port-buffer` *port* ⟶ *buffer or* `#f` [procedure]
`port-lock` *port* ⟶ *value* [procedure]
`port-status` *port* ⟶ *integer-status* [procedure]
`port-data` *port* ⟶ *value* [procedure]
`port-index` *port* ⟶ *integer or* `#f` [procedure]
`port-limit` *port* ⟶ *integer or* `#f` [procedure]
`port-pending-eof?` *port* ⟶ *boolean* [procedure]

Accessors for the port fields:

`handler` The handler is the functional parameterization mechanism: it provides all the port's operations, such as reading/writing blocks, disclosing (see Section 4.6.2 [Writer], page 70) the port, closing the port, *&c*. See Section 4.5.2.2 [Port handlers], page 61.

`buffer` The buffer is used for buffered ports, where it is a byte vector (see Section 4.3 [Bitwise manipulation], page 55). It may be any value for unbuffered ports.

`lock` This misnamed field was originally used for a mutual exclusion lock, before optimistic concurrency was made the native synchronization mechanism in Scheme48. It is now used as a 'timestamp' for buffered ports: it is provisionally written to with a unique value when a thread resets the `index` to reüse the buffer, and it is provisionally read from when reading from the buffer. In this way, if the buffer is reset while another thread is reading from it, the other thread's proposal is invalidated by the different value in memory than what was there when it logged the old timestamp in its proposal.

`status` A mask from the `port-status-options` enumeration; see Section 4.5.3 [Miscellaneous I/O internals], page 65.

`data` Arbitrary data for particular kinds of ports. For example, for a port that tracks line & column information (see Section 6.5 [I/O extensions], page 106), this might be a record containing the underlying port, the line number, and the column number.

`index` The current index into a buffered port's buffer. If the port is not buffered, this is `#f`.

`limit` The limit of the `index` field for a buffered port's buffer. When the `index` field is equal to the `limit` field, the buffer is full. If the port is not buffered, this is `#f`.

`pending-eof?`
For output ports, this is a boolean flag indicating whether the buffer has been forced to output recently. For input ports, this is a boolean flag indicating whether an end of file is pending after reading through the current buffer.

`set-port-lock!` *port value* ⟶ *unspecified* [procedure]
`set-port-status!` *port status* ⟶ *unspecified* [procedure]
`set-port-data!` *port data* ⟶ *unspecified* [procedure]
`set-port-index!` *port index* ⟶ *unspecified* [procedure]
`set-port-limit!` *port index* ⟶ *unspecified* [procedure]
`set-port-pending-eof?!` *port pending-eof?* ⟶ *unspecified* [procedure]
These assign respective fields of ports. The `buffer` and `handler` fields, however, are immutable.

`provisional-port-handler` *port* ⟶ *port-handler* [procedure]
`provisional-port-lock` *port* ⟶ *value* [procedure]
`provisional-port-status` *port* ⟶ *integer-status* [procedure]
`provisional-port-data` *port* ⟶ *value* [procedure]
`provisional-port-index` *port* ⟶ *integer or* `#f` [procedure]
`provisional-port-limit` *port* ⟶ *integer or* `#f` [procedure]
`provisional-port-pending-eof?` *port* ⟶ *boolean* [procedure]
`provisional-set-port-lock!` *port value* ⟶ *unspecified* [procedure]
`provisional-set-port-status!` *port status* ⟶ *unspecified* [procedure]
`provisional-set-port-data!` *port data* ⟶ *unspecified* [procedure]
`provisional-set-port-index!` *port index* ⟶ *unspecified* [procedure]
`provisional-set-port-limit!` *port index* ⟶ *unspecified* [procedure]
`provisional-set-port-pending-eof?!` *port pending-eof?* ⟶ *unspecified* [procedure]
Provisional versions of the above port accessors & modifiers; that is, accessors & modifiers that log in the current proposal, if there is one.

4.5.2.2 Port handlers

Port handlers store a port's specific operations for the general port operations, such as block reads and writes, buffer flushing, *&c*. Port handler constructors, including `make-port-handler` & the buffered port handlers in the next section, are available from the `i/o-internal` structure.

`make-port-handler` *discloser closer char-reader/writer block-reader/writer readiness-tester buffer-forcer* ⟶ *port-handler* [procedure]
Basic port handler constructor. The arguments are used for the port handler fields. Each field contains a procedure. The expected semantics of each procedure depend on whether the port is for input or output. Input ports do not use the `buffer-forcer` field. The first two fields are independent of the type of port:

discloser *port* ⟶ *disclosed*
: Returns a disclosed representation of the port, *i.e.* a list whose `car` is the 'type name' of this handler (usually with a suffix of either `-input-port` or `-output-port`) followed by a list of all of the components to be printed; see Section 4.6.2 [Writer], page 70.

closer *port* ⟶ *ignored*
: Closes *port*. This operation corresponds with the `close-input-port` & `close-output-port` procedures.

For input ports, the remaining fields are:

char-reader *port consume?* ⟶ *char*
: Reads a single character from *port*. If *consume?* is true, the character should be consumed from *port*; if *consume?* is `#f`, however, the character should be left in *port*'s input stream. If *consume?* is true, this operation corresponds with `read-char`; if it is `#f`, this operation corresponds with `peek-char`.

block-reader *port block start count wait?* ⟶ *count-written or EOF*
: Attempts to read *count* characters from *port*'s input stream into the string or byte vector *block*, starting at *start*. In the case that an insufficient number of characters is available, if *wait?* is true, the procedure should wait until all of the wanted characters are available; otherwise, if *wait?* is `#f`, the block reader should immediately return. In either case, it returns the number of characters that were read into *block*, or an end of file object if it immediately reached the end of the stream. Buffered ports will typically just copy elements from the buffer into *block*, rather than reading from any internal I/O channel in *port*. This operation corresponds with `read-block`.

readiness-tester *port* ⟶ *boolean*
: Returns a true value if there is a character available to be read in *port* or `#f` if not. This operation corresponds with the `char-ready?` procedure.

For output ports, the remaining fields are:

char-writer *port char* ⟶ *ignored*
: Writes the single character *char* to *port*. This operation corresponds with `write-char`.

block-writer *port block start count* ⟶ *count-written*
: Writes *count* characters to *port* from *block*, starting at *start*. *Block* may be a string or a byte vector. This will usually involve copying contents of *block* to *port*'s buffer, if it is buffered. This operation corresponds with `write-block`.

readiness-tester *port* ⟶ *boolean*
: Returns a true value if *port* is ready to receive a character and `#f` if not.

buffer-forcer *port necessary?* ⟶ *ignored*
: For buffered ports, this is intended to force all buffered output to the actual internal I/O channel of *port*. *Necessary?* tells whether or not it is

absolutely necessary to force all the output immediately; if it is `#t`, the buffer forcer is required to force all output in the buffer before it returns. If *necessary?* is `#f`, not only may it just register an I/O transaction without waiting for it to complete, but it also should *not* signal an error if *port* is already closed. For unbuffered ports, this operation need not do anything at all.

4.5.2.3 Buffered ports & handlers

Along with bare port handlers, Scheme48 provides conveniences for many patterns of buffered ports & port handlers. These names are exported by the `i/o-internal` structure. Buffered ports are integrated with Scheme48's optimistic concurrency (see Section 5.2 [Optimistic concurrency], page 77) facilities.

Note: Although internally buffered ports are integrated with optimistic concurrency, operations on buffered ports, like operations on channels, cannot be reliably fusibly atomic.

`make-buffered-input-port` *handler data buffer index limit* $\longrightarrow$ *input-port* [procedure]
`make-buffered-output-port` *handler data buffer index limit* $\longrightarrow$ *output-port* [procedure]

Constructors for buffered ports. *Handler* is the port's handler, which is usually constructed with one of the buffered port handler constructors (see below). *Data* is arbitrary data to go in the port's `data` field. *Buffer* is a byte vector whose length is greater than or equal to both *index* & *limit*. *Index* is the initial index into *buffer* to go in the port's `index` field. *Limit* is the limit in the port's buffer, to go into the port's `limit` field; nothing will be written into *buffer* at or past *limit*.

`make-unbuffered-input-port` *handler data* $\longrightarrow$ *input-port* [procedure]
`make-unbuffered-output-port` *handler data* $\longrightarrow$ *output-port* [procedure]

Conveniences for ports that are explicitly *not* buffered. Only the relevant fields are passed; all fields pertaining to buffering are initialized with `#f`.

`make-buffered-input-port-handler` *discloser closer buffer-filler readiness-tester* $\longrightarrow$ *port-handler* [procedure]

This creates a port handler for buffered input ports. The arguments are as follows:

discloser *port-data* $\longrightarrow$ *disclosed*
closer *port-data* $\longrightarrow$ *ignored*
: *Discloser* & *closer* are like the similarly named regular port handler fields, but they are applied directly to the port's data, not to the port itself.

buffer-filler *port wait?* $\longrightarrow$ *committed?*
: Used to fill *port*'s buffer when it no longer has contents from which to read in its current buffer. *Wait?* is a boolean flag, `#t` if the operation should wait until the I/O transaction necessary to fill the buffer completes, or `#f` if it may simply initiate an I/O transaction but not wait until it completes (*e.g.*, use `channel-maybe-commit-and-read`, but not wait on the condition variable passed to `channel-maybe-commit-and-read`). *Buffer-filler* is called with a fresh proposal in place, and it is the responsibility of *buffer-filler* to commit it. It returns a boolean flag denoting whether

the proposal was committed. The last call in *buffer-filler* is usually either `(maybe-commit)` or a call to a procedure that causes that effect (*e.g.*, one of the operation on condition variables that commits the current proposal. See Section 5.3 [Higher-level synchronization], page 83.)

readiness-tester *port* ⟶ *[committed? ready?]*
Called when `char-ready?` is applied to *port* and the buffer of *port* is empty. Like *buffer-filler*, *readiness-tester* is applied with a fresh proposal in place, which it should attempt to commit. *Readiness-tester* should return two values, each a boolean flag: the first denotes whether or not the current proposal was successfully committed, and, if it was successful, whether or not a character is ready.

`make-buffered-output-port-handler` *discloser buffer-emptier readiness-tester* ⟶ *port-handler* [procedure]
This creates a port handler for buffered output ports. *Discloser* & *closer* are as with buffered input ports. The remaining fields are as follows:

buffer-emptier *port necessary?* ⟶ *committed?*
Buffer-emptier is used when *port*'s buffer is full and needs to be emptied. It is called with a fresh proposal in place. It should reset *port*'s `index` field, call `note-buffer-reuse!` to invalidate other threads' transactions on the recycled buffer, and attempt to commit the new proposal installed. It returns a boolean flag indicating whether or not the commit succeeded.

readiness-tester *port* ⟶ *[committed? ready?]*
Readiness-tester is applied to *port* when its buffer is full (*i.e.* its `index` & `limit` fields are equal) and `output-port-ready?` is applied to *port*. After performing the test, it should attempt to commit the current proposal and then return two values: whether it succeeded in committing the current proposal, and, if it was successful, whether or not a character is ready to be outputted.

`default-buffer-size` ⟶ *integer* [constant]
The default size for port buffers. This happens to be 4096 in the current version of Scheme48.

`note-buffer-reuse!` *port* ⟶ *unspecified* [procedure]
`check-buffer-timestamp!` *port* ⟶ *unspecified* [procedure]
These are used to signal the resetting of a buffer between multiple threads. `Note-buffer-reuse!` is called — in the case of an output port — when a buffer fills up, is emptied, and flushed; or — in the case of an input port — when a buffer is emptied and needs to be refilled. `Note-buffer-reuse!` logs in the current proposal a fresh value to store in *port*. When that proposal is committed, this fresh value is stored in the port. Other threads that were using *port*'s buffer call `check-buffer-timestamp!`, which logs a read in the current proposal. If another thread commits a buffer reüse to memory, that read will be invalidated, invalidating the whole transaction.

4.5.3 Miscellaneous I/O internals

All of these but `port-status-options` are exported by the `i/o-internal` structure; the `port-status-options` enumeration is exported by the `architecture` structure, but it deserves mention in this section.

`port-status-options` [enumeration]

```
(define-enumeration port-status-options
  (input
   output
   open-for-input
   open-for-output))
```

Enumeration of indices into a port's `status` field bit set.

`open-input-port?` *port* ⟶ *boolean* [procedure]
`open-output-port?` *port* ⟶ *boolean* [procedure]
These return true values if *port* is both an input or output port, respectively, and open.

`open-input-port-status` ⟶ *integer-status* [constant]
`open-output-port-status` ⟶ *integer-status* [constant]
The bitwise masks of enumerands from the `port-status-options` enumeration signifying an open input or output port, respectively.

`make-input-port-closed!` *port* ⟶ *unspecified* [procedure]
`make-output-port-closed!` *port* ⟶ *unspecified* [procedure]
These set the status of *port*, which must be an input or output port, respectively, to indicate that it is closed.

`eof-object` ⟶ *eof-object* [procedure]
Returns the EOF object token. This is the only value that will answer true to R5RS's `eof-object?` predicate.

`force-output-if-open` *port* ⟶ *unspecified* [procedure]
This forces *port*'s output if it is an open output port, and does not block.

`periodically-force-output!` *port* ⟶ *unspecified* [procedure]
`periodically-flushed-ports` ⟶ *port-list* [procedure]
`Periodically-force-output!` registers *port* to be forced periodically. Only a weak reference to *port* in this registry is held, however, so this cannot cause accidental space leaks. `Periodically-flushed-ports` returns a list of all ports in this registry. Note that the returned list holds strong references to all of its elements. `Periodically-flushed-ports` does not permit thread context switches, or interrupts of any sort, while it runs.

4.5.4 Channels

Channels represent the OS's native I/O transmission channels. On Unix, channels are essentially boxed file descriptors, for example. The only operations on channels are block reads & writes. Blocks in this sense may be either strings or byte vectors (see Section 4.3 [Bitwise manipulation], page 55).

4.5.4.1 Low-level channel operations

The low-level base of the interface to channels described here is exported from the `channels` structure.

`channel?` $\longrightarrow$ *boolean* [procedure]
Disjoint type predicate for channels.

`channel-id` *channel* $\longrightarrow$ *value* [procedure]
`channel-status` *channel* $\longrightarrow$ *integer-enumerand* [procedure]
`channel-os-index` *channel* $\longrightarrow$ *integer* [procedure]
`Channel-id` returns *channel*'s id. The id is some identifying characteristic of channels. For example, file channels' ids are usually the corresponding filenames; channels such as the standard input, output, or error output channels have names like `"standard input"` and `"standard output"`. `Channel-status` returns the current status of *channel*; see the `channel-status-option` enumeration below. `Channel-os-index` returns the OS-specific integer index of *channel*. On Unix, for example, this is the channel's file descriptor.

`open-channel` *filename option close-silently?* $\longrightarrow$ *channel* [procedure]
`Open-channel` opens a channel for a file given its filename. *Option* specifies what type of channel this is; see the `channel-status-option` enumeration below. *Close-silently?* is a boolean that specifies whether a message should be printed (on Unix, to `stderr`) when the resulting channel is closed after a garbage collector finds it unreachable.

`close-channel` *channel* $\longrightarrow$ *unspecified* [procedure]
Closes *channel* after aborting any potential pending I/O transactions it may have been involved with.

`channel-ready?` *channel* $\longrightarrow$ *boolean* [procedure]
If *channel* is an input channel: returns `#t` if there is input ready to be read from *channel* or `#f` if not; if *channel* is an output channel: returns `#t` if a write would immediately take place upon calling `channel-maybe-write`, *i.e.* `channel-maybe-write` would not return `#f`, or `#f` if not.

`channel-maybe-read` *channel buffer start-index octet-count wait?* [procedure]
$\longrightarrow$ *octet count read, error status cell, EOF object, or* `#f`
`channel-maybe-write` *channel buffer start-index octet-count* $\longrightarrow$ [procedure]
octet count written, error status cell, or `#f`
`channel-abort` *channel* $\longrightarrow$ *unspecified* [procedure]
`Channel-maybe-read` attempts to read *octet-count* octets from *channel* into *buffer*, starting at *start-index*. If a low-level I/O error occurs, it returns a cell containing a token given by the operating system indicating what kind of error occurred. If *wait?* is `#t`, and *channel* is not ready to be read from, *channel* is registered for the VM's event polling mechanism, and `channel-maybe-read` returns `#f`. Otherwise, it returns either the number of octets read, or an EOF object if *channel* was was at the end.

`Channel-maybe-write` attempts to write *octet-count* octets to *channel* from *buffer*, starting at *start-index*. If a low-level I/O error occurs, it returns a cell indicating a

token given by the operating system indicating what kind of error occurred. If no such low-level error occurs, it registers *channel* for the VM's event polling mechanism and returns `#f` iff zero octets were immediately written or the number of octets immediately written if any were.

`Channel-abort` aborts any pending operation registered for the VM's event polling mechanism.

`open-channels-list` ⟶ *channel-list* [procedure]
Returns a list of all open channels in order of the `os-index` field.

`channel-status-option` [enumeration]

```
(define-enumeration channel-status-option
  (closed
   input
   output
   special-input
   special-output))
```

Enumeration for a channel's status. The `closed` enumerand is used only after a channel has been closed. Note that this is *not* suitable for a bit mask; that is, one may choose exactly one of the enumerands, not use a bit mask of status options. For example, to open a file `frob` for input that one wishes the garbage collector to be silent about on closing it:

```
(open-channel "frob"
              (enum channel-status-option input)
              #t)
   ⇒ #{Input-channel "frob"}
```

4.5.4.2 Higher-level channel operations

More convenient abstractions for operating on channels, based on condition variables (see Section 5.3 [Higher-level synchronization], page 83), are provided from the `channel-i/o` structure. They are integrated with Scheme48's optimistic concurrency (see Section 5.2 [Optimistic concurrency], page 77) facilities.

Note: Transactions on channels can *not* be atomic in the sense of optimistic concurrency. Since they involve communication with the outside world, they are irrevocable transactions, and thus an invalidated proposal cannot retract the transaction on the channel.

`channel-maybe-commit-and-read` *channel buffer start-index octet-count condvar wait?* ⟶ *committed?* [procedure]
`channel-maybe-commit-and-write` *channel buffer start-index octet-count condvar* ⟶ *committed?* [procedure]
These attempt to commit the current proposal. If they fail, they immediately return `#f`; otherwise, they proceed, and return `#t`. If the commit succeeded, these procedures attempt an I/O transaction, without blocking. `Channel-maybe-commit-and-read` attempts to read *octet-count* octets into *buffer*, starting at *start-index*, from *channel*. `Channel-maybe-commit-and-write` attempts to write *octet-count* octets from *buffer*, starting at *start-index*, to *channel*. *Condvar* is noted as waiting for the completion of the I/O transaction. When the I/O transaction finally completes — in the case of a

read, there are octets ready to be read into *buffer* from *channel* or the end of the file was struck; in the case of a write, *channel* is ready to receive some octets —, *condvar* is set to the result of the I/O transaction: the number of octets read, an I/O error condition, or an EOF object, for reads; and the number of octets written or an I/O error condition, for writes.

`channel-maybe-commit-and-close` *channel closer* ⟶ *committed?* [procedure]
Attempts to commit the current proposal; if successful, this aborts any wait on *channel*, sets the result of any condvars waiting on *channel* to the EOF object, closes *channel* by applying *closer* to *channel* (in theory, *closer* could be anything; usually, however, it is `close-channel` from the `channels` structure or some wrapper around it), and returns `#t`. If the commit failed, `channel-maybe-commit-and-close` immediately returns `#f`.

`channel-write` *channel buffer start-index octet-count* ⟶ *octet-count-written* [procedure]
Atomically attempts to write *octet-count* octets to *channel* from *buffer*, starting at *start-index* in *buffer*. If no I/O transaction immediately occurs — what would result in `channel-maybe-write` returning `#f` —, `channel-write` blocks until something does happen. It returns the number of octets written to *channel*.

`wait-for-channel` *channel condvar* ⟶ *unspecified* [procedure]
Registers *condvar* so that it will be set to the result of some prior I/O transaction when some I/O event regarding *channel* occurs. (Contrary to the name, this does not actually wait or block. One must still use `maybe-commit-and-wait-for-condvar` on *condvar*; see Section 5.3 [Higher-level synchronization], page 83.) This is useful primarily in conjunction with calling foreign I/O routines that register channels with the VM's event polling system.

Note: `wait-for-channel` must be called with interrupts disabled.

4.5.5 Channel ports

Built-in to Scheme48 are ports made atop channels. These are what are created by R5RS's standard file operations. The following names are exported by the `channel-ports` structure.

`call-with-input-file` *filename receiver* ⟶ *values* [procedure]
`call-with-output-file` *filename receiver* ⟶ *values* [procedure]
`with-input-from-file` *filename thunk* ⟶ *values* [procedure]
`with-output-to-file` *filename thunk* ⟶ *values* [procedure]
`open-input-file` *filename* ⟶ *input-port* [procedure]
`open-output-file` *filename* ⟶ *output-port* [procedure]
Standard R5RS file I/O operations. (These are also exported by the `scheme` structure.) The `call-with-...put-file` operations open the specified type of port and apply *receiver* to it; after *receiver* returns normally (*i.e.* nothing is done if there is a throw out of *receiver*), they close the port and return the values that *receiver* returned. `With-input-from-file` & `with-output-to-file` do similarly, but, rather than applying *thunk* to the port, they dynamically bind the current input & output ports, respectively, to the newly opened ports. `Call-with-input-file`, `call-with-output-file`, `with-input-from-file`, and `with-output-to-file` return the values

that *thunk* returns. `Open-input-file` & `open-output-file` just open input & output ports; users of these operations must close them manually.

`input-channel->port` *channel* [*buffer-size*] ⟶ *port* [procedure]
`output-channel->port` *channel* [*buffer-size*] ⟶ *port* [procedure]
These create input & output ports atop the given channels and optional buffer sizes. The default buffer size is 4096 bytes.

`input-channel+closer->port` *channel closer* [*buffer-size*] ⟶ *port* [procedure]
`output-channel+closer->port` *channel closer* [*buffer-size*] ⟶ *port* [procedure]
Similarly, these create input & output ports atop the given channels and optional buffer sizes, but they allow for extra cleanup when the resulting ports are closed.

`port->channel` *port* ⟶ *channel or* `#f` [procedure]
If *port* is a port created by the system's channel ports facility, `port->channel` returns the channel it was created atop; otherwise `port->channel` returns `#f`.

`force-channel-output-ports!` ⟶ *unspecified* [procedure]
This attempts to force as much output as possible from all of the ports based on channels. This is used by Scheme48's POSIX libraries before forking the current process.

4.6 Reader & writer

Scheme48 has simple S-expression reader & writer libraries, with some facilities beyond R5RS's `read` & `write` procedures.

4.6.1 Reader

Scheme48's reader facility is exported by the `reading` structure. The `read` binding thereby exported is identical to that of the `scheme` structure, which is the binding that R5RS specifies under the name `read`.

`read` [*port*] ⟶ *readable-value* [procedure]
Reads a single S-expression from *port*, whose default value is the current input port. If the end of the stream is encountered before the beginning of an S-expression, `read` will return an EOF object. It will signal a read error if text read from *port* does not constitute a complete, well-formed S-expression.

`define-sharp-macro` *char proc* ⟶ *unspecified* [procedure]
Defines a sharp/pound/hash/octothorpe (`#`) reader macro. The next time the reader is invoked, if it encounters an octothorpe/sharp followed by *char*, it applies *proc* to *char* and the input port being read from. *Char* is *not* consumed in the input port. If *char* is alphabetic, it should be lowercase; otherwise the reader will not recognize it, since the reader converts the character following octothorpes to lowercase.

`reading-error` *port message irritant* ... ⟶ *unspecified* [procedure]
Signals an error while reading, for custom sharp macros. It is not likely that calls to `reading-error` will return.

`gobble-line` *port* ⟶ *unspecified* [procedure]
Reads until a newline from *port*. The newline character sequence is consumed.

4.6.2 Writer

Scheme48's `writing` structure exports its writer facility. The `write` and `display` bindings from it are identical to those from the `scheme` structure, which are the same bindings that R5RS specifies.

`write` *object* [*port*] ⟶ *unspecified* [procedure]
: Writes *object* to *port*, which defaults to the current output port, in a machine-readable manner. Strings are written with double- quotes; characters are prefixed by `#\`. Any object that is unreadable — anything that does not have a written representation as an S-expression — is written based on its *disclosed* representation. Such unreadable objects are converted to a disclosed representation by the `disclose` generic procedure (see below).

`display` *object* [*port*] ⟶ *unspecified* [procedure]
: Displays *object* to *port*, which defaults to the value of the current output port, in a more human-readable manner. Strings are written without surrounding double-quotes; characters are written as themselves with no prefix.

`recurring-write` *object port recur* ⟶ *unspecified* [procedure]
: Writes *object* to *port*. Every time this recurs upon a new object, rather than calling itself or its own looping procedure, it calls *recur*. This allows customized printing routines that still take advantage of the existence of Scheme48's writer. For example, `display` simply calls `recurring-write` with a recurring procedure that prints strings and characters specially and lets `recurring-write` handle everything else.

`display-type-name` *name port* ⟶ *unspecified* [procedure]
: If *name* is a symbol with an alphabetic initial character, this writes *name* to *port* with the first character uppercased and the remaining character lowercased; otherwise, `display-type-name` simply writes *name* to *port* with `display`.

```
(display-type-name 'foo)
    ⊣ Foo

(display-type-name (string->symbol "42foo"))
    ⊣ 42foo

(display-type-name (cons "foo" "bar"))
    ⊣ (foo . bar)

(display-type-name (string->symbol "fOo-BaR"))
    ⊣ Foo-bar
```

This is used when printing disclosed representations (see below).

4.6.2.1 Object disclosure

The `methods` structure (see Section 4.4 [Generic dispatch system], page 56) exports the generic procedure `disclose` and its method table `&disclose`. When `recurring-write` encounters an object it is unable to write in a rereadable manner, it applies `disclose` to the unreadable object to acquire a *disclosed representation*. (If `disclose` returns `#f`, *i.e.*

the object has no disclosed representation, the writer will write `#{Random object}`.) After converting a value to its disclosed representation, *e.g.* a list consisting of the symbol `foo`, the symbol `bar`, a byte vector, and a pair `(1 . 2)`, the writer will write `#{Foo #{Byte-vector} bar (1 . 2)}`. That is: contents of the list are surrounded by `#{` and `}`, the first element of the list (the 'type name') is written with `display-type-name`, and then the remaining elements of the list are recursively printed out with the *recur* argument.

Typically, when a programmer creates an abstract data type by using Scheme48's record facility, he will not add methods to `&disclose` but instead define the record type's discloser with the `define-record-discloser` procedure; see Section 4.7 [Records], page 71.

Example:

```
(define-record-type pare rtd/pare
  (kons a d)
  pare?
  (a kar set-kar!)
  (d kdr set-kdr!))

(define-record-discloser rtd/pare
  (lambda (pare)
    `(pare ,(kar pare) *dot* ,(kdr pare))))

(write (kons (kons 5 3) (kons 'a 'b)))
    ⊣ #{Pare #{Pare 5 *dot* 3} *dot* #{Pare a *dot* b}}
```

4.7 Records

Scheme48 provides several different levels of a record facility. Most programmers will probably not care about the two lower levels; the syntactic record type definers are sufficient for abstract data types.

At the highest level, there are two different record type definition macros. Richard Kelsey's is exported from the `defrecord` structure; Jonathan Rees's is exported from `define-record-types`. They both export a `define-record-type` macro and the same `define-record-discloser` procedure; however, the macros are dramatically different. Scheme48 also provides [SRFI 9], which is essentially Jonathan Rees's record type definition macro with a slight syntactic difference, in the `srfi-9` structure. Note, however, that `srfi-9` does not export `define-record-discloser`. The difference between Jonathan Rees's and Richard Kelsey's record type definition macros is merely syntactic convenience; Jonathan Rees's more conveniently allows for arbitrary naming of the generated variables, whereas Richard Kelsey's is more convenient if the naming scheme varies little.

4.7.1 Jonathan Rees's `define-record-type` macro

`define-record-type` [syntax]

```
(define-record-type record-type-name record-type-variable
  (constructor constructor-argument ...)
  [predicate]
  (field-tag field-accessor [field-modifier])
  ...)
```

This defines *record-type-variable* to be a record type descriptor. *Constructor* is defined to be a procedure that accepts the listed field arguments and creates a record of the newly defined type with those fields initialized to the corresponding arguments. *Predicate*, if present, is defined to be the disjoint (as long as abstraction is not violated by the lower-level record interface) type predicate for the new record type. Each *field-accessor* is defined to be a unary procedure that accepts a record type and returns the value of the field named by the corresponding *field-tag*. Each *field-modifier*, if present, is defined to be a binary procedure that accepts a record of the new type and a value, which it assigns the field named by the corresponding *field-tag* to. Every *constructor-argument* must have a corresponding *field-tag*, though *field-tag*s that are not used as arguments to the record type's constructor are simply uninitialized when created. They should have modifiers: otherwise they will never be initialized.

It is worth noting that Jonathan Rees's `define-record-type` macro does not introduce identifiers that were not in the original macro's input form.

For example:

```
(define-record-type pare rtd/pare
  (kons a d)
  pare?
  (a kar)
  (d kdr set-kdr!))

(kar (kons 5 3))
    ⇒ 5

(let ((p (kons 'a 'c)))
  (set-kdr! p 'b)
  (kdr p))
    ⇒ b

(pare? (kons 1 2))
    ⇒ #t

(pare? (cons 1 2))
    ⇒ #f
```

There is also a variant of Jonathan Rees's `define-record-type` macro for defining record types with fields whose accessors and modifiers respect optimistic concurrency (see Section 5.2 [Optimistic concurrency], page 77) by logging in the current proposal.

4.7.2 Richard Kelsey's `define-record-type` macro

`define-record-type` [syntax]

```
(define-record-type type-name
  (argument-field-specifier ...)
  (nonargument-field-specifier ...))

    argument-field-specifier  ⟶
```

```
    field-tag                Immutable field
  | (field-tag)              Mutable field
nonargument-field-specifier  ⟶
    field-tag                Uninitialized field
  | (field-tag exp)          Initialized with exp's value
```

This defines `type/`*`type-name`* to be a record type descriptor for the newly defined record type, *`type-name`*`-maker` to be a constructor for the new record type that accepts arguments for every field in the argument field specifier list, *`type-name`*`?` to be the disjoint type predicate for the new record type, accessors for each field tag *field-tag* by constructing an identifier *`type-name-field-tag`*, and modifiers for each argument field tag that was specified to be mutable as well as each nonargument field tag. The name of the modifier for a field tag *field-tag* is constructed to be `set-`*`type-name-field-tag`*`!`.

Note that Richard Kelsey's `define-record-type` macro *does* concatenate & introduce new identifiers, unlike Jonathan Rees's.

For example, a use of Richard Kelsey's `define-record-type` macro

```
(define-record-type pare
  (kar
   (kdr))
  (frob
   (mumble 5)))
```

is equivalent to the following use of Jonathan Rees's macro

```
(define-record-type pare type/pare
  (%pare-maker kar kdr mumble)
  pare?
  (kar pare-kar)
  (kdr pare-kdr set-pare-kdr!)
  (frob pare-frob set-pare-frob!)
  (mumble pare-mumble set-pare-mumble!))

(define (pare-maker kar kdr)
  (%pare-maker kar kdr 5))
```

4.7.3 Record types

Along with two general record type definition facilities, there are operations directly on the record type descriptors themselves, exported by the `record-types` structure. (Record type descriptors are actually records themselves.)

`make-record-type` *name field-tags* ⟶ *record-type-descriptor* [procedure]
`record-type?` *object* ⟶ *boolean* [procedure]

`Make-record-type` makes a record type descriptor with the given name and field tags. `Record-type?` is the disjoint type predicate for record types.

`record-type-name` *rtype-descriptor* ⟶ *symbol* [procedure]
`record-type-field-names` *rtype-descriptor* ⟶ *symbol-list* [procedure]

Accessors for the two record type descriptor fields.

`record-constructor` *rtype-descriptor argument-field-tags* ⟶ *constructor-procedure* [procedure]
`record-predicate` *rtype-descriptor* ⟶ *predicate-procedure* [procedure]
`record-accessor` *rtype-descriptor field-tag* ⟶ *accessor-procedure* [procedure]
`record-modifier` *rtype-descriptor field-tag* ⟶ *modifier-procedure* [procedure]
Constructors for the various procedures relating to record types. `Record-constructor` returns a procedure that accepts arguments for each field in *argument-field-tags* and constructs a record whose record type descriptor is *rtype-descriptor*, initialized with its arguments. `Record-predicate` returns a disjoint type predicate for records whose record type descriptor is *rtype-descriptor*. `Record-accessor` and `record-modifier` return accessors and modifiers for records whose record type descriptor is *rtype-descriptor* for the given fields.

`define-record-discloser` *rtype-descriptor discloser* ⟶ *unspecific* [procedure]
Defines the method by which records of type *rtype-descriptor* are disclosed (see Section 4.6.2 [Writer], page 70). This is also exported by `define-record-types` and `defrecord`.

`define-record-resumer` *rtype-descriptor resumer* ⟶ *unspecified* [procedure]
Sets *rtype-descriptor*'s record resumer to be *resumer*. If *resumer* is `#t` (the default), records of this type require no particular reinitialization when found in dumped heap images (see Section 4.8 [Suspending and resuming heap images], page 75); if *resumer* is `#f`, records of the type *rtype-descriptor* may not be dumped in heap images; finally, if it is a procedure, and the heap image is resumed with the usual image resumer (see Section 4.8 [Suspending and resuming heap images], page 75), it is applied to each record whose record type descriptor is *rtype-descriptor* after the run-time system has been initialized and before the argument to `usual-resumer` is called.

The `records-internal` structure also exports these:

`:record-type` [record type]
The record type of record types.

`disclose-record` *record* ⟶ *disclosed* [procedure]
This applies *record*'s record type descriptor's discloser procedure to *record* to acquire a disclosed representation; see Section 4.6.2 [Writer], page 70.

For expository purposes, the record type record type might have been defined like so with Jonathan Rees's `define-record-type` macro:

```
(define-record-type record-type :record-type
  (make-record-type name field-names)
  record-type?
  (name record-type-name)
  (field-names record-type-field-names))
```

or like so with Richard Kelsey's `define-record-type` macro:

```
(define-record-type record-type
  (name field-names)
  ())
```

Of course, in reality, these definitions would have severe problems with circularity of definition.

4.7.4 Low-level record manipulation

Internally, records are represented very similarly to vectors, and as such have low-level operations on them similar to vectors, exported by the `records` structure. Records usually reserve the slot at index 0 for their record type descriptor.

Warning: The procedures described here can be very easily misused to horribly break abstractions. Use them very carefully, only in very limited & extreme circumstances!

`make-record` *length init* $\longrightarrow$ *record* [procedure]
`record` *elt ...* $\longrightarrow$ *record* [procedure]
`record?` *object* $\longrightarrow$ *boolean* [procedure]
`record-length` *record* $\longrightarrow$ *integer* [procedure]
`record-ref` *record index* $\longrightarrow$ *value* [procedure]
`record-set!` *record index object* $\longrightarrow$ *unspecified* [procedure]
Exact analogues of similarly named vector operation procedures.

`record-type` *record* $\longrightarrow$ *value* [procedure]
This returns the record type descriptor of *record*, *i.e.* the value of the slot at index 0 in *record*.

4.8 Suspending and resuming heap images

Scheme48's virtual machine operates by loading a heap image into memory and calling the initialization procedure specified in the image dump. Heap images can be produced in several different ways: programmatically with `write-image`, using the command processor's facilities (see Section 2.4.11 [Image-building commands], page 20), or with the static linker. This section describes only `write-image` and the related system resumption & initialization.

Heap image dumps begin with a sequence of characters terminated by an ASCII form-feed/page character (codepoint 12). This content may be anything; for example, it might be a Unix `#!` line that invokes `scheme48vm` on the file, or it might be a silly message to whomever reads the top of the heap image dump file. (The command processor's `,dump` & `,build` commands (see Section 2.4.11 [Image-building commands], page 20) write a blank line at the top; the static linker puts a message stating that the image was built by the static linker.)

`Write-image` is exported by the `write-images` structure.

`write-image` *filename startup-proc message* $\longrightarrow$ *unspecified* [procedure]
Writes a heap image whose startup procedure is *startup-proc* and that consists of every object accessible in some way from *startup-proc*. *Message* is put at the start of the heap image file before the ASCII form-feed/page character. When the image is resumed, *startup-proc* is passed a vector of program arguments, an input channel for standard input, an output channel for standard output, an output channel for standard error, and a vector of records to be resumed. This is typically simplified by `usual-resumer` (see below). On Unix, *startup-proc* must return an integer exit code; otherwise the program will crash and burn with a very low-level VM error message when *startup-proc* returns.

4.8.1 System initialization

When suspended heap images are resumed by the VM, the startup procedure specified in the heap image is applied to five arguments: a vector of command-line arguments (passed after the `-a` argument to the VM), an input channel for standard input, an output channel for standard output, an output channel for standard error, and a vector of records to be resumed. The startup procedure is responsible for performing any initialization necessary — including initializing the Scheme48 run-time system — as well as simply running the program. Typically, this procedure is not written manually: resumers are ordinarily created using the *usual resumer* abstraction, exported from the structure `usual-resumer`.

`usual-resumer` *startup-proc* $\longrightarrow$ *resumer-proc* [procedure]
: This returns a procedure that is suitable as a heap image resumer procedure. When the heap image is resumed, it initializes the run-time system — it resumes all the records, initializes the thread system, the dynamic state, the interrupt system, I/O system, *&c.* — and applies *startup-proc* to a list (not a vector) of the command-line arguments.

Some records may contain machine-, OS-, or other session-specific data. When suspended in heap images and later resumed, this data may be invalidated, and it may be necessary to reinitialize this data upon resumption of suspended heap images. For this reason Scheme48 provides *record resumers*; see `define-record-resumer` from the `record-types` structure (see Section 4.7 [Records], page 71).

4.8.2 Manual system initialization

If a programmer chooses not to use `usual-resumer` — which is *not* a very common thing to do —, he is responsible for manual initialization of the run-time system, including the I/O system, resumption of records, the thread system and the root thread scheduler, the interrupt system, and the condition system.

Warning: Manual initialization of the run-time system is a *very* delicate operation. Although one can potentially vastly decrease the size of dumped heap images by doing it manually,[10] it is very error-prone and difficult to do without exercising great care, which is why the usual resumer facility exists. Unless you **really** know what you are doing, you should just use the usual resumer.

At the present, documentation of manual system initialization is absent. However, if the reader knows enough about what he is doing that he desires to manually initialize the run-time system, he is probably sufficiently familiar with it already to be able to find the necessary information directly from Scheme48's source code and module descriptions.

[10] For example, the author of this manual, merely out of curiosity, compared the sizes of two images: one that used the usual resumer and printed each of its command-line arguments, and one that performed *no* run-time system initialization — which eliminated the run-time system in the image, because it was untraceable from the resumer — and wrote directly to the standard output channel. The difference was a factor of about twenty. However, also note that the difference is constant; the run-time system happened to account for nineteen twentieths of the larger image.

5 Multithreading

This chapter describes Scheme48's fully preëmptive and sophisticated user-level thread system. Scheme48 supports customized and nested thread schedulers, user-designed synchronization mechanisms, optimistic concurrency, useful thread synchronization libraries, a high-level event algebra based on Reppy's Concurrent ML [Reppy 99], and common pessimistic concurrency/mutual-exclusion-based thread synchronization facilities.

5.1 Basic thread operations

This section describes the `threads` structure.

`spawn` *thunk* [*name*] $\longrightarrow$ *thread* [procedure]
: `Spawn` constructs a new thread and instructs the current thread scheduler to commence running the new thread. *Name*, if present, is used for debugging. The new thread has a fresh dynamic environment (see Section 4.1.4 [Fluid/dynamic bindings], page 41).

There are several miscellaneous facilities for thread operations.

`relinquish-timeslice` $\longrightarrow$ *unspecified* [procedure]
`sleep` *count* $\longrightarrow$ *unspecified* [procedure]
: `Relinquish-timeslice` relinquishes the remaining quantum that the current thread has to run; this allows the current scheduler run the next thread immediately. `Sleep` suspends the current thread for *count* milliseconds.

`terminate-current-thread` $\longrightarrow$ (*does not return*) [procedure]
: Terminates the current thread, running all `dynamic-wind` exit points. `Terminate-current-thread` obviously does not return.

Threads may be represented and manipulated in first-class thread descriptor objects.

`current-thread` $\longrightarrow$ *thread* [procedure]
`thread?` *object* $\longrightarrow$ *boolean* [procedure]
`thread-name` *thread* $\longrightarrow$ *value* [procedure]
`thread-uid` *thread* $\longrightarrow$ *unique-integer-id* [procedure]
: `Current-thread` returns the thread descriptor for the currently running thread. `Thread?` is the thread descriptor disjoint type predicate. `Thread-name` returns the name that was passed to `spawn` when spawning *thread*, or `#f` if no name was passed. `Thread-uid` returns a thread descriptor's unique integer identifier, assigned by the thread system.

5.2 Optimistic concurrency

Scheme48's fundamental thread synchronization mechanism is based on a device often used in high-performance database systems: optimistic concurrency. The basic principle of optimistic concurrency is that, rather than mutually excluding other threads from data involved in one thread's transaction, a thread keeps a log of its transaction, not actually modifying the data involved, only touching the log. When the thread is ready to commit its changes,

it checks that all of the reads from memory retained their integrity — that is, all of the memory that was read from during the transaction has remained the same, and is consistent with what is there at the time of the commit. If, and only if, all of the reads remained valid, the logged writes are committed; otherwise, the transaction has been invalidated. While a thread is transacting, any number of other threads may be also transacting on the same resource. All that matters is that the values each transaction read are consistent with every write that was committed during the transaction. This synchronization mechanism allows for wait-free, lockless systems that easily avoid confusing problems involving careful sequences of readily deadlock-prone mutual exclusion.

In the Scheme48 system, every thread has its own log of transactions, called a *proposal*. There are variants of all data accessors & modifiers that operate on the current thread's proposal, rather than actual memory: after the initial read of a certain part of memory — which *does* perform a real read —, the value from that location in memory is cached in the proposal, and thenceforth reads from that location in memory will actually read the cache; modifications touch only the proposal, until the proposal is committed.

All of the names described in this section are exported by the `proposals` structure.

5.2.1 High-level optimistic concurrency

There are several high-level operations that abstract the manipulation of the current thread's proposal.

`call-ensuring-atomicity` *thunk* ⟶ *values* [procedure]
`call-ensuring-atomicity!` *thunk* ⟶ *unspecified* [procedure]
These ensure that the operation of *thunk* is atomic. If there is already a current proposal in place, these are equivalent to calling *thunk*. If there is not a current proposal in place, these install a new proposal, call *thunk*, and attempt to commit the new proposal. If the commit succeeded, these return. If it failed, these retry with a new proposal until they do succeed. `Call-ensuring-atomicity` returns the values that *thunk* returned when the commit succeeded; `call-ensuring-atomicity!` returns zero values — it is intended for when *thunk* is used for its effects only.

`call-atomically` *thunk* ⟶ *values* [procedure]
`call-atomically!` *thunk* ⟶ *unspecified* [procedure]
These are like *call-ensuring-atomicity* and *call-ensuring-atomicity!*, respectively, except that they always install a new proposal (saving the old one and restoring it when they are done).

`ensure-atomicity` *body* ⟶ *values* [syntax]
`ensure-atomicity!` *body* ⟶ *unspecified* [syntax]
`atomically` *body* ⟶ *values* [syntax]
`atomically!` *body* ⟶ *unspecified* [syntax]
These are syntactic sugar over `call-ensuring-atomicity`, `call-ensuring-atomicity!`, `call-atomically`, and `call-atomically!`, respectively.

Use these high-level optimistic concurrency operations to make the body atomic. `Call-ensuring-atomicity` *&c.* simply ensure that the transaction will be atomic, and may 'fuse' it with an enclosing atomic transaction if there already is one, *i.e.* use the proposal for that transaction already in place, creating one only if there is not already

one. `Call-atomically` *&c.* are for what might be called 'subatomic' transactions, which cannot be fused with other atomic transactions, and for which there is always created a new proposal.

However, code within `call-ensuring-atomicity` *&c.* or `call-atomically` *&c.* should *not* explicitly commit the current proposal; those operations above *automatically* commit the current proposal when the atomic transaction is completed. (In the case of `call-atomically` *&c.*, this is when the procedure passed returns; in the case of `call-ensuring-atomicity` *&c.*, this is when the outermost enclosing atomic transaction completes, or the same as `call-atomically` if there was no enclosing atomic transaction.) To explicitly commit the current proposal — for example, to perform some particular action if the commit fails rather than just to repeatedly retry the transaction, or to use operations from the customized thread synchronization (see Section 5.6 [Custom thread synchronization], page 92) facilities that commit the current proposal after their regular function, or the operations on condition variables (see Section 5.3 [Higher-level synchronization], page 83) that operate on the condition variable and then commit the current proposal —, one must use the `with-new-proposal` syntax as described below, not these operations.

5.2.2 Logging variants of Scheme procedures

`provisional-car` *pair* ⟶ *value* [procedure]
`provisional-cdr` *pair* ⟶ *value* [procedure]
`provisional-set-car!` *pair value* ⟶ *unspecified* [procedure]
`provisional-set-cdr!` *pair value* ⟶ *unspecified* [procedure]
`provisional-cell-ref` *cell* ⟶ *value* [procedure]
`provisional-cell-set!` *cell value* ⟶ *unspecified* [procedure]
`provisional-vector-ref` *vector index* ⟶ *value* [procedure]
`provisional-vector-set!` *vector index value* ⟶ *unspecified* [procedure]
`provisional-string-ref` *string index* ⟶ *char* [procedure]
`provisional-string-set!` *string index value* ⟶ *unspecified* [procedure]
`provisional-byte-vector-ref` *byte-vector index* ⟶ *char* [procedure]
`provisional-byte-vector-set!` *byte-vector index byte* ⟶ *unspecified* [procedure]
`attempt-copy-bytes!` *from fstart to tstart count* ⟶ *unspecified* [procedure]

These are variants of most basic Scheme memory accessors & modifiers that log in the current proposal, rather than performing the actual memory access/modification. All of these do perform the actual memory access/modification, however, if there is no current proposal in place when they are called. `Attempt-copy-bytes!` copies a sequence of *count* bytes from the byte vector or string *from*, starting at the index *fstart*, to the byte vector or string *to*, starting at the index *tstart*.

5.2.3 Synchronized records

`define-synchronized-record-type` [syntax]

```
(define-synchronized-record-type tag type-name
  (constructor-name parameter-field-tag ...)
  [(sync-field-tag ...)]
  predicate-name
```

```
(field-tag accessor-name [modifier-name])
...)
```

This is exactly like `define-record-type` from the `define-record-types` structure, except that the accessors & modifiers for each field in *sync-field-tag* ... are defined to be provisional, *i.e.* to log in the current proposal. If the list of synchronized fields is absent, all of the fields are synchronized, *i.e.* it is as if all were specified in that list.

The `proposals` structure also exports `define-record-discloser` (see Section 4.7 [Records], page 71). Moreover, the `define-sync-record-types` structure, too, exports `define-synchronized-record-type`, though it does not export `define-record-discloser`.

5.2.4 Optimistic concurrency example

Here is a basic example of using optimistic concurrency to ensure the synchronization of memory. We first present a simple mechanism for counting integers by maintaining internal state, which is expressed easily with closures:

```
(define (make-counter value)
  (lambda ()
    (let ((v value))
      (set! value (+ v 1))
      v)))
```

This has a problem: between obtaining the value of the closure's slot for `value` and updating that slot, another thread might be given control and modify the counter, producing unpredictable results in threads in the middle of working with the counter. To remedy this, we might add a mutual exclusion lock to counters to prevent threads from simultaneously accessing the cell:

```
(define (make-counter value)
  (let ((lock (make-lock)))
    (lambda ()
      (dynamic-wind
        (lambda () (obtain-lock lock))
        (lambda ()
          (let ((v value))
            (set! value (+ v 1))
            v))
        (lambda () (release-lock lock))))))
```

This poses another problem, however. Suppose we wish to write an atomic `(step-counters! counter ...)` procedure that increments each of the supplied counters by one; supplying a counter n times should have the effect of incrementing it by n. The naïve definition of it is this:

```
(define (step-counters! . counters)
  (for-each (lambda (counter) (counter))
            counters))
```

Obviously, though, this is not atomic, because each individual counter is locked when it is used, but not the whole iteration across them. To work around this, we might use an obfuscated control structure to allow nesting the locking of counters:

```
(define (make-counter value)
  (let ((lock (make-lock)))
    (lambda args
      (dynamic-wind
        (lambda () (obtain-lock lock))
        (lambda ()
          (if (null? args)
              (let ((v value))
                (set! value (+ v 1))
                v)
              ((car args))))
        (lambda () (release-lock lock))))))

(define (step-counters! . counters)
  (let loop ((cs counters))
    (if (null? cs)
        (for-each (lambda (counter) (counter))
                  counters)
        ((car cs) (lambda () (loop (cdr cs)))))))
```

Aside from the obvious matter of the obfuscation of the control structures used here, however, this has another problem: we cannot step one counter multiple times atomically. Though different locks can be nested, nesting is very dangerous, because accidentally obtaining a lock that is already obtained can cause deadlock, and there is no modular, transparent way to avoid this in the general case.

Instead, we can implement counters using optimistic concurrency to synchronize the shared data. The state of counters is kept explicitly in a cell (see Section 4.1.7 [Cells], page 44), in order to use a provisional accessor & modifier, as is necessary to make use of optimistic concurrency, and we surround with `call-ensuring-atomicity` any regions we wish to be atomic:

```
(define (make-counter initial)
  (let ((cell (make-cell initial)))
    (lambda ()
      (call-ensuring-atomicity
        (lambda ()
          (let ((value (provisional-cell-ref cell)))
            (provisional-cell-set! cell (+ value 1))
            value))))))

(define (step-counters! . counters)
  (call-ensuring-atomicity!
    (lambda ()
      (for-each (lambda (counter) (counter))
                counters))))
```

This approach has a number of advantages:

- The original control structure is preserved, only with provisional operators for shared memory access that we explicitly wish to be synchronized and with `call-ensuring-atomicity` wrapping the portions of code that we explicitly want to be atomic.
- Composition of transactions is entirely transparent; it is accomplished automatically simply by `call-ensuring-atomicity`.
- Transactions can be nested arbitrarily deeply, and there is no problem of accidentally locking the same resource again at a deeper nesting level to induce deadlock.
- No explicit mutual exclusion or blocking is necessary. Threads proceed without heed to others, but do not actually write data to the shared memory until its validity is ensured. There is no deadlock at all.

5.2.5 Low-level optimistic concurrency

Along with the higher-level operations described above, there are some lower-level primitives for finer control over optimistic concurrency.

`make-proposal` ⟶ *proposal* [procedure]
`current-proposal` ⟶ *proposal* [procedure]
`set-current-proposal!` *proposal* ⟶ *unspecified* [procedure]
`remove-current-proposal!` ⟶ *unspecified* [procedure]

`Make-proposal` creates a fresh proposal. `Current-proposal` returns the current thread's proposal. `Set-current-proposal!` sets the current thread's proposal to *proposal*. `Remove-current-proposal!` sets the current thread's proposal to `#f`.

`maybe-commit` ⟶ *boolean* [procedure]
`invalidate-current-proposal!` ⟶ *unspecified* [procedure]

`Maybe-commit` checks that the current thread's proposal is still valid. If it is, the proposal's writes are committed, and `maybe-commit` returns `#t`; if not, the current thread's proposal is set to `#f` and `maybe-commit` returns `#f`. `Invalidate-current-proposal!` causes an inconsistency in the current proposal by caching a read and then directly writing to the place that read was from.

`with-new-proposal` (*lose*) *body* ⟶ *values* [syntax]

Convenience for repeating a transaction. `With-new-proposal` saves the current proposal and will reinstates it when everything is finished. After saving the current proposal, it binds *lose* to a nullary procedure that installs a fresh proposal and that evaluates *body*; it then calls *lose*. Typically, the last thing, or close to last thing, that *body* will do is attempt to commit the current proposal, and, if that fails, call *lose* to retry. `With-new-proposal` expands to a form that returns the values that *body* returns.

This `retry-at-most` example tries running the transaction of *thunk*, and, if it fails to commit, retries at most *n* times. If the transaction is successfully committed before *n* repeated attempts, it returns true; otherwise, it returns false.

```
(define (retry-at-most n thunk)
  (with-new-proposal (lose)
    (thunk)
```

```
(cond ((maybe-commit) #t)
      ((zero? n)      #f)
      (else (set! n (- n 1))
            (lose)))))
```

5.3 Higher-level synchronization

This section details the various higher-level thread synchronization devices that Scheme48 provides.

5.3.1 Condition variables

Condition variables are multiple-assignment cells on which readers block. Threads may wait on condition variables; when some other thread assigns a condition variable, all threads waiting on it are revived. The `condvars` structure exports all of these condition-variable-related names.

In many concurrency systems, condition variables are operated in conjunction with mutual exclusion locks. On the other hand, in Scheme48, they are used in conjunction with its optimistic concurrency devices.

`make-condvar` [*id*] ⟶ *condvar* [procedure]
`condvar?` *object* ⟶ *boolean* [procedure]
Condition variable constructor & disjoint type predicate. *Id* is used purely for debugging.

`maybe-commit-and-wait-for-condvar` *condvar* ⟶ *boolean* [procedure]
`maybe-commit-and-set-condvar!` *condvar value* ⟶ *boolean* [procedure]
`Maybe-commit-and-wait-for-condvar` attempts to commit the current proposal. If the commit succeeded, the current thread is blocked on *condvar*, and when the current thread is woken up, `maybe-commit-and-wait-for-condvar` returns `#t`. If the commit did not succeed, `maybe-commit-and-wait-for-condvar` immediately returns `#f`. `Maybe-commit-and-set-condvar!` attempts to commit the current proposal as well. If it succeeds, it is noted that *condvar* has a value, *condvar*'s value is set to be *value*, and all threads waiting on *condvar* are woken up.

Note: Do not use these in atomic transactions as delimited by `call-ensuring-atomicity` *&c.*; see the note in Section 5.2 [Optimistic concurrency], page 77, on this matter for details.

`condvar-has-value?` *condvar* ⟶ *boolean* [procedure]
`condvar-value` *condvar* ⟶ *value* [procedure]
`Condvar-has-value?` tells whether or not *condvar* has been assigned. If it has been assigned, `condvar-value` accesses the value to which it was assigned.

`set-condvar-has-value?!` *condvar boolean* ⟶ *unspecified* [procedure]
`set-condvar-value!` *condvar value* ⟶ *unspecified* [procedure]
`Set-condvar-has-value?!` is used to tell whether or not *condvar* is assigned. `Set-condvar-value!` sets *condvar*'s value.

Note: `Set-condvar-has-value?!` should be used only with a second argument of `#f`. `Set-condvar-value!` is a very dangerous routine, and `maybe-commit-and-set-`

`condvar!` is what one should almost always use, except if one wishes to clean up after unassigning a condition variable.

5.3.2 Placeholders

Placeholders are similar to condition variables, except that they may be assigned only once; they are in general a much simpler mechanism for throw-away temporary synchronization devices. They are provided by the `placeholders` structure.

`make-placeholder` [*id*] ⟶ *placeholder* [procedure]
`placeholder?` *object* ⟶ *boolean* [procedure]
Placeholder constructor & disjoint type predicate. *Id* is used only for debugging purposes when printing placeholders.

`placeholder-value` *placeholder* ⟶ *value* [procedure]
`placeholder-set!` *placeholder value* ⟶ *unspecified* [procedure]
`Placeholder-value` blocks until *placeholder* is assigned, at which point it returns the value assigned. `Placeholder-set!` assigns *placeholder*'s value to *value*, awakening all threads waiting for *placeholder*. It is an error to assign a placeholder with `placeholder-set!` that has already been assigned.

5.3.3 Value pipes

Value pipes are asynchronous communication pipes between threads. The `value-pipes` structure exports these value pipe operations.

`make-pipe` [*size* [*id*]] ⟶ *value-pipe* [procedure]
`pipe?` *object* ⟶ *boolean* [procedure]
`Make-pipe` is the value pipe constructor. *Size* is a limit on the number of elements the pipe can hold at one time. *Id* is used for debugging purposes only in printing pipes. `Pipe?` is the disjoint type predicate for value pipes.

`empty-pipe?` *pipe* ⟶ *boolean* [procedure]
`empty-pipe!` *pipe* ⟶ *unspecified* [procedure]
`Empty-pipe?` returns `#t` if *pipe* has no elements in it and `#f` if not. `Empty-pipe!` removes all elements from `pipe`.

`pipe-read!` *pipe* ⟶ *value* [procedure]
`pipe-maybe-read!` *pipe* ⟶ *value or* `#f` [procedure]
`pipe-maybe-read?!` *pipe* ⟶ [*boolean value*] [procedure]
`Pipe-read!` reads a value from *pipe*, removing it from the queue. It blocks if there are no elements available in the queue. `Pipe-maybe-read!` attempts to read & return a single value from *pipe*; if no elements are available in its queue, it instead returns `#f`. `Pipe-maybe-read?!` does similarly, but it returns two values: a boolean, signifying whether or not a value was read; and the value, or `#f` if no value was read. `Pipe-maybe-read?!` is useful when *pipe* may contain the value `#f`.

`pipe-write!` *pipe value* ⟶ *unspecified* [procedure]
`pipe-push!` *pipe value* ⟶ *unspecified* [procedure]

`pipe-maybe-write!` *pipe value* $\longrightarrow$ *boolean* [procedure]

`Pipe-write!` attempts to add *value* to *pipe*'s queue. If *pipe*'s maximum size, as passed to `make-pipe` when constructing the pipe, is either `#f` or greater than the number of elements in *pipe*'s queue, `pipe-write!` will not block; otherwise it will block until a space has been made available in the pipe's queue by another thread reading from it. `Pipe-push!` does similarly, but, in the case where the pipe is full, it pushes the first element to be read out of the pipe. `Pipe-maybe-write!` is also similar to `pipe-write!`, but it returns `#t` if the pipe was not full, and it *immediately* returns `#f` if the pipe was full.

5.4 Concurrent ML

Scheme48 provides a high-level event synchronization facility based on on Reppy's *Concurrent ML* [Reppy 99]. The primary object in CML is the *rendezvous*[1], which represents a point of process synchronization. A rich library for manipulating rendezvous and several useful, high-level synchronization abstractions are built atop rendezvous.

5.4.1 Rendezvous concepts

When access to a resource must be synchronized between multiple processes, for example to transmit information from one process to another over some sort of communication channel, the resource provides a *rendezvous* to accomplish this, which represents a potential point of synchronization between processes. The use of rendezvous occurs in two stages: *synchronization* and *enablement*. Note that creation of rendezvous is an unrelated matter, and it does not (or should not) itself result in any communication or synchronization between processes.

When a process requires an external resource for which it has a rendezvous, it *synchronizes* that rendezvous. This first polls whether the resource is immediately available; if so, the rendezvous is already *enabled*, and a value from the resource is immediately produced from the synchronization. Otherwise, the synchronization of the rendezvous is recorded somehow externally, and the process is blocked until the rendezvous is enabled by an external entity, usually one that made the resource available. Rendezvous may be reüsed arbitrarily many times; the value produced by an enabled, synchronized rendezvous is not cached. Note, however, that the construction of a rendezvous does not (or should not) have destructive effect, such as sending a message to a remote server or locking a mutex; the only destructive effects should be incurred at synchronization or enablement time. For effecting initialization prior to the synchronization of a rendezvous, see below on *delayed rendezvous*.

Rendezvous may consist of multiple rendezvous choices, any of which may be taken when enabled but only one of which actually is. If, when a composite rendezvous is initially synchronized, several components are immediately enabled, each one has a particular numeric priority which is used to choose among them. If several are tied for the highest priority, a random one is chosen. If none is enabled when the choice is synchronized, however, the synchronizer process is suspended until the first one is enabled and revives the process. When this happens, any or all of the other rendezvous components may receive a negative acknowledgement; see below on *delayed rendezvous with negative acknowledgement*.

[1] In the original CML, these were called *events*, but that term was deemed too overloaded and confusing when Scheme48's library was developed.

A rendezvous may also be a rendezvous *wrapped* with a procedure, which means that, when the internal rendezvous becomes enabled, the wrapper one also becomes enabled, and the value it produces is the result of applying its procedure to the value that the internal rendezvous produced. This allows the easy composition of complex rendezvous from simpler ones, and it also provides a simple mechanism for performing different actions following the enablement of different rendezvous, rather than conflating the results of several possible rendezvous choices into one value and operating on that (though this, too, can be a useful operation).

5.4.2 Delayed rendezvous

A rendezvous may be *delayed*, which means that its synchronization requires some processing that could not or would not be reasonable to perform at its construction. It consists of a nullary procedure to generate the actual rendezvous to synchronize when the delayed rendezvous is itself synchronized.

For example, a rendezvous for generating unique identifiers, by sending a request over a network to some server and waiting for a response, could not be constructed by waiting for a response from the server, because that may block, which should not occur until synchronization. It also could not be constructed by first sending a request to the server at all, because that would have a destructive effect, which is not meant to happen when creating a rendezvous, only when synchronizing or enabling one.

Instead, the unique identifier rendezvous would be implemented as a delayed rendezvous that, when synchronized, would send a request to the server and generate a rendezvous for the actual synchronization that would become enabled on receiving the server's response.

5.4.2.1 Negative acknowledgements

Delayed rendezvous may also receive negative acknowledgements. Rather than a simple nullary procedure being used to generate the actual rendezvous for synchronization, the procedure is unary, and it is passed a *negative acknowledgement rendezvous*, or *nack* for short. This nack is enabled if the actual rendezvous was not chosen among a composite group of rendezvous being synchronized. This allows not only delaying initialization of rendezvous until necessary but also aborting or rescinding initialized transactions if their rendezvous are unchosen and therefore unused.

For example, a complex database query might be the object of some rendezvous, but it is pointless to continue constructing the result if that rendezvous is not chosen. A nack can be used to prematurely abort the query to the database if another rendezvous was chosen in the stead of that for the database query.

5.4.3 Rendezvous combinators

The `rendezvous` structure exports several basic rendezvous combinators.

`never-rv` ⟶ *rendezvous* [Constant]

A rendezvous that is never enabled. If synchronized, this will block the synchronizing thread indefinitely.

`always-rv` *value* ⟶ *rendezvous* [procedure]

Returns a rendezvous that is always enabled with the given value. This rendezvous will never block the synchronizing thread.

`guard` *rv-generator* ⟶ *rendezvous* [procedure]
`with-nack` *rv-generator* ⟶ *rendezvous* [procedure]

`Guard` returns a delayed rendezvous, generated by the given procedure *rv-generator*, which is passed zero arguments whenever the resultant rendezvous is synchronized. `With-nack` returns a delayed rendezvous for which a negative acknowledgement rendezvous is constructed. If the resultant rendezvous is synchronized as a part of a composite rendezvous, the procedure `rv-generator` is passed a nack for the synchronization, and it returns the rendezvous to actually synchronize. If the delayed rendezvous was synchronized as part of a composite group of rendezvous, and another rendezvous among that group is enabled and chosen first, the nack is enabled.

`choose` *rendezvous ...* ⟶ *composite-rendezvous* [procedure]

Returns a rendezvous that, when synchronized, synchronizes all of the given components, and chooses only the first one to become enabled, or the highest priority one if there are any that are already enabled. If any of the rendezvous that were not chosen when the composite became enabled were delayed rendezvous with nacks, their nacks are enabled.

`wrap` *rendezvous procedure* ⟶ *rendezvous* [procedure]

Returns a rendezvous equivalent to *rendezvous* but wrapped with *procedure*, so that, when the resultant rendezvous is synchronized, *rendezvous* is transitively synchronized, and when *rendezvous* is enabled, the resultant rendezvous is also enabled, with the value that *procedure* returns when passed the value produced by *rendezvous*.

```
(sync (wrap (always-rv 4)
            (lambda (x) (* x x))))        ⟶ 16
```

`sync` *rendezvous* ⟶ *value* (*may block*) [procedure]
`select` *rendezvous ...* ⟶ *value* (*may block*) [procedure]

`Sync` and `select` synchronize rendezvous. `Sync` synchronizes a single one; `select` synchronizes any from the given set of them. `Select` is equivalent to `(sync (apply choose` *`rendezvous`* `...))`, but it may be implemented more efficiently.

5.4.3.1 Timing rendezvous

The `rendezvous-time` structure exports two constructors for rendezvous that become enabled only at a specific time or after a delay in time.

`at-real-time-rv` *milliseconds* ⟶ *rendezvous* [procedure]
`after-time-rv` *milliseconds* ⟶ *rendezvous* [procedure]

`At-real-time-rv` returns a rendezvous that becomes enabled at the time *milliseconds* relative to the start of the Scheme program. `After-time-rv` returns a rendezvous that becomes enabled at least *milliseconds* after synchronization (*not* construction).

5.4.4 Rendezvous communication channels

5.4.4.1 Synchronous channels

The `rendezvous-channels` structure provides a facility for *synchronous channels*: channels for communication between threads such that any receiver blocks until another thread sends a message, or any sender blocks until another thread receives the sent message. In CML, synchronous channels are also called merely 'channels.'

`make-channel` ⟶ *channel* [procedure]
`channel?` *object* ⟶ *boolean* [procedure]

`Make-channel` creates and returns a new channel. `Channel?` is the disjoint type predicate for channels.

`send-rv` *channel message* ⟶ *rendezvous* [procedure]
`send` *channel message* ⟶ *unspecified* (*may block*) [procedure]

`Send-rv` returns a rendezvous that, when synchronized, becomes enabled when a reception rendezvous for *channel* is synchronized, at which point that reception rendezvous is enabled with a value of *message*. When enabled, the rendezvous returned by `send-rv` produces an unspecified value. `Send` is like `send-rv`, but it has the effect of immediately synchronizing the rendezvous, so it therefore may block, and it does not return a rendezvous; `(send` *`channel message`*`)` is equivalent to `(sync (send-rv` *`channel message`*`))`.

`receive-rv` *channel* ⟶ *rendezvous* [procedure]
`receive` *channel* ⟶ *value* (*may block*) [procedure]

`Receive-rv` returns a rendezvous that, when synchronized, and when a sender rendezvous for *channel* with some message is synchronized, becomes enabled with that message, at which point the sender rendezvous is enabled with an unspecified value. `Receive` is like `receive-rv`, but it has the effect of immediately synchronizing the reception rendezvous, so it therefore may block, and it does not return the rendezvous but rather the message that was sent; `(receive` *`channel`*`)` is equivalent to `(sync (receive-rv` *`channel`*`))`.

5.4.4.2 Asynchronous channels

The `rendezvous-async-channels` provides an *asynchronous channel*[2] facility. Like synchronous channels, any attempts to read from an asynchronous channel will block if there are no messages waiting to be read. Unlike synchronous channels, however, sending a message will never block. Instead, a queue of messages or a queue of recipients is maintained: if a message is sent and there is a waiting recipient, the message is delivered to that recipient; otherwise it is added to the queue of messages. If a thread attempts to receive a message from an asynchronous channel and there is a pending message, it receives that message; otherwise it adds itself to the list of waiting recipients and then blocks.

Note: Operations on synchronous channels from the structure `rendezvous-channels` do not work on asynchronous channels.

`make-async-channel` ⟶ *async-channel* [procedure]
`async-channel?` *obj* ⟶ *boolean* [procedure]

`Make-async-channel` creates and returns an asynchronous channel. `Async-channel?` is the disjoint type predicate for asynchronous channels.

`receive-async-rv` *channel* ⟶ *rendezvous* [procedure]
`receive-async` *channel* ⟶ *value* (*may block*) [procedure]

`Receive-async-rv` returns a rendezvous that, when synchronized, becomes enabled when a message is available in *channel*'s queue of messages. `Receive-async`

[2] Known as *mailboxes* in Reppy's original CML.

has the effect of immediately synchronizing such a rendezvous and, when the rendezvous becomes enabled, returning the value itself, rather than the rendezvous; `(receive-async` *`channel`*`)` is equivalent to `(sync (receive-async-rv` *`channel`*`))`.

`send-async` *channel message* $\longrightarrow$ *unspecified* [procedure]
Sends a message to the asynchronous channel *channel*. Unlike the synchronous channel `send` operation, this procedure never blocks arbitrarily long.[3] There is, therefore, no need for a `send-async-rv` like the `send-rv` for synchronous channels. If there is a waiting message recipient, the message is delivered to that recipient; otherwise, it is added to the channel's message queue.

5.4.5 Rendezvous-synchronized cells

5.4.5.1 Placeholders: single-assignment cells

Placeholders[4] are single-assignment cells on which readers block until they are assigned.

Note: These placeholders are disjoint from and incompatible with the placeholder mechanism provided in the `placeholders` structure, and attempts to apply operations on one to values of the other are errors.

`make-placeholder` [*id*] $\longrightarrow$*empty placeholder* [procedure]
`placeholder?` *object* $\longrightarrow$ *boolean* [procedure]
`Make-placeholder` creates and returns a new, empty placeholder. *Id* is used only for debugging purposes; it is included in the printed representation of the placeholder. `Placeholder?` is the disjoint type predicate for placeholders.

`placeholder-value-rv` *placeholder* $\longrightarrow$ *rendezvous* [procedure]
`placeholder-value` *placeholder* $\longrightarrow$ *value* (*may block*) [procedure]
`Placeholder-value-rv` returns a rendezvous that, when synchronized, becomes enabled when *placeholder* has a value, with that value. `Placeholder-value` has the effect of immediately synchronizing such a rendezvous, and it returns the value directly, but possibly after blocking.

`placeholder-set!` *placeholder value* $\longrightarrow$ *unspecified* [procedure]
Sets *placeholder*'s value to be *value*, and enables all rendezvous for *placeholder*'s value with that value. It is an error if *placeholder* has already been assigned.

5.4.5.2 Jars: multiple-assignment cells

Jars[5] are multiple-assignment cells on which readers block. Reading from a full jar has the effect of emptying it, enabling the possibility of subsequent assignment, unlike placeholders; and jars may be assigned multiple times, but, like placeholders, only jars that are empty may be assigned.

---

[3] However, asynchronous channels are implemented by a thread that manages two synchronous channels (one for sends & one for receives), so this may block briefly if the thread is busy receiving other send or receive requests.

[4] Called *I-variables* in Reppy's CML, and *I-structures* in ID-90.

[5] Termed *M-variables* in Reppy's CML.

`make-jar` [*id*] ⟶ *empty jar* [procedure]
`jar?` *object* ⟶ *boolean* [procedure]

`Make-jar` creates and returns a new, empty jar. *Id* is used only for debugging purposes; it is included in the printed representation of the jar. `Jar?` is the disjoint type predicate for jars.

`jar-take-rv` *jar* ⟶ *rendezvous* [procedure]
`jar-take` *jar* ⟶ *value* (*may block*) [procedure]

`Jar-take-rv` returns a rendezvous that, when synchronized, becomes enabled when *jar* has a value, which is what value the rendezvous becomes enabled with; when that rendezvous is enabled, it also removes the value from *jar*, putting the jar into an empty state. `Jar-take` has the effect of synchronizing such a rendezvous, may block because of that, and returns the value of the jar directly, not a rendezvous.

`jar-put!` *jar value* ⟶ *unspecified* [procedure]

`Jar-put!` puts *value* into the empty jar *jar*. If any taker rendezvous are waiting, the first is enabled with the value, and the jar is returned to its empty state; otherwise, the jar is put in the full state. `Jar-put!` is an error if applied to a full jar.

5.4.6 Concurrent ML to Scheme correspondence

CML name	Scheme name
structure `CML`	structure `threads`
`version`	(no equivalent)
`banner`	(no equivalent)
`spawnc`	(no equivalent; use `spawn` and `lambda`)
`spawn`	`spawn`
`yield`	`relinquish-timeslice`
`exit`	`terminate-current-thread`
`getTid`	`current-thread`
`sameTid`	`eq?` (R5RS)
`tidToString`	(no equivalent; use the writer)
	structure `threads-internal`
`hashTid`	`thread-uid`
	structure `rendezvous`
`wrap`	`wrap`
`guard`	`guard`
`withNack`	`with-nack`
`choose`	`choose`
`sync`	`sync`
`select`	`select`
`never`	`never-rv`
`alwaysEvt`	`always-rv`
`joinEvt`	(no equivalent)
	structure `rendezvous-channels`
`channel`	`make-channel`
`sameChannel`	`eq?` (R5RS)
`send`	`send`

`recv`	`receive`
`sendEvt`	`send-rv`
`recvEvt`	`receive-rv`
`sendPoll`	(no equivalent)
`recvPoll`	(no equivalent)
	structure `rendezvous-time`
`timeOutEvt`	`after-time-rv`
`atTimeEvt`	`at-real-time-rv`
structure `SyncVar`	structure `rendezvous-placeholders`
exception `Put`	(no equivalent)
`iVar`	`make-placeholder`
`iPut`	`placeholder-set!`
`iGet`	`placeholder-value`
`iGetEvt`	`placeholder-value-rv`
`iGetPoll`	(no equivalent)
`sameIVar`	`eq?` (R5RS)
	structure `jars`
`mVar`	`make-jar`
`mVarInit`	(no equivalent)
`mPut`	`jar-put!`
`mTake`	`jar-take`
`mTakeEvt`	`jar-take-rv`
`mGet`	(no equivalent)
`mGetEvt`	(no equivalent)
`mTakePoll`	(no equivalent)
`mGetPoll`	(no equivalent)
`mSwap`	(no equivalent)
`mSwapEvt`	(no equivalent)
`sameMVar`	`eq?` (R5RS)
structure `Mailbox`	structure `rendezvous-async-channels`
`mailbox`	`make-async-channel`
`sameMailbox`	`eq?` (R5RS)
`send`	`send-async`
`recv`	`receive-async`
`recvEvt`	`receive-async-rv`
`recvPoll`	(no equivalent)

5.5 Pessimistic concurrency

While Scheme48's primitive thread synchronization mechanisms revolve around optimistic concurrency, Scheme48 still provides the more well-known mechanism of pessimistic concurrency, or mutual exclusion, with locks. Note that Scheme48's pessimistic concurrency facilities are discouraged, and very little of the system uses them (at the time this documentation was written, none of the system uses locks), and the pessimistic concurrency libraries are limited to just locks; condition variables are integrated only with optimistic concurrency. Except for inherent applications of pessimistic concurrency, it is usually better to use optimistic concurrency in Scheme48.

These names are exported by the `locks` structure.

`make-lock` ⟶ *lock* [procedure]
`lock?` ⟶ *boolean* [procedure]
`obtain-lock` *lock* ⟶ *unspecified* [procedure]
`maybe-obtain-lock` *lock* ⟶ *boolean* [procedure]
`release-lock` *lock* ⟶ *unspecified* [procedure]

`Make-lock` creates a new lock in the 'released' lock state. `Lock?` is the disjoint type predicate for locks. `Obtain-lock` atomically checks to see if *lock* is in the 'released' state: if it is, *lock* is put into the 'obtained' lock state; otherwise, `obtain-lock` waits until *lock* is ready to be obtained, at which point it is put into the 'obtained' lock state. `Maybe-obtain-lock` atomically checks to see if *lock* is in the 'released' state: if it is, *lock* is put into the 'obtained' lock state, and `maybe-obtain-lock` returns `#t`; if it is in the 'obtained' state, `maybe-obtain-lock` immediately returns `#f`. `Release-lock` sets *lock*'s state to be 'released,' letting the next thread waiting to obtain it do so.

5.6 Custom thread synchronization

Along with several useful thread synchronization abstraction facilities built-in to Scheme48, there is also a simple and lower-level mechanism for suspending & resuming threads. The following bindings are exported from the `threads-internal` structure.

Threads have a field for a cell (see Section 4.1.7 [Cells], page 44) that is used when the thread is suspended. When it is ready to run, it is simply `#f`. Suspending a thread involves setting its cell to a cell accessible outside, so the thread can later be awoken. When the thread is awoken, its cell field and the contents of the cell are both set to `#f`. Often, objects involved in the synchronization of threads will have a queue (see Section 4.1.8 [Queues], page 45) of thread cells. There are two specialized operations on thread cell queues that simplify filtering out cells of threads that have already been awoken.

`maybe-commit-and-block` *cell* ⟶ *boolean* [procedure]
`maybe-commit-and-block-on-queue` ⟶ *boolean* [procedure]

These attempt to commit the current proposal. If the commit fails, they immediately return `#f`. Otherwise, they suspend the current thread. `Maybe-commit-and-block` first sets the current thread's cell to *cell*, which should contain the current thread. `Maybe-commit-and-block-on-queue` adds a cell containing the current thread to *queue* first. When the current thread is finally resumed, these return `#t`.

`maybe-commit-and-make-ready` *thread-or-queue* ⟶ *boolean* [procedure]

Attempts to commit the current proposal. If the commit fails, this returns `#f`. Otherwise, `maybe-commit-and-make-ready` awakens the specified thread[s] by clearing the thread/each thread's cell and sending a message to the relevant scheduler[s] and returns `#t`. If *thread-or-queue* is a thread, it simply awakens that; if it is a queue, it empties the queue and awakens each thread in it.

`maybe-dequeue-thread!` *thread-cell-queue* ⟶ *thread or boolean* [procedure]
`thread-queue-empty?` *thread-cell-queue* ⟶ *boolean* [procedure]

`Maybe-dequeue-thread!` returns the next thread cell's contents in the queue of thread cells *thread-cell-queue*. It removes cells that have been emptied, *i.e.* whose threads

have already been awoken. `Thread-queue-empty?` returns `#t` if there are no cells in *thread-cell-queue* that contain threads, *i.e.* threads that are still suspended. It too removes cells that have been emptied.

For example, the definition of placeholders (see Section 5.3 [Higher-level synchronization], page 83) is presented here. Placeholders contain two fields: the cached value (set when the placeholder is set) & a queue of threads waiting (set to `#f` when the placeholder is assigned).

```
(define-synchronized-record-type placeholder :placeholder
  (really-make-placeholder queue)
  (value queue)                          ; synchronized fields
  placeholder?
  (queue placeholder-queue set-placeholder-queue!)
  (value placeholder-real-value set-placeholder-value!))

(define (make-placeholder)
  (really-make-placeholder (make-queue)))

(define (placeholder-value placeholder)
  ;; Set up a new proposal for the transaction.
  (with-new-proposal (lose)
    (cond ((placeholder-queue placeholder)
           ;; There's a queue of waiters.  Attempt to commit the
           ;; proposal and block.  We'll be added to the queue if the
           ;; commit succeeds; if it fails, retry.
           => (lambda (queue)
                (or (maybe-commit-and-block-on-queue queue)
                    (lose))))))
  ;; Once our thread has been awoken, the placeholder will be set.
  (placeholder-real-value placeholder))

(define (placeholder-set! placeholder value)
  ;; Set up a new proposal for the transaction.
  (with-new-proposal (lose)
    (cond ((placeholder-queue placeholder)
           => (lambda (queue)
                ;; Clear the queue, set the value field.
                (set-placeholder-queue! placeholder #f)
                (set-placeholder-value! placeholder value)
                ;; Attempt to commit our changes and awaken all of the
                ;; waiting threads.  If the commit fails, retry.
                (if (not (maybe-commit-and-make-ready queue))
                    (lose))))
          (else
           ;; Someone assigned it first.  Since placeholders are
           ;; single-assignment cells, this is an error.
           (error "placeholder is already assigned"
                  placeholder
```

```
                    (placeholder-real-value placeholder))))))
```

6 Libraries

This chapter details a number of useful libraries built-in to Scheme48.

6.1 Boxed bitwise-integer masks

Scheme48 provides a facility for generalized boxed bitwise-integer masks. Masks represent sets of elements. An element is any arbitrary object that represents an index into a bit mask; mask types are parameterized by an isomorphism between elements and their integer indices. Usual abstract set operations are available on masks. The mask facility is divided into two parts: the `mask-types` structure, which provides the operations on the generalized mask type descriptors; and the `masks` structure, for the operations on masks themselves.

6.1.1 Mask types

`make-mask-type` *name elt? index->elt elt->index size* $\longrightarrow$ *mask-type* [procedure]
`mask-type?` *object* $\longrightarrow$ *boolean* [procedure]
`mask?` *object* $\longrightarrow$ *boolean* [procedure]

`Make-mask-type` constructs a mask type with the given name. Elements of this mask type must satisfy the predicate *elt?*. *Integer->elt* is a unary procedure that maps bit mask indices to possible set elements; *elt->integer* maps possible set elements to bit mask indices. *Size* is the number of possible elements of masks of the new type, *i.e.* the number of bits needed to represent the internal bit mask. `Mask?` is the disjoint type predicate for mask objects.

`mask-type` *mask* $\longrightarrow$ *mask-type* [procedure]
`mask-has-type?` *mask type* $\longrightarrow$ *boolean* [procedure]

`Mask-type` returns *mask*'s type. `Mask-has-type?` returns `#t` if *mask*'s type is the mask type *type* or `#f` if not.

The `mask-types` structure, not the `masks` structure, exports `mask?` and `mask-has-type?`: it is expected that programmers who implement mask types will define type predicates for masks of their type based on `mask?` and `mask-has-type?`, along with constructors *&c.* for their masks.

`integer->mask` *type integer* $\longrightarrow$ *mask* [procedure]
`list->mask` *type elts* $\longrightarrow$ *mask* [procedure]

`Integer->mask` returns a mask of type *type* that contains all the possible elements *e* of the type *type* such that the bit at *e*'s index is set. `List->mask` returns a mask whose type is *type* containing all of the elements in the list *elts*.

6.1.2 Masks

`mask->integer` *mask* $\longrightarrow$ *integer* [procedure]
`mask->list` *mask* $\longrightarrow$ *element-list* [procedure]

`Mask->integer` returns the integer bit set that *mask* uses to represent the element set. `Mask->list` returns a list of all the elements that *mask* contains.

`mask-member?` *mask elt* $\longrightarrow$ *boolean* [procedure]
`mask-set` *mask elt ...* $\longrightarrow$ *mask* [procedure]
`mask-clear` *mask elt ...* $\longrightarrow$ *mask* [procedure]

`Mask-member?` returns true if *elt* is a member of the mask *mask*, or `#f` if not. `Mask-set` returns a mask with all the elements in *mask* as well as each *elt* .... `Mask-clear` returns a mask with all the elements in *mask* but with none of *elt* ....

`mask-union` $mask_1$ $mask_2$ ... $\longrightarrow$ *mask* [procedure]
`mask-intersection` $mask_1$ $mask_2$ ... $\longrightarrow$ *mask* [procedure]
`mask-subtract` $mask_a$ $mask_b$ $\longrightarrow$ *mask* [procedure]
`mask-negate` *mask* $\longrightarrow$ *mask* [procedure]

Set operations on masks. `Mask-union` returns a mask containing every element that is a member of any one of its arguments. `Mask-intersection` returns a mask containing every element that is a member of every one of its arguments. `Mask-subtract` returns a mask of every element that is in $mask_a$ but not also in $mask_b$. `Mask-negate` returns a mask whose members are every possible element of *mask*'s type that is not in *mask*.

6.2 Enumerated/finite types and sets

(This section was derived from work copyrighted © 1993–2005 by Richard Kelsey, Jonathan Rees, and Mike Sperber.)

The structure `finite-types` has two macros for defining *finite* or *enumerated record types*. These are record types for which there is a fixed set of instances, all of which are created at the same time as the record type itself. Also, the structure `enum-sets` has several utilities for building sets of the instances of those types, although it is generalized beyond the built-in enumerated/finite type device. There is considerable overlap between the boxed bitwise-integer mask library (see Section 6.1 [Boxed bitwise-integer masks], page 95) and the enumerated set facility.

6.2.1 Enumerated/finite types

`define-enumerated-type` [syntax]

```
(define-enumerated-type dispatcher type
  predicate
  instance-vector
  name-accessor
  index-accessor
  (instance-name
   ...))
```

This defines a new record type, to which *type* is bound, with as many instances as there are *instance-name*s. *Predicate* is defined to be the record type's predicate. *Instance-vector* is defined to be a vector containing the instances of the type in the same order as the *instance-name* list. *Dispatcher* is defined to be a macro of the form (*dispatcher instance-name*); it evaluates to the instance with the given name, which is resolved at macro-expansion time. *Name-accessor* & *index-accessor* are defined to be unary procedures that return the symbolic name & index into the instance vector, respectively, of the new record type's instances.

For example,

```
(define-enumerated-type colour :colour
  colour?
  colours
  colour-name
  colour-index
  (black white purple maroon))

(colour-name (vector-ref colours 0))    ⇒ black
(colour-name (colour white))            ⇒ white
(colour-index (colour purple))          ⇒ 2
```

`define-finite-type` [syntax]

```
(define-finite-type dispatcher type
  (field-tag ...)
  predicate
  instance-vector
  name-accessor
  index-accessor
  (field-tag accessor [modifier])
  ...
  ((instance-name field-value ...)
   ...))
```

This is like `define-enumerated-type`, but the instances can also have added fields beyond the name and the accessor. The first list of field tags lists the fields that each instance is constructed with, and each instance is constructed by applying the unnamed constructor to the initial field values listed. Fields not listed in the first field tag list must be assigned later.

For example,

```
(define-finite-type colour :colour
  (red green blue)
  colour?
  colours
  colour-name
  colour-index
  (red   colour-red)
  (green colour-green)
  (blue  colour-blue)
  ((black    0   0   0)
   (white  255 255 255)
   (purple 160  32 240)
   (maroon 176  48  96)))

(colour-name (colour black))            ⇒ black
(colour-name (vector-ref colours 1))    ⇒ white
(colour-index (colour purple))          ⇒ 2
```

```
(colour-red (colour maroon))            ⇒ 176
```

6.2.2 Sets over enumerated types

`define-enum-set-type` [syntax]

```
(define-enum-set-type set-syntax type
                      predicate
                      list->x-set
  element-syntax
  element-predicate
  element-vector
  element-index)
```

This defines *set-syntax* to be a syntax for constructing sets, *type* to be an object that represents the type of enumerated sets, *predicate* to be a predicate for those sets, and *list->x-set* to be a procedure that converts a list of elements into a set of the new type.

Element-syntax must be the name of a macro for constructing set elements from names (akin to the *dispatcher* argument to the `define-enumerated-type` & `define-finite-type` forms). *Element-predicate* must be a predicate for the element type, *element-vector* a vector of all values of the element type, and *element-index* a procedure that returns the index of an element within *element-vector*.

`enum-set->list` *enum-set* ⟶ *element list* [procedure]
`enum-set-member?` *enum-set element* ⟶ *boolean* [procedure]
`enum-set=?` $enum\text{-}set_a$ $enum\text{-}set_b$ ⟶ *boolean* [procedure]
`enum-set-union` $enum\text{-}set_a$ $enum\text{-}set_b$ ⟶ *enum-set* [procedure]
`enum-set-intersection` $enum\text{-}set_a$ $enum\text{-}set_b$ ⟶ *enum-set* [procedure]
`enum-set-negation` *enum-set* ⟶ *enum-set* [procedure]

`Enum-set->list` returns a list of elements within *enum-set*. `Enum-set-member?` tests whether *element* is a member of *enum-set*. `Enum-set=?` tests whether two enumerated sets are equal, *i.e.* contain all the same elements. The other procedures perform standard set algebra operations on enumerated sets. It is an error to pass an element that does not satisfy *enum-set*'s predicate to `enum-set-member?` or to pass two enumerated sets of different types to `enum-set=?` or the enumerated set algebra operators.

Here is a simple example of enumerated sets built atop the enumerated types described in the previous section:

```
(define-enumerated-type colour :colour
  colour?
  colours
  colour-name
  colour-index
  (red blue green))

(define-enum-set-type colour-set :colour-set
                      colour-set?
```

```
                        list->colour-set
  colour colour? colours colour-index)

(enum-set->list (colour-set red blue))
    ⇒ (#{Colour red} #{Colour blue})
(enum-set->list (enum-set-negation (colour-set red blue)))
    ⇒ (#{Colour green})
(enum-set-member? (colour-set red blue) (colour blue))
    ⇒ #t
```

6.3 Macros for writing loops

(This section was derived from work copyrighted © 1993–2005 by Richard Kelsey, Jonathan Rees, and Mike Sperber.)

`Iterate` & `reduce` are extensions of named-`let` for writing loops that walk down one or more sequences, such as the elements of a list or vector, the characters read from a port, or an arithmetic series. Additional sequences can be defined by the user. `Iterate` & `reduce` are exported by the structure `reduce`.

6.3.1 Main looping macros

`iterate` [syntax]

```
(iterate loop-name ((seq-type elt-var arg ...)
                    ...)
    ((state-var init)
     ...)
  body
  [tail-exp])
```

`Iterate` steps the *elt-var*s in parallel through the sequences, while each *state-var* has the corresponding *init* for the first iteration and later values supplied by the body. If any sequence has reached the limit, the value of the `iterate` expression is the value of *tail-exp*, if present, or the current values of the *state-var*s, returned as multiple values. If no sequence has reached its limit, *body* is evaluated and either calls *loop-name* with new values for the *state-var*s or returns some other value(s).

The *loop-name* and the *state-var*s & *init*s behave exactly as in named-`let`, in that *loop-name* is bound only in the scope of *body*, and each *init* is evaluated parallel in the enclosing scope of the whole expression. Also, the arguments to the sequence constructors will be evaluated in the enclosing scope of the whole expression, or in an extension of that scope peculiar to the sequence type. The named-`let` expression

```
(let loop-name ((state-var init) ...)
  body
  ...)
```

is equivalent to an iterate expression with no sequences (and with an explicit `let` wrapped around the body expressions to take care of any internal definitions):

```
(iterate loop-name ()
    ((state-var init) ...)
```

```
(let () body ...))
```

The *seq-type*s are keywords (actually, macros of a particular form, which makes it easy to add additional types of sequences; see below). Examples are `list*`, which walks down the elements of a list, and `vector*`, which does the same for vectors. For each iteration, each *elt-var* is bound to the next element of the sequence. The *arg*s are supplied to the sequence processors as other inputs, such as the list or vector to walk down.

If there is a *tail-exp*, it is evaluated when the end of one or more sequences is reached. If the body does not call *loop-name*, however, the *tail-exp* is not evaluated. Unlike named-`let`, the behaviour of a non-tail-recursive call to *loop-name* is unspecified, because iterating down a sequence may involve side effects, such as reading characters from a port.

`reduce` [syntax]

```
(reduce ((seq-type elt-var arg ...)
         ...)
    ((state-var init)
     ...)
  body
  [tail-exp])
```

If an `iterate` expression is not meant to terminate before a sequence has reached its end, the body will always end with a tail call to *loop-name*. `Reduce` is a convenient macro that makes this common case explicit. The syntax of `reduce` is the same as that of `iterate`, except that there is no *loop-name*, and the body updates the state variables by returning multiple values in the stead of passing the new values to *loop-name*: the body must return as many values as there are state variables. By special dispension, if there are no state variables, then the body may return any number of values, all of which are ignored.

The value(s) returned by an instance of `reduce` is (are) the value(s) returned by the *tail-exp*, if present, or the current value(s) of the state variables when the end of one or more sequences is reached.

A `reduce` expression can be rewritten as an equivalent `iterate` expression by adding a *loop-name* and a wrapper for the body that calls the *loop-name*:

```
(iterate loop ((seq-type elt-var arg ...)
               ...)
    ((state-var init)
     ...)
  (call-with-values (lambda () body)
    loop)
  [tail-exp])
```

6.3.2 Sequence types

`list*` *elt-var list* [sequence type]
`vector*` *elt-var vector* [sequence type]
`string*` *elt-var string* [sequence type]

`count*` *elt-var start* [*end* [*step*]] [sequence type]
`input*` *elt-var input-port reader-proc* [sequence type]
`stream*` *elt-var proc initial-seed* [sequence type]

For lists, vectors, & strings, the *elt-var* is bound to the successive elements of the list or vector, or the successive characters of the string.

For `count*`, the *elt-var* is bound to the elements of the sequence `start, start + step, start + 2*step, ..., end`, inclusive of *start* and exclusive of *end*. The default *step* is `1`, and the sequence does not terminate if no *end* is given or if there is no $N > 0$ such that *end* = *start* + *N*step. (`=` is used to test for termination.) For example, `(count* i 0 -1)` does not terminate because it begins past the *end* value, and `(count* i 0 1 2)` does not terminate because it skips over the *end* value.

For `input*`, the elements are the results of successive applications of *reader-proc* to *input-port*. The sequence ends when the *reader-proc* returns an end-of-file object, *i.e.* a value that satisfies `eof-object?`.

For `stream*`, the *proc* receives the current seed as an argument and must return two values, the next value of the sequence & the next seed. If the new seed is `#f`, then the previous element was the last one. For example, `(list* elt list)` is the same as

```
(stream* elt
         (lambda (list)
           (if (null? list)
               (values 'ignored #f)
               (values (car list) (cdr list))))
         list)
```

6.3.3 Synchronous sequences

When using the sequence types described above, a loop terminates when any of its sequences terminate. To help detect bugs, it is useful to also have sequence types that check whether two or more sequences end on the same iteration. For this purpose, there is a second set of sequence types called *synchronous sequences*. Synchronous sequences are like ordinary asynchronous sequences in every respect except that they cause an error to be signalled if a loop is terminated by a synchronous sequence and some other synchronous sequence did not reach its end on the same iteration.

Sequences are checked for termination in order from left to right, and if a loop is terminated by an asynchronous sequence no further checking is done.

`list%` *elt-var list* [synchronous sequence type]
`vector%` *elt-var vector* [synchronous sequence type]
`string%` *elt-var string* [synchronous sequence type]
`count%` *elt-var start end* [*step*] [synchronous sequence type]
`input%` *elt-var input-port reader-proc* [synchronous sequence type]
`stream%` *elt-var proc initial-seed* [synchronous sequence type]

These are all identical to their asynchronous equivalents above, except that they are synchronous. Note that `count%`'s *end* argument is required, unlike `count*`'s, because it would be nonsensical to check for termination of a sequence that does not terminate.

6.3.4 Examples

Gathering the indices of list elements that answer true to some predicate.

```
(define (select-matching-items list pred)
  (reduce ((list* elt list)
           (count* i 0))
      ((hits '()))
    (if (pred elt)
        (cons i hits)
        hits)
    (reverse hits)))
```

Finding the index of an element of a list that satisfies a predicate.

```
(define (find-matching-item list pred)
  (iterate loop ((list* elt list)
                 (count* i 0))
      ()                                ; no state variables
    (if (pred elt)
        i
        (loop))))
```

Reading one line of text from an input port.

```
(define (read-line port)
  (iterate loop ((input* c port read-char))
      ((chars '()))
    (if (char=? c #\newline)
        (list->string (reverse chars))
        (loop (cons c chars)))
    (if (null? chars)
        (eof-object)                    ; from the PRIMITIVES structure
        (list->string (reverse chars)))))
```

Counting the lines in a file. This must be written in a way other than with `count*` because it needs the value of the count after the loop has finished, but the count variable would not be bound then.

```
(define (line-count filename)
  (call-with-input-file filename
    (lambda (inport)
      (reduce ((input* line inport read-line))
          ((count 0))
        (+ count 1)))))
```

6.3.5 Defining sequence types

The sequence types are object-oriented macros similar to enumerations. An asynchronous sequence macro needs to supply three values: `#f` to indicate that it is not synchronous, a list of state variables and their initializers, and the code for one iteration. The first two methods are written in continuation-passing style: they take another macro and argument to which to pass their result. See [Friedman 00] for more details on the theory behind how CPS macros work. The `sync` method receives no extra arguments. The `state-vars`

method is passed a list of names that will be bound to the arguments of the sequence. The final method, for stepping the sequence forward, is passed the list of names bound to the arguments and the list of state variables. In addition, there is a variable to be bound to the next element of the sequence, the body expression for the loop, and an expression for terminating the loop.

As an example, the definition of `list*` is:

```
(define-syntax list*
  (syntax-rules (SYNC STATE-VARS STEP)
    ((LIST* SYNC (next more))
     (next #F more))
    ((LIST* STATE-VARS (start-list) (next more))
     (next ((list-var start-list)) more))
    ((LIST* STEP (start-list) (list-var) value-var loop-body tail-exp)
     (IF (NULL? list-var)
         tail-exp
         (LET ((value-var (CAR list-var))
               (list-var  (CDR list-var)))
           loop-body)))))
```

Synchronized sequences are similar, except that they need to provide a termination test to be used when some other synchronized method terminates the loop. To continue the example:

```
(define-syntax list%
  (syntax-rules (SYNC DONE)
    ((LIST% SYNC (next more))
     (next #T more))
    ((LIST% DONE (start-list) (list-var))
     (NULL? list-var))
    ((LIST% . anything-else)
     (LIST* . anything-else))))
```

6.3.6 Loop macro expansion

Here is an example of the expansion of the `reduce` macro:

```
(reduce ((list* x '(1 2 3)))
    ((r '()))
  (cons x r))
    ↦
(let ((final (lambda (r) (values r)))
      (list '(1 2 3))
      (r '()))
  (let loop ((list list) (r r))
    (if (null? list)
        (final r)
        (let ((x    (car list))
              (list (cdr list)))
          (let ((continue (lambda (r)
                            (loop list r))))
```

```
(continue (cons x r)))))))
```

The only mild inefficiencies in this code are the `final` & `continue` procedures, both of which could trivially be substituted in-line. The macro expander could easily perform the substitution for `continue` when there is no explicit proceed variable, as in this case, but not in general.

6.4 Library data structures

Scheme48 includes several libraries for a variety of data structures.

6.4.1 Multi-dimensional arrays

The `arrays` structure exports a facility for multi-dimensional arrays, based on Alan Bawden's interface.

`make-array` *value dimension ...* ⟶ *array* [procedure]
`array` *dimensions element ...* ⟶ *array* [procedure]
`copy-array` *array ...* ⟶ *array* [procedure]

Array constructors. `Make-array` constructs an array with the given dimensions, each of which must be an exact, non-negative integer, and fills all of the elements with *value*. `Array` creates an array with the given list of dimensions, which must be a list of exact, non-negative integers, and fills it with the given elements in row-major order. The number of elements must be equal to the product of *dimensions*. `Copy-array` constructs an array with the same dimensions and contents as *array*.

`array?` *object* ⟶ *boolean* [procedure]

Disjoint type predicate for arrays.

`array-shape` *array* ⟶ *integer-list* [procedure]

Returns the list of dimensions of *array*.

`array-ref` *array index ...* ⟶ *value* [procedure]
`array-set!` *array value index ...* ⟶ *unspecified* [procedure]

Array element dereferencing and assignment. Each *index* must be in the half-open interval $[0,d)$, where d is the respective dimension of *array* corresponding with that index.

`array->vector` *array* ⟶ *vector* [procedure]

Creates a vector of the elements in *array* in row-major order.

`make-shared-array` *array linear-map dimension ...* ⟶ *array* [procedure]

Creates a new array that shares storage with *array* and uses the procedure *linear-map* to map indices in the new array to indices in *array*. *Linear-map* must accept as many arguments as *dimension ...*, each of which must be an exact, non-negative integer; and must return a list of exact, non-negative integers equal in length to the number of dimensions of *array*, and which must be valid indices into *array*.

6.4.2 Red/black search trees

Along with hash tables for general object maps, Scheme48 also provides red/black binary search trees generalized across key equality comparison & ordering functions, as opposed to key equality comparison & hash functions with hash tables. These names are exported by the `search-trees` structure.

`make-search-tree` *key= key<* $\longrightarrow$ *search-tree* [procedure]
`search-tree?` *object* $\longrightarrow$ *boolean* [procedure]

`Make-search-tree` creates a new search tree with the given key equality comparison & ordering functions. `Search-tree?` is the disjoint type predicate for red/black binary search trees.

`search-tree-ref` *search-tree key* $\longrightarrow$ *value or* `#f` [procedure]
`search-tree-set!` *search-tree key value* $\longrightarrow$ *unspecified* [procedure]
`search-tree-modify!` *search-tree key modifier* $\longrightarrow$ *unspecified* [procedure]

`Search-tree-ref` returns the value associated with *key* in *search-tree*, or `#f` if no such association exists. `Search-tree-set!` assigns the value of an existing association in *search-tree* for *key* to be *value*, if the association already exists; or, if not, it creates a new association with the given key and value. If *value* is `#f`, however, any association is removed. `Search-tree-modify!` modifies the association in *search-tree* for *key* by applying *modifier* to the previous value of the association. If no association previously existed, one is created whose key is *key* and whose value is the result of applying *modifier* to `#f`. If *modifier* returns `#f`, the association is removed. This is equivalent to `(search-tree-set!` *`search-tree key`* `(`*`modifier`* `(search-tree-ref` *`search-tree key`*`)))`, but it is implemented more efficiently.

`search-tree-max` *search-tree* $\longrightarrow$ *value or* `#f` [procedure]
`search-tree-min` *search-tree* $\longrightarrow$ *value or* `#f` [procedure]
`pop-search-tree-max!` *search-tree* $\longrightarrow$ *value or* `#f` [procedure]
`pop-search-tree-min!` *search-tree* $\longrightarrow$ *value or* `#f` [procedure]

These all return two values: the key & value for the association in *search-tree* whose key is the maximum or minimum of the tree. `Search-tree-max` and `search-tree-min` do not remove the association from *search-tree*; `pop-search-tree-max!` and `pop-search-tree-min!` do. If *search-tree* is empty, these all return the two values `#f` and `#f`.

`walk-search-tree` *proc search-tree* $\longrightarrow$ *unspecified* [procedure]

This applies *proc* to two arguments, the key & value, for every association in *search-tree*.

6.4.3 Sparse vectors

Sparse vectors, exported by the structure `sparse-vectors`, are vectors that grow as large as necessary without leaving large, empty spaces in the vector. They are implemented as trees of subvectors.

`make-sparse-vector` $\longrightarrow$ *sparse-vector* [procedure]

Sparse vector constructor.

`sparse-vector-ref` *sparse-vector index* ⟶ *value or* `#f` [procedure]
`sparse-vector-set!` *sparse-vector index value* ⟶ *unspecified* [procedure]
Sparse vector element accessor and modifier. In the case of `sparse-vector-ref`, if *index* is beyond the highest index that was inserted into *sparse-vector*, it returns `#f`; if `sparse-vector-set!` is passed an index beyond what was already assigned, it simply extends the vector.

`sparse-vector->list` *sparse-vector* ⟶ *list* [procedure]
Creates a list of the elements in *sparse-vector*. Elements that uninitialized gaps comprise are denoted by `#f` in the list.

6.5 I/O extensions

These facilities are all exported from the `extended-ports` structure.

Tracking ports track the line & column number that they are on.

`make-tracking-input-port` *sub-port* ⟶ *input-port* [procedure]
`make-tracking-output-port` *sub-port* ⟶ *output-port* [procedure]
Tracking port constructors. These simply create wrapper ports around *sub-port* that track the line & column numbers.

`current-row` *port* ⟶ *integer or* `#f` [procedure]
`current-column` *port* ⟶ *integer or* `#f` [procedure]
Accessors for line (row) & column number information. If *port* is a not a tracking port, these simply return `#f`.

`fresh-line` *port* ⟶ *unspecified* [procedure]
This writes a newline to port with `newline`, unless it can be determined that the previous character was a newline — that is, if `(current-column` *port*`)` does not evaluate to zero.

These are ports based on procedures that produce and consume single characters at a time.

`char-source->input-port` *char-producer* [*readiness-tester closer*] [procedure]
⟶ *input-port*
`char-sink->output-port` *char-consumer* ⟶ *output-port* [procedure]
`Char-source->input-port` creates an input port that calls *char-producer* with zero arguments when a character is read from it. If *readiness-tester* is present, it is used for the `char-ready?` operation on the resulting port; likewise with *closer* and `close-input-port`. `Char-sink->output-port` creates an output port that calls *char-consumer* for every character written to it.

Scheme48 also provides ports that collect and produce output to and from strings.

`make-string-input-port` *string* ⟶ *input-port* [procedure]
Constructs an input port whose contents are read from *string*.

`make-string-output-port` ⟶ *output-port* [procedure]
`string-output-port-output` *string-port* ⟶ *string* [procedure]
`call-with-string-output-port` *receiver* ⟶ *string* [procedure]
`Make-string-output-port` makes an output port that collects its output in a string. `String-output-port-output` returns the string that *string-port* collected. `Call-with-string-output-port` creates a string output port, applies *receiver* to it, and returns the string that the string output port collected.

Finally, there is a facility for writing only a limited quantity of output to a given port.

`limit-output` *port count receiver* ⟶ *unspecified* [procedure]
`Limit-output` applies *receiver* to a port that will write at most *count* characters to *port*.

6.6 TCP & UDP sockets

Scheme48 provides a simple facility for TCP & UDP sockets. Both the structures `sockets` and `udp-sockets` export several general socket-related procedures:

`close-socket` *socket* ⟶ *unspecified* [procedure]
`socket-port-number` *socket* ⟶ *integer* [procedure]
`get-host-name` ⟶ *string* [procedure]
`Close-socket` closes *socket*, which may be any type of socket. `Socket-port-number` returns the port number through which *socket* is communicating. `Get-host-name` returns the network name of the current machine.

Note: Programmers should be wary of storing the result of a call to `get-host-name` in a dumped heap image, because the actual machine's host name may vary from invocation to invocation of the Scheme48 VM on that image, since heap images may be resumed on multiple different machines.

6.6.1 TCP sockets

The `sockets` structure provides simple TCP socket facilities.

`open-socket` [*port-number*] ⟶ *socket* [procedure]
`socket-accept` *socket* ⟶ [*input-port output-port*] [procedure]
The server interface. `Open-socket` creates a socket that listens on *port-number*, which defaults to a random number above 1024. `Socket-accept` blocks until there is a client waiting to be accepted, at which point it returns two values: an input port & an output port to send & receive data to & from the client.

`socket-client` *host-name port-number* ⟶ [*input-port output-port*] [procedure]
Connects to the server at *port-number* denoted by the machine name *host-name* and returns an input port and an output port for sending & receiving data to & from the server. `Socket-client` blocks the current thread until the server accepts the connection request.

6.6.2 UDP sockets

The `udp-sockets` structure defines a UDP socket facility.

`open-udp-socket` [*port-number*] ⟶ *socket* [procedure]
Opens a UDP socket on *port-number*, or a random port number if none was passed. `Open-udp-socket` returns two values: an input UDP socket and an output UDP socket.

`udp-send` *socket address buffer count* ⟶ *count-sent* [procedure]
`udp-receive` *socket buffer* ⟶ [*count-received remote-address*] [procedure]
`Udp-send` attempts to send *count* elements from the string or byte vector *buffer* from the output UDP socket *socket* to the UDP address *address*, and returns the number of octets it successfully sent. `Udp-receive` receives a UDP message from *socket*, reading it into *buffer* destructively. It returns two values: the number of octets read into *buffer* and the address whence the octets came.

`lookup-udp-address` *name port* ⟶ *udp-address* [procedure]
`udp-address?` *object* ⟶ *boolean* [procedure]
`udp-address-address` *address* ⟶ *c-byte-vector* [procedure]
`udp-address-port` *address* ⟶ *port-number* [procedure]
`udp-address-hostname` *address* ⟶ *string-address* [procedure]
`Lookup-udp-address` returns a UDP address for the machine name *name* at the port number *port*. `Udp-address?` is the disjoint type predicate for UDP addresses. `Udp-address-address` returns a byte vector that contains the C representation of *address*, suitable for passing to C with Scheme48's C FFI. `Udp-address-port` returns the port number of *address*. `Udp-address-hostname` returns a string representation of the IP address of *address*.

6.7 Common-Lisp-style formatting

Scheme48 provides a simple Common-Lisp-style `format` facility in the `formats` structure. It does not provide nearly as much functionality as Common Lisp, however: the considerable complexity of Common Lisp's `format` was deliberately avoided because it was deemed inconsistent with Scheme48's design goals. Scheme48's `format` is suitable for most simple purposes, anyhow.

`format` *port control-string argument ...* ⟶ *unspecified or string* [procedure]
Prints *control-string* to *port*. If, anywhere in *control-string*, the character `~` (tilde) occurs, the following character determines what to print in the place of the tilde and following character. Some formatting directives consume arguments from *argument ...*. Formatting directive characters are case-insensitive. If *port* is `#t`, the output is printed to to the value of `(current-output-port)`; if *port* is false, the output is collected in a string and returned.

The complete list of formatting directives:

`~~` Prints a single `~` (tilde), and does not consume an argument.

`~A` Consumes and prints the first remaining argument with `display`. ('A'ny)

`~D` Consumes and prints the first remaining argument as a decimal number using `number->string`. ('D'ecimal)

`~S` Consumes and prints the first remaining argument with `write`. ('S'-expression)

`~%` Prints a newline with `newline`.

`~&` Prints a newline with `newline`, unless it can be determined that a newline was immediately previously printed to *port* (see Section 6.5 [I/O extensions], page 106).

`~?` Recursively formats. The first remaining argument is consumed and must be another control string; the argument directly thereafter is also consumed, and it must be a list of arguments corresponding with that control string. The control string is formatted with those arguments using `format`.

`Format` examples:

```
(format #t "Hello, ~A!~%" "world")
    ⊣ Hello, world!
    ⊣

(format #t "Hello~?~S~%" "~A world" '(#\,) '!)
    ⊣ Hello, world!
    ⊣

(format #f "~A~A ~A." "cH" "uMBLE" "spuZz")
    ⇒ "cHuMBLE spuZz."

(let ((x 10) (y .1))
  (format #t "x: ~D~%~&y: ~D~%~&" x y))
    ⊣ x: 10
    ⊣ y: .1
```

6.8 Library utilities

Scheme48 provides various miscellaneous library utilities for common general-purpose tasks.

6.8.1 Destructuring

The `destructuring` structure exports a form for destructuring S-expressions.

`destructure` ((*pattern value*) ...) *body* [syntax]

For each (*pattern value*) pair, binds every name in *pattern* to the corresponding location in the S-expression *value*. For example,

```
(destructure (((x . y) (cons 5 3))
              ((#(a b) c) '(#((1 2) 3) (4 5))))
  body)
```

binds x to `5`, y to `3`, a to `(1 2)`, b to `3`, and c to `(4 5)`, in *body*.

6.8.2 Pretty-printing

The `pp` structure exports a simple pretty-printer.

`p` *object* [*port*] ⟶ *unspecified* [procedure]
`pretty-print` *object port position* ⟶ *unspecified* [procedure]

`P` is a convenient alias for `pretty-print`; it passes 0 for *position* and the value of `(current-output-port)` if *port* is not passed. `Pretty-print` pretty-prints *object* to *port*, using a left margin of *position*. For example:

```
(p '(define (fact n)
      (let loop ((p 1) (c 1))
        (if (> c n) p (loop (* p c) (+ c 1))))))
⊣ (define (fact n)
⊣   (let loop ((p 1) (c 1))
⊣     (if (> c n)
⊣         p
⊣         (loop (* p c) (+ c 1)))))
```

The pretty-printer is somewhat extensible as well:

`define-indentation` *name count* ⟶ *unspecified* [procedure]

Sets the number of subforms to be indented past *name* in pretty-printed output to be *count*. For example:

```
(define-indentation 'frobozz 3)
(p '(frobozz (foo bar baz quux zot) (zot quux baz bar foo)
             (mumble frotz gargle eek) (froomble zargle hrumph)))
⊣ (frobozz (foo bar baz quux zot)
⊣          (zot quux baz bar foo)
⊣          (mumble frotz gargle eek)
⊣   (froomble zargle hrumph))
```

6.8.3 Strongly connected graph components

The `strong` structure exports a routine for finding a list of the strongly connected components in a graph.

`strongly-connected-components` *vertices to slot set-slot!* ⟶ *sorted-strong-vertices* [procedure]

Returns the components of a graph containing vertices from the list *vertices* that are strongly connected, in a reversed topologically sorted list. *To* should be a procedure of one argument, a vertex, that returns a list of all vertices that have an edge to its argument. *Slot* & *set-slot!* should be procedures of one & two arguments, respectively, that access & modify arbitrary slots used by the algorithm. The slot for every vertex should initially be `#f` before calling `strongly-connected-components`, and the slots are reverted to `#f` before `strongly-connected-components` returns.

6.8.4 Nondeterminism

The `nondeterminism` structure provides a simple nondeterministic ambivalence operator, like McCarthy's `AMB`, and a couple utilities atop it, built with Scheme's `call-with-current-continuation`.

`with-nondeterminism` *thunk* ⟶ *values* [procedure]
: Initializes the nondeterminism system and calls *thunk*; this returns the values *thunk* returns after then tearing down what was set up.

`either` *option* ... ⟶ *value* [syntax]
`one-value` *exp* ⟶ *value* [syntax]
`all-values` *exp* ⟶ *list* [syntax]
: `Either` evaluates to the value of any one of the options. It is equivalent to McCarthy's `AMB`. It may return any number of times. `One-value` returns the only value that *exp* could produce; it will return only once, although it may actually return any number of values (if *exp* contains a call to `values`). `All-values` returns a list of all of the single values, not multiple values, that *exp* could nondeterministically evaluate to.

`fail` ⟶ *does not return* [procedure]
: Signals a nondeterministic failure. This is invalid outside of a `with-nondeterminism`-protected dynamic extent.

6.8.5 Miscellaneous utilities

The `big-util` structure exports a variety of miscellaneous utilities.

`concatenate-symbol` *elt* ... ⟶ *symbol* [procedure]
: Returns a symbol containing the contents of the sequence *elt* .... Each *elt* may be another symbol, a string, or a number. Numbers are converted to strings in base ten.

`error` *format-string argument* ... ⟶ *values* (*may not return*) [procedure]
`breakpoint` *format-string argument* ... ⟶ *values* (*may not return*) [procedure]
: `Error` signals an error whose message is formatted by `format` (see Section 6.7 [Common-Lisp-style formatting], page 108) with the given formatting template string and arguments. `Breakpoint` signals a breakpoint with a message similarly constructed and causes the command processor to push a new command level (see Section 2.4.5 [Command levels], page 13).

`atom?` *x* ⟶ *boolean* [procedure]
: Returns true if *x* is not a pair or false if it is.

`neq?` *x y* ⟶ *boolean* [procedure]
`n=` *x y* ⟶ *boolean* [procedure]
: Negations of the `eq?` and `=` predicates.

`identity` *value* ⟶ *value* [procedure]
`no-op` *value* ⟶ *value* [procedure]
: These simply return their arguments. The difference between them is that `no-op` is guaranteed not to be integrated by the compiler, whereas `identity` may be.

`null-list?` *object* ⟶ *boolean* [procedure]
: Returns `#t` if *object* is the null list, returns `#f` if *object* is a pair, or signals an error if *object* is neither the null list nor a pair.

`reverse!` *list* ⟶ *reversed-list* [procedure]
: Returns a list containing the reverse elements of *list*. Note that the original *list* is *not* reversed; it becomes garbage. `Reverse!` simply re-uses its structure.

`memq?` *object list* ⟶ *boolean* [procedure]
Returns `#t` if *object* is a member of *list*, as determined by `eq?`; or `#f` if not.

`first` *predicate list* ⟶ *elt or* `#f` [procedure]
`any` *predicate list* ⟶ *elt or* `#f` [procedure]
`First` returns the first element of *list* that satisfies *predicate*, or `#f` if no element does. `Any` returns an element of *list* that satisfies *predicate*. Note that `any` may choose any element of the list, whereas `first` explicitly returns the *first* element that satisfies *predicate*.

`any?` *predicate list* ⟶ *boolean* [procedure]
`every?` *predicate list* ⟶ *boolean* [procedure]
`Any?` returns `#t` if any element of *list* satisfies *predicate*, or `#f` if none do. `Every?` returns `#t` if every element of *list* satisfies *predicate*, or `#f` if there exists an element that does not.

`filter` *predicate list* ⟶ *filtered-list* [procedure]
`filter!` *predicate list* ⟶ *filtered-list* [procedure]
These return a list of all elements in *list* that satisfy *predicate*. `Filter` is not allowed to modify *list*'s structure; `filter!` may, however.

`filter-map` *proc list* ⟶ *list* [procedure]
This is a combination of `filter` and `map`. For each element *e* in *list*: if `(`*proc* `e)` returns a true value, that true value is collected in the output list. `Filter-map` does not modify *list*'s structure.

`remove-duplicates` *list* ⟶ *uniquified-list* [procedure]
Returns a unique list of all elements in *list*; that is, if there were any duplicates of any element *e* in *list*, only a single *e* will occur in the returned list. `Remove-duplicates` does not modify *list*'s structure.

`partition-list` *predicate list* ⟶ [*satisfied unsatisfied*] [procedure]
`partition-list!` *predicate list* ⟶ [*satisfied unsatisfied*] [procedure]
These return two values: a list of all elements in *list* that do satisfy *predicate* and a list of all elements that do not. `Partition-list` is not allowed to modify *list*'s structure; `partition-list!` is.

`delq` *object list* ⟶ *list* [procedure]
`delq!` *object list* ⟶ *list* [procedure]
These return a list containing all elements of *list* except for *object*. `Delq` is not allowed to modify *list*'s structure; `delq!` is.

`delete` *predicate list* ⟶ *list* [procedure]
Returns a list of all elements in *list* that do not satisfy *predicate*. Note that, despite the lack of exclamation mark in the name, this *may* modify *list*'s structure.

`string->immutable-string` *string* ⟶ *immutable-string* [procedure]
Returns an immutable string with *string*'s contents. If *string* is already immutable, it is returned; otherwise, an immutable copy is returned.

6.8.6 Multiple value binding

The `receiving` structure exports the `receive` macro, a convenient syntax atop R5RS's `call-with-values`.

`receive` *formals producer body* [syntax]

Binds the variables in the lambda parameter list *formals* to the return values of *producer* in *body*.

```
(receive formals
         producer
  body)
    ≡
(call-with-values
    (lambda () producer)
  (lambda formals body))
```

For sequences of multiple value bindings, the `mvlet` structure exports two convenient macros.

`mvlet*` [syntax]
`mvlet` [syntax]

`Mvlet*` is a multiple-value version of `let` or a linearly nested version of `receive`:

```
(mvlet* ((formals0 producer0)
         (formals1 producer1)
         ...)
  body)
    ≡
(call-with-values
    (lambda () producer0)
  (lambda formals0
    (call-with-values
        (lambda () producer1)
      (lambda formals1
        ...body...))))
```

`Mvlet` is similar, but each *producer* is evaluated in an environment where none of the variables in any of the *formals* is bound, and the order in which each producer expression is evaluated is unspecified.

6.8.7 Object dumper

Scheme48 has a rudimentary object dumper and retriever in the structure `dump/restore`. It is not a 'real' object dumper in the sense that it will not handle cycles in object graphs correctly; it simply performs a recursive descent and will diverge if it reaches a cycle or stop after a recursive depth parameter.

The types of objects that the dumper supports are: several miscellaneous constants (`()`, `#t`, `#f`, & the unspecific token), pairs, vectors, symbols, numbers, strings, characters, and byte vectors.

`dump` *object char-writer depth* $\longrightarrow$ *unspecified* [procedure]
Dumps *object* by repeatedly calling *char-writer*, which must be a procedure that accepts exactly one character argument, on the characters of the serialized representation. If the dumper descends into the object graph whose root is *object* for more than *depth* recursions, an ellipsis token is dumped in the place of the vertex at *depth*.

`restore` *char-reader* $\longrightarrow$ *object* [procedure]
Restores the object whose serialized components are retrieved by repeatedly calling *char-reader*, which must be a procedure that accepts zero arguments and returns a character.

6.8.8 Simple time access

The `time` structure exports a simple facility for accessing time offsets in two different flavours.

`real-time` $\longrightarrow$ *milliseconds* [procedure]
Returns the real time in milliseconds that has passed since some unspecified moment in time.[1] Though not suitable for measurements relative to entities outside the Scheme48 image, the real time is useful for measuring time differences within the Scheme image with reasonable precision; for example, thread sleep timing is implemented with this real time primitive.

`run-time` $\longrightarrow$ *ticks* [procedure]
Returns the run time as an integer representing processor clock ticks since the start of the Scheme48 process. This is much less precise than the real time, but it is useful for measuring time actually spent in the Scheme48 process, as opposed to time in general.

[1] In the current implementation on Unix, this moment happens to be the first call to `real-time`; on Win32, this is the start of the Scheme process.

7 C interface

This chapter describes an interface for calling C functions from Scheme, calling Scheme procedures from C, and working with the Scheme heap in C. Scheme48 manages stub functions in C that negotiate between the calling conventions of Scheme & C and the memory allocation policies of both worlds. No stub generator is available yet, but writing stubs is a straightforward task.

7.1 Overview of the C interface

The following facilities are available for interfacing between Scheme48 & C:

- Scheme code can call C functions.
- The external interface provides full introspection for all Scheme objects. External code may inspect, modify, and allocate Scheme objects arbitrarily.
- External code may raise exceptions back to Scheme48 to signal errors.
- External code may call back into Scheme. Scheme48 correctly unrolls the process stack on non-local exits.
- External modules may register bindings of names to values with a central registry accessible from Scheme. Conversely, Scheme code can register shared bindings for access by C code.

7.1.1 Scheme structures

On the Scheme side of the C interface, there are three pertinent structures: `shared-bindings` (see Section 7.2 [Shared bindings between Scheme and C], page 116), which provides the Scheme side of the facility for sharing data between Scheme and C; `external-calls` (see Section 7.3 [Calling C functions from Scheme], page 118), which exports several ways to call C functions from Scheme, along with some useful facilities, such as object finalizers, which are also available from elsewhere; and `load-dynamic-externals` (see Section 7.4 [Dynamic loading of C modules], page 119), which provides a dynamic external object loading facility. Also, the old dynamic loading facility is still available from the `dynamic-externals` structure, but its use is deprecated, and it will most likely vanish in a later release.

7.1.2 C naming conventions

Scheme48's C bindings all have strict naming conventions. Variables & procedures have `s48_` prefixed to them; macros, `S48_`. Whenever a C name is derived from a Scheme identifier, hyphens are replaced with underscores. Also, procedures or variables are converted to lowercase, while macros are converted to uppercase. The `?` suffix, generally appended to predicates, is converted to `_p` (or `_P` in macro names). Trailing `!` is dropped. For example, the C macro that corresponds with Scheme's `pair?` predicate is named `S48_PAIR_P`, and the C macro to assign the car of a pair is named `S48_SET_CAR`. Procedures and macros that do not verify the types of their arguments have 'unsafe' in their names.

All of the C functions and macros described have prototypes or definitions in the file `c/scheme48.h` of Scheme48's standard distribution. The C type for Scheme values is defined there to be `s48_value`.

7.1.3 Garbage collection

Scheme48 uses a copying garbage collector. The collector must be able to locate all references to objects allocated in the Scheme48 heap in order to ensure that storage is not reclaimed prematurely and to update references to objects moved by the collector. The garbage collector may run whenever an object is allocated in the heap. C variables whose values are Scheme48 objects and which are live across heap allocation calls need to be registered with the garbage collector. For more information, see Section 7.7 [Interacting with the Scheme heap in C], page 125.

7.2 Shared bindings between Scheme and C

Shared bindings are the means by which named values are shared between Scheme & C code. There are two separate tables of shared bindings, one for values defined in Scheme and accessed from C and the other for the opposite direction. Shared bindings actually bind names to cells, to allow a name to be resolved before it has been assigned. This is necessary because C initialization code may run before or after the corresponding Scheme code, depending on whether the Scheme code is in the resumed image or run in the current session. The Scheme bindings described here are available from the `shared-bindings` structure.

7.2.1 Scheme shared binding interface

`shared-binding?` *object* ⟶ *boolean* [Scheme procedure]
`shared-binding-is-import?` *shared-binding* ⟶ *boolean* [Scheme procedure]
`Shared-binding?` is the disjoint type predicate for all shared bindings, imported or exported; `shared-binding-is-import?` returns true if *shared-binding* was imported into Scheme from C, and false if it has the converse direction.

`shared-binding-ref` *shared-binding* ⟶ *value* [Scheme procedure]
`shared-binding-set!` *shared-binding value* ⟶ *unspecified* [Scheme procedure]
`Shared-binding-ref` returns the value of *shared-binding*; `shared-binding-set!` sets the value of *shared-binding* to be *value*.

`lookup-imported-binding` *name* ⟶ *shared-binding* [Scheme procedure]
`define-imported-binding` *name value* ⟶ *unspecified* [Scheme procedure]
`undefine-imported-binding` *name* ⟶ *unspecified* [Scheme procedure]
`Lookup-imported-binding` returns the binding imported from C to Scheme with the given name; a binding is created if none exists. `Define-imported-binding` creates a new such binding, anomalously from within Scheme; such bindings are usually created instead from within C using the C `s48_define_exported_binding` function. `Undefine-imported-binding` removes the shared binding whose name is *name* from the table of imported bindings.

`lookup-exported-binding` *name* ⟶ *shared-binding* [Scheme procedure]
`define-exported-binding` *name value* ⟶ *unspecified* [Scheme procedure]

`undefine-exported-binding` *name* ⟶ *unspecified* [Scheme procedure]
Equivalents of the above three procedures, but for bindings exported from Scheme to C. `Define-imported-binding`, unlike `define-exported-binding`, is customary to use in Scheme, as its intended use is to make a Scheme value available to C code from within Scheme.

`find-undefined-imported-bindings` ⟶ *vector* [Scheme procedure]
Returns a vector of all bindings imported into Scheme from C with undefined values, *i.e.* those created implicitly by lookups that have not yet been assigned rather than those created explicitly by the shared binding definers (`define-exported-binding`, *&c.*).

7.2.2 C shared binding interface

`s48_value S48_SHARED_BINDING_P (`*s48_value* `obj`) [C macro]
`s48_value S48_SHARED_BINDING_NAME (`*s48_value* `shared_binding`) [C macro]
`s48_value S48_SHARED_BINDING_IS_IMPORTP (`*s48_value* `shared-binding`) [C macro]
`s48_value S48_SHARED_BINDING_REF (`*s48_value* `shared_binding`) [C macro]
`void S48_SHARED_BINDING_SET (`*s48_value* `shared_binding`, *s48_value* `value`) [C macro]
These macros are C counterparts to Scheme's `shared-binding?`, `shared-binding-name`, `shared-binding-is-import?`, `shared-binding-ref`, and `shared-binding-set!`, respectively.

statement `S48_SHARED_BINDING_CHECK (`*s48_value* `binding`) [C macro]
Signals an exception if and only if *binding*'s value is Scheme48's 'unspecific' value.

Huh?: Undefined shared bindings are not initialized with the 'unspecific' value, but rather with an entirely different special token referred to internally as 'undefined,' used in circumstances such as this — yet `S48_SHARED_BINDING_CHECK`, as defined in `scheme48.h`, definitely checks whether *binding*'s value is the 'unspecific' value.

`s48_value s48_get_imported_binding (`*char* `*name`) [C function]
Returns the shared binding defined in Scheme for *name*, creating it if necessary.

`void s48_define_exported_binding (`*char* `*name`, *s48_value* `value`) [C function]
Defines a shared binding named *name* with the value *value* that can be accessed from Scheme.

`void S48_EXPORT_FUNCTION (`*fn*`)` [C macro]
This is a convenience for the common case of exporting a C function to Scheme. This expands into

```
s48_define_exported_binding("fn",
                            s48_enter_pointer(fn))
```

which boxes the function into a Scheme48 byte vector and then exports it as a shared binding. Note that `s48_enter_pointer` allocates space in the Scheme heap and may trigger a garbage collection; see Section 7.7 [Interacting with the Scheme heap in C], page 125.

7.3 Calling C functions from Scheme

The `external-calls` structure exports several ways to call C functions from Scheme, along with several other related utilities, many of which are also available from other structures. There are two different ways to call C functions from Scheme, depending on how the C function was obtained:

`call-imported-binding` *binding argument ...* $\longrightarrow$ *value* [Scheme procedure]
`call-external-value` *byte-vector name argument ...* $\longrightarrow$ *value* [Scheme procedure]

Each of these applies its first argument, a C function, to the rest of the arguments. For `call-imported-binding`, the function argument must be an imported binding. For `call-external-value`, the function argument must be a byte vector that contains a pointer to a C function, and *name* should be a string that names the function. The *name* argument is used only for printing error messages.

For both of these, the C function is passed the argument values, and the value returned is that returned by the C function. No automatic representation conversion occurs for either arguments or return values. Up to twelve arguments may be passed. There is no method supplied for returning multiple values to Scheme from C or vice versa (mainly because C does not have multiple return values).

Keyboard interrupts that occur during a call to a C function are ignored until the function returns to Scheme.[1]

`import-definition` *name* [*c-string*] [Scheme syntax]
`import-lambda-definition` *name formals* [*c-string*] [Scheme syntax]

These macros simplify importing bindings from C into Scheme and wrapping such bindings in Scheme procedures. `Import-definition` defines *name* to be the shared binding named by *c-string*, whose value, if it is not supplied, is by default a string of *name*, downcased and with all hyphens translated to underscores.

```
(define name (lookup-imported-binding c-string))
```

For example,

```
(import-definition my-foo)
    ↦ (define my-foo (lookup-imported-binding "my_foo"))
```

`Import-lambda-definition` imports the named C binding, using either the provided C binding name or by translating the Scheme name as with `import-definition`, and defines *name* to be a procedure with the given formal parameter list that calls the imported C binding with its arguments:

```
(define binding (lookup-imported-binding c-string))
(define (name formal ...)
  (call-imported-binding binding formal ...))
```

Examples:

```
(import-lambda-definition integer->process-id (int)
                          "posix_getpid")
    ↦
```

[1] This is clearly a problem; we are working on a solution.

```
(define binding0
        (lookup-imported-binding "posix_getpid"))
(define (integer->process-id int)
  (call-imported-binding binding0 int))

(import-lambda-definition s48-system (string))
    ↦
(define binding1
        (lookup-imported-binding "s48_system"))
(define (s48-system string)
  (call-imported-binding binding1 string))
```

where $binding_0$ and $binding_1$ are fresh, unused variable names.

Warning: `Import-lambda-definition`, as presently implemented, requires a fixed parameter list; it does not allow 'rest list' arguments.

`lookup-imported-binding` *name* ⟶ *shared-binding* [Scheme procedure]
`define-exported-binding` *shared-binding* ⟶ *unspecified* [Scheme procedure]
`shared-binding-ref` *shared-binding* ⟶ *value* [Scheme procedure]

These are identical to the procedures accessible with the same names from the `shared-bindings` structure (see Section 7.2 [Shared bindings between Scheme and C], page 116).

`add-finalizer!` *object procedure* ⟶ *unspecified* [Scheme procedure]

Registers *procedure* as the finalizer for *object*. When *object* is later about to be reclaimed by the garbage collector, *procedure* is applied to one argument, *object*. All finalizers are applied in a child of the root scheduler thread that is spawned after every garbage collection. If an error occurs in any finalizer, it will be printed to the standard error output port, and all other finalizers will be aborted before they are given a chance to run. Because of this, and the fact that finalizers are collected and run after every garbage collection, they should perform as little computation as possible. *Procedure* may also create new references to *object* elsewhere in the heap, in which case the object will not be reclaimed, but its associated finalizer will be forgotten.

Warning: Finalizers are expensive. Use sparingly.

`define-record-resumer` *record-type resumer* ⟶ *unspecified* [Scheme procedure]

Identical to the procedure accessible with the same name from the `record-types` structure (see Section 4.7 [Records], page 71). Record resumers are often useful in working with foreign C data, which is in many cases specific to the program image within the operating system, and which cannot straightforwardly be relocated to a different address space.

7.4 Dynamic loading of C modules

External code can be loaded into a running Scheme48 on most Unices and on Windows. Such external code must be stored in shared objects; see below on details of the C side. The relevant Scheme procedures are available in the `load-dynamic-externals` structure:

`load-dynamic-externals` *filename add-file-type? reload-on-repeat? reload-on-resume?* ⟶ *dynamic-externals* [procedure]
`import-dynamic-externals` *filename* ⟶*dynamic-externals* [procedure]
`unload-dynamic-externals` *dynamic-externals* ⟶ *unspecified* [procedure]

`Load-dynamic-external` loads a shared object from *filename*, with an appropriate file type appended if *add-file-type?* is true (`.so` on Unix and `.dll` on Windows), and returns a *dynamic externals* object representing the loaded shared object. If the shared object was already loaded, then if *reload-on-repeat?* is true, it is reloaded; otherwise, the `load-dynamic-externals` call has no effect. If the dynamic externals descriptor is stored in a dumped heap image, when that heap image is resumed, if `reload-on-resume?` is true, the shared object corresponding with that dynamic external descriptor is reloaded. `Unload-dynamic-externals` unloads the given dynamic externals object.

`Import-dynamic-externals` is a convenient wrapper for the common case of `load-dynamic-externals`; it is equivalent to `(load-dynamic-externals #t #f #t)`, *i.e.* it will append a file type, it will not reload the shared object if it was already loaded, and the shared object will be loaded if part of a resumed heap image.

`reload-dynamic-externals` *filename* ⟶ *unspecified* [procedure]

Reloads the shared object named by *filename*. This is intended as an interactive utility, which is why it accepts the filename of the shared object and not a dynamic externals descriptor.

Shared objects intended to be loaded into Scheme48 must define two functions:

`void s48_on_load` (*void*) [C function]
`void s48_on_reload` (*void*) [C function]

`s48_on_load` is called when the shared object is initially loaded by Scheme48. It typically consists of a number of invocations of `S48_EXPORT_FUNCTION` to make C functions available to Scheme48 code. `s48_on_reload` is called when the shared object is reloaded after it has been initially loaded once; it typically just calls `s48_on_load`, but it may perform other reinitializations.

On Linux, the following commands compile the C source file `foo.c` into a shared object `foo.so` that can be loaded dynamically by Scheme48:

```
% gcc -c -o foo.o foo.c
% ld -shared -o foo.so foo.o
```

7.4.1 Old dynamic loading interface

The old `dynamic-externals` structures, which exported `dynamic-load`, `get-external`, `lookup-external`, `lookup-all-externals`, `external?`, `external-name`, `external-value`, and `call-external`, is still supported, but it will not work on Windows, its use is deprecated, and it is likely to vanish in a future release. The old documentation is preserved to aid updating of old code:

On architectures that support it, external code can be loaded into a running Scheme48 process, and C object file bindings can be accessed at runtime & their values called. These Scheme procedures are exported by the structure `dynamic-externals`.

In some Unices, retrieving a value from the current process may require a non-trivial amount of computation. We recommend that a dynamically loaded file contain a single initialization function that creates shared bindings for the values exported by the file.

`dynamic-load` *string* ⟶ *unspecified* [Scheme procedure]
Loads the filename named by *string* into the current process. An exception is raised if the file cannot be found or if dynamic loading is not supported by the host operating system. The file must have been compiled & linked appropriately. For Linux, for example, the following commands compile `foo.c` into a file `foo.so` that can be loaded dynamically:

```
% gcc -c -o foo.o foo.c
% ld -shared -o foo.so foo.o
```

`get-external` *string* ⟶ *external* [Scheme procedure]
`external?` *object* ⟶ *boolean* [Scheme procedure]
`external-name` *external* ⟶ *string* [Scheme procedure]
`external-value` *external* ⟶ *byte-vector* [Scheme procedure]
These procedures access external values bound in the current process. `Get-external` returns a *external* object that contains the value of the C binding with the name *string*. It signals a warning if there is no such binding in the current process. `External?` is the disjoint type predicate for externals, and `external-name` & `external-value` return the name & value of an external. The value is represented as a byte vector (see Section 4.3 [Bitwise manipulation], page 55) of length four on 32-bit architectures. The value is that of the C binding from when `get-external` (or `lookup-external`, as described below) was called.

`lookup-external` *external* ⟶ *boolean* [Scheme procedure]
`lookup-all-externals` ⟶ *boolean* [Scheme procedure]
`Lookup-external` updates the value of *external* by looking up its binding in the current process. It returns `#t` if the external is bound and `#f` if not. `Lookup-all-externals` calls `lookup-external` on all externals in the current Scheme48 image. It returns `#t` if all were bound and `#f` if there was at least one unbound external.

`call-external` *external argument* ... ⟶ *value* [Scheme procedure]
Calls the C function pointed to by *external* with the given arguments, and returns the value that the C function returned. This is like `call-imported-binding` and `call-external-value` except that the function argument is represented as an external, not as an imported binding or byte vector containing a pointer. For more details, see Section 7.3 [Calling C functions from Scheme], page 118.

7.5 Accessing Scheme data from C

The C header file `scheme48.h` provides access to Scheme48 data structures. The type `s48_value` is used for Scheme values. When the type of a value is known, such as the integer returned by the Scheme procedure `vector-length` or the boolean returned by `pair`, the corresponding C function returns a C value of the appropriate type, not an `s48_value`. Predicates return `1` for true and `0` for false.

`s48_value S48_FALSE` [C macro]
`s48_value S48_TRUE` [C macro]
`s48_value S48_NULL` [C macro]
`s48_value S48_UNSPECIFIC` [C macro]
`s48_value S48_EOF` [C macro]
`long S48_MAX_FIXNUM_VALUE` [C macro]
`long S48_MIN_FIXNUM_VALUE` [C macro]

These C macros denote various Scheme constants. `S48_FALSE` is the boolean false value, written in Scheme as `#f`. `S48_TRUE` is the boolean true value, or `#t`. `S48_NULL` is the empty list `()`. `S48_UNSPECIFIC` is a miscellaneous value returned by procedures that have no meaningful return value (accessed in Scheme48 by the nullary procedure `unspecific` in the `util` structure). `S48_EOF` is the end-of-file object (which the Scheme procedure `eof-object?` answers true for). `S48_MAX_FIXNUM_VALUE` is the maximum integer as a `long` that can be represented in a Scheme48 fixnum. `S48_MIN_FIXNUM_VALUE` is similar, but the minimum integer.

`int S48_EXTRACT_BOOLEAN` (*s48_value* `boolean`) [C macro]
`unsigned char s48_extract_char` (*s48_value* `char`) [C function]
`char * s48_extract_string` (*s48_value* `string`) [C function]
`char * s48_extract_byte_vector` (*s48_value* `bytev`) [C function]
`long s48_extract_integer` (*s48_value* `integer`) [C function]
`double s48_extract_double` (*s48_value* `double`) [C function]
`s48_value S48_ENTER_BOOLEAN` (*int* `boolean`) [C macro]
`s48_value s48_enter_char` (*unsigned char* `char`) [C function]
`s48_value s48_enter_string` (*char* *`string`) [C function]
`s48_value s48_enter_byte_vector` (*char* *`bytev`, *long* `length`) [C function]
`s48_value s48_enter_integer` (*long* `integer`) [C function]
`s48_value s48_enter_double` (*double* `double`) [C function]

These functions & macros convert values between their respective Scheme & C representations.

`S48_EXTRACT_BOOLEAN` returns `0` if *boolean* is `#f` and `1` otherwise. `S48_ENTER_BOOLEAN` returns the Scheme value `#f` if its argument is zero and `#t` otherwise.

`s48_extract_char` & `s48_enter_char` convert between Scheme characters and C `char`s.

`s48_extract_string` & `s48_extract_byte_vector` return pointers to the actual storage used by *string* or *bytev*. These pointers are valid only until the next garbage collection, however; see Section 7.7 [Interacting with the Scheme heap in C], page 125. `s48_enter_string` & `s48_enter_byte_vector` allocate space on the Scheme48 heap for the given strings or byte vectors. `s48_enter_string` copies the data starting from the pointer it is given up to the first ASCII `NUL` character, whereas `s48_enter_byte_vector` is given the number of bytes to copy into the Scheme heap.

`s48_extract_integer` returns a C `long` that represents the Scheme integer as input. If the Scheme integer is too large to be represented in a long, an exception is signalled. (The Scheme integer may be a fixnum or a bignum.) `s48_enter_integer` converts back to Scheme integers, and it will never signal an exception.

`s48_extract_double` & `s48_enter_double` convert between Scheme & C double-precision floating point representations.

Of these, `s48_enter_string`, `s48_enter_byte_vector`, `s48_enter_integer`, & `s48_enter_double` may cause the garbage collector to be invoked: the former two copy the string or byte vector onto the Scheme heap first, `s48_enter_integer` may need to allocate a bignum (since C `long`s are wider than Scheme48 fixnums), and floats are heap-allocated in Scheme48.

`int S48_TRUE_P (`*s48_value* *object*`)` [C macro]
`int S48_FALSE_P (`*s48_value* *object*`)` [C macro]

`S48_TRUE_P` returns true if *object* is the true constant `S48_TRUE` and false if otherwise. `S48_FALSE_P` returns true if its argument is the false constant `S48_FALSE` and false if otherwise.

`int S48_FIXNUM_P (`*s48_value* *object*`)` [C macro]
`long s48_extract_fixnum (`*s48_value* *fixnum*`)` [C function]
`s48_value s48_enter_fixnum (`*long* *integer*`)` [C function]

`S48_FIXNUM_P` is the C predicate for Scheme48 fixnums, delimited in range by `S48_MIN_FIXNUM_VALUE` & `S48_MAX_FIXNUM_VALUE`. `s48_extract_fixnum` returns the C `long` representation of the Scheme fixnum, and `s48_enter_fixnum` returns the Scheme fixnum representation of the C `long`. These are identical to `s48_extract_integer` & `s48_enter_integer`, except that `s48_extract_fixnum` will never raise a range exception, but `s48_enter_fixnum` may, and `s48_enter_fixnum` will never return a bignum; this is due to the fact that C `long`s have a wider range than Scheme48 fixnums.

`int S48_EQ_P (`*s48_value* `a`, *s48_value* `b``)` [C macro]
`int S48_CHAR_P (`*s48_value* *object*`)` [C macro]
`int S48_PAIR_P (`*s48_value* *object*`)` [C macro]
`int S48_VECTOR_P (`*s48_value* *object*`)` [C macro]
`int S48_STRING_P (`*s48_value* *object*`)` [C macro]
`int S48_SYMBOL_P (`*s48_value* *object*`)` [C macro]
`int S48_BYTE_VECTOR_P (`*s48_value* *object*`)` [C macro]
`s48_value S48_CAR (`*s48_value* *pair*`)` [C macro]
`s48_value S48_CDR (`*s48_value* *pair*`)` [C macro]
`void S48_SET_CAR (`*s48_value* *pair*, *s48_value* *object*`)` [C macro]
`void S48_SET_CDR (`*s48_value* *pair*, *s48_value* *object*`)` [C macro]
`s48_value s48_cons (`*s48_value* *car*, *s48_value* *cdr*`)` [C function (may GC)]
`s48_value s48_length (`*s48_value* *list*`)` [C function]
`long S48_VECTOR_LENGTH (`*s48_value* *vector*`)` [C macro]
`s48_value S48_VECTOR_REF (`*s48_value* *vector*, *long* *index*`)` [C macro]
`void S48_VECTOR_SET (`*s48_value* *vector*, *long* *index*, *s48_value* *object*`)` [C macro]
`s48_value s48_make_vector (`*long* *length*, *s48_value* *fill*`)` [C function (may GC)]
`long S48_STRING_LENGTH (`*s48_value* *string*`)` [C macro]
`char S48_STRING_REF (`*s48_value* *string*, *long* *index*`)` [C macro]
`void S48_STRING_SET (`*s48_value* *string*, *long* *index*, *char* *char*`)` [C macro]

`s48_value s48_make_string` (*long* `length`, *char* `fill`) [C function (may GC)]
`s48_value S48_SYMBOL_TO_STRING` (*s48_value* `symbol`) [C macro]
`long S48_BYTE_VECTOR_LENGTH` (*s48_value* `bytev`) [C macro]
`char S48_BYTE_VECTOR_REF` (*s48_value* `bytev`, *long* `index`) [C macro]
`void S48_BYTE_VECTOR_SET` (*s48_value* `bytev`, *long* `index`, *char* `byte`) [C macro]
`s48_value s48_make_byte_vector` (*long* `length`) [C function (may GC)]

C versions of miscellaneous Scheme procedures. The names were derived from their Scheme counterparts by replacing hyphens with underscores, `?` suffixes with `_P`, and dropping `!` suffixes.

7.6 Calling Scheme procedures from C

`s48_value s48_call_scheme` (*s48_value* `proc`, *long* `nargs`, ...) [C function]

Calls the Scheme procedure *proc* on *nargs* arguments, which are passed as additional arguments to `s48_call_scheme`. There may be at most twelve arguments. The value returned by the Scheme procedure is returned to the C procedure. Calling any Scheme procedure may potentially cause a garbage collection.

There are some complications that arise when mixing calls from C to Scheme with continuations & threads. C supports only downward continuations (via `longjmp()`). Scheme continuations that capture a portion of the C stack have to follow the same restriction. For example, suppose Scheme procedure `s0` captures continuation `a` and then calls C function `c0`, which in turn calls Scheme procedure `s1`. `S1` can safely call the continuation `a`, because that is a downward use. When `a` is called, Scheme48 will remove the portion of the C stack used by the call to `c0`. On the other hand, if `s1` captures a continuation, that continuation cannot be used from `s0`, because, by the time control returns to `s0`, the C stack used by `s0` will no longer be valid. An attempt to invoke an upward continuation that is closed over a portion of the C stack will raise an exception.

In Scheme48, threads are implemented using continuations, so the downward restriction applies to them as well. An attempt to return from Scheme to C at a time when the appropriate C frame is not on the top of the C stack will cause the current thread to block until the frame is available. For example, suppose thread `t0` calls a C function that calls back to Scheme, at which point control switches to thread `t1`, which also calls C & then back to Scheme. At this point, both `t0` & `t1` have active calls to C on the C stack, with `t1`'s C frame above `t0`'s. If `t0` attempts to return from Scheme to C, it will block, because the frame is not yet accessible. Once `t1` has returned to C and from there back to Scheme, `t0` will be able to resume. The return to Scheme is required because context switches can occur only while Scheme code is running. `T0` will also be able to resume if `t1` uses a continuation to throw past its call out to C.

7.7 Interacting with the Scheme heap in C

Scheme48 uses a precise copying garbage collector. Any code that allocates objects within the Scheme48 heap may trigger a garbage collection. Variables bound to values in the Scheme48 heap need to be registered with the garbage collector so that the value will be safely held and so that the variables will be updated if the garbage collector moves the object. The garbage collector has no facility for updating pointers to the interiors of objects, so such pointers, for example the ones returned by `S48_EXTRACT_STRING`, will likely become invalid when a garbage collection occurs.

`S48_DECLARE_GC_PROTECT` (*n*) [C macro]
`void S48_GC_PROTECT_`*n* (*s48_value* `var`$_1$, ..., *s48_value* `var`$_n$) [C macro]
`void S48_GC_UNPROTECT` () [C macro]

`S48_DECLARE_GC_PROTECT`, where 1 <= *n* <= 9, allocates storage for registering *n* variables. At most one use of `S48_DECLARE_GC_PROTECT` may occur in a block. After declaring a GC protection, `S48_GC_PROTECT_`*n* registers the *n* variables with the garbage collector. It must be within the scope that the `S48_DECLARE_GC_PROTECT` occurred in and before any code that can cause a garbage collection. `S48_GC_UNPROTECT` removes the current block's protected variables from the garbage collector's list. It must be called at the end of the block after any code that may cause a garbage collection. Omitting any of the three may cause serious and hard-to-debug problems, because the garbage collector may relocate an object and invalidate unprotected `s48_value` pointers. If not `S48_DECLARE_GC_PROTECT` is matched with a `S48_GC_UNPROTECT` or vice versa, a `gc-protection-mismatch` exception is raised when a C procedure returns to Scheme.

`void * S48_GC_PROTECT_GLOBAL` (*global*) [C macro]
`void S48_GC_UNPROTECT_GLOBAL` (*void *handle*) [C macro]

`S48_GC_PROTECT_GLOBAL` permanently registers the l-value *global* with the system as a garbage collection root. It returns a pointer which may then be supplied to `S48_GC_UNPROTECT_GLOBAL` to unregister the l-value as a root.

7.7.1 Keeping C data structures in the Scheme heap

C data structures can be stored within the Scheme heap by embedding them inside byte vectors (see Section 4.3 [Bitwise manipulation], page 55). The following macros can be used to create and access embedded C objects.

`s48_value S48_MAKE_VALUE` (*type*) [C macro]
type `S48_EXTRACT_VALUE` (*s48_value bytev*, *type*) [C macro]
type `* S48_EXTRACT_VALUE_POINTER` (*s48_value bytev*, *type*) [C macro]
`void S48_SET_VALUE` (*s48_value bytev*, *type*, *type value*) [C macro]

`S48_MAKE_VALUE` allocates a byte vector large enough to hold a C value whose type is *type*. `S48_EXTRACT_VALUE` returns the contents of the byte vector *bytev* cast to *type*, and `S48_EXTRACT_VALUE_POINTER` returns a pointer to the contents of the byte vector, which is valid only until the next garbage collection. `S48_SET_VALUE` stores a value into the byte vector.

7.7.2 C code and heap images

Scheme48 uses dumped heap images to restore a previous system state. The Scheme48 heap is written into a file in a machine-independent and operating-system-independent format. The procedures described above, however, may be used to create objects in the Scheme heap that contain information specific to the current machine, operating system, or process. A heap image containing such objects may not work correctly when resumed.

To address this problem, a record type may be given a *resumer* procedure. On startup, the resumer procedure for a record type is applied to each record of that type in the image being restarted. This procedure can update the record in a manner appropriate to the machine, operating system, or process used to resume the image. Note, though, that there is no reliable order in which record resumer procedures are applied. To specify the resumer for a record type, use the `define-record-resumer` procedure from the `record-types` structure (see Section 4.7 [Records], page 71).

7.8 Using Scheme records in C

External C code can create records and access record slots positionally using these functions & macros. Note, however, that named access to record fields is not supported, only indexed access, so C code must be synchronized carefully with the corresponding Scheme that defines record types.

`s48_value s48_make_record` (*s48_value* ***record-type***) [C function (may GC)]
`int S48_RECORD_P` (*s48_value* ***object***) [C macro]
`s48_value S48_RECORD_TYPE` (*s48_value* ***record***) [C macro]
`s48_value S48_RECORD_REF` (*s48_value* ***record***, *long* ***index***) [C macro]
`void S48_RECORD_SET` (*s48_value* ***record***, *long* ***index***, *s48_value* ***value***) [C macro]
`void s48_check_record_type` (*s48_value* ***record***, *s48_value* ***type-binding***) [C function]

`s48_make_record` allocates a record on Scheme's heap with the given record type; its arguments must be a shared binding whose value is a record type descriptor (see Section 4.7 [Records], page 71). `S48_RECORD_P` is the type predicate for records. `S48_RECORD_TYPE` returns the record type descriptor of *record*. `S48_RECORD_REF` & `S48_RECORD_SET` operate on records similarly to how `S48_VECTOR_REF` & `S48_VECTOR_SET` work on vectors. `s48_check_record_type` checks whether *record* is a record whose type is the value of the shared binding *type_binding*. If this is not the case, it signals an exception. (It also signals an exception if *type_binding*'s value is not a record.) Otherwise, it returns normally.

For example, with this record type definition:

```
(define-record-type thing :thing
  (make-thing a b)
  thing?
  (a thing-a)
  (b thing-b))
```

the identifier `:thing` is bound to the record type and can be exported to C thus:

```
(define-exported-binding "thing-record-type" :thing)
```

and `thing` records can be made in C:

```
static s48_value thing_record_type = S48_FALSE;
void initialize_things(void)
{
  S48_GC_PROTECT_GLOBAL(thing_record_type);
  thing_record_type = s48_get_imported_binding("thing-record-type");
}

s48_value make_thing(s48_value a, s48_value b)
{
  s48_value thing;

  S48_DECLARE_GC_PROTECT(2);
  S48_GC_PROTECT_2(a, b);

  thing = s48_make_record(thing_record_type);
  S48_RECORD_SET(thing, 0, a);
  S48_RECORD_SET(thing, 1, b);

  S48_GC_UNPROTECT();

  return thing;
}
```

Note that the variables `a` & `b` must be protected against the possibility of a garbage collection occurring during the call to `s48_make_record`.

7.9 Raising exceptions from C

The following macros raise certain errors, immediately returning to Scheme48. Raising an exception performs all necessary clean-up actions to properly return to Scheme48, including adjusting the stack of protected variables.

`s48_raise_scheme_exception` (*int* `type`, *int* `nargs`, ...) [C function]
: The base procedure for raising exceptions. *Type* is the type of exception; it should be one of the `S48_EXCEPTION_...` constants defined in `scheme48.h`. *Nargs* is the number of additional values to be included in the exception; these follow the *nargs* argument and should all have the type `s48_value`. *Nargs* may not be greater than ten. `s48_raise_scheme_exception` never returns.

`s48_raise_argument_type_error` (*s48_value* `arg`) [C function]

`s48_raise_argument_number_error` (*s48_value* `nargs`, *s48_value* `min`, *s48_value* `max`) [C function]

`s48_raise_range_error` (*s48_value* `value`, *s48_value* `min`, *s48_value* `max`) [C function]

`s48_raise_closed_channel_error` () [C function]
`s48_raise_os_error` (*int* *errno*) [C function]
`s48_raise_out_of_memory_error` () [C function]

Conveniences for raising certain kinds of exceptions. Argument type errors are due to procedures receiving arguments of the incorrect type. Argument number errors are due to the number of arguments being passed to a procedure, *nargs*, not being between *min* or *max*, inclusive. Range errors are similar, but they are intended for larger ranges, not argument numbers. Closed channel errors occur when a channel (see Section 4.5.4 [Channels], page 65) was operated upon with the expectation that it would not be closed. OS errors originate from the OS, and they are denoted with Unix `errno` values.

`void S48_CHECK_BOOLEAN` (*s48_value* *object*) [C macro]
`void S48_CHECK_SYMBOL` (*s48_value* *object*) [C macro]
`void S48_CHECK_PAIR` (*s48_value* *object*) [C macro]
`void S48_CHECK_STRING` (*s48_value* *object*) [C macro]
`void S48_CHECK_INTEGER` (*s48_value* *object*) [C macro]
`void S48_CHECK_CHANNEL` (*s48_value* *object*) [C macro]
`void S48_CHECK_BYTE_VECTOR` (*s48_value* *object*) [C macro]
`void S48_CHECK_RECORD` (*s48_value* *object*) [C macro]
`void S48_CHECK_SHARED_BINDING` (*s48_value* *object*) [C macro]

Conveniences for checking argument types. These signal argument type errors with `s48_raise_argument_type_error` if their argument is not of the type being tested.

7.10 Unsafe C macros

All of the C functions & macros described previously verify that their arguments have the appropriate types and lie in the appropriate ranges. The following macros are identical to their safe counterparts, except that the unsafe variants, by contrast, do *not* verify coherency of their arguments. They are provided for the purpose of writing more efficient code; their general use is not recommended.

`char S48_UNSAFE_EXTRACT_CHAR` (*s48_value* *char*) [C macro]
`char * S48_UNSAFE_EXTRACT_STRING` (*s48_value* *string*) [C macro]
`long S48_UNSAFE_EXTRACT_INTEGER` (*s48_value* *integer*) [C macro]
`double S48_UNSAFE_EXTRACT_DOUBLE` (*s48_value* *double*) [C macro]
`long S48_UNSAFE_EXTRACT_FIXNUM` (*s48_value* *fixnum*) [C macro]
`s48_value S48_UNSAFE_ENTER_FIXNUM` (*long* *integer*) [C macro]

`s48_value S48_UNSAFE_CAR` (*s48_value* *pair*) [C macro]
`s48_value S48_UNSAFE_CDR` (*s48_value* *pair*) [C macro]
`void S48_UNSAFE_SET_CAR` (*s48_value* *pair*, *s48_value* *value*) [C macro]
`void S48_UNSAFE_SET_CDR` (*s48_value* *pair*, *s48_value* *value*) [C macro]

`long S48_UNSAFE_VECTOR_LENGTH (`*s48_value* `vector``)` [C macro]
`s48_value S48_UNSAFE_VECTOR_REF (`*s48_value* `vector`, *long* `index``)` [C macro]
`void S48_UNSAFE_VECTOR_SET (`*s48_value* `vector`, *long* `index`, *s48_value* `value``)` [C macro]

`long S48_UNSAFE_STRING_LENGTH (`*s48_value* `string``)` [C macro]
`char S48_UNSAFE_STRING_REF (`*s48_value* `string`, *long* `index``)` [C macro]
`void S48_UNSAFE_STRING_SET (`*s48_value* `string`, *long* `index`, *char* `char``)` [C macro]

`void S48_UNSAFE_SYMBOL_TO_STRING (`*s48_value* `symbol``)` [C macro]

`long S48_UNSAFE_BYTE_VECTOR_LENGTH (`*s48_value* `bytev``)` [C macro]
`char S48_UNSAFE_BYTE_VECTOR_REF (`*s48_value* `bytev`, *long* `index``)` [C macro]
`void S48_UNSAFE_BYTE_VECTOR_SET (`*s48_value* `bytev`, *long* `index`, *char* `byte``)` [C macro]

`s48_value S48_UNSAFE_SHARED_BINDING_REF (`*s48_value* `shared_binding``)` [C macro]
`int S48_UNSAFE_SHARED_BINDING_IS_IMPORTP (`*s48_value* `shared_binding``)` [C macro]
`s48_value S48_UNSAFE_SHARED_BINDING_NAME (`*s48_value* `shared_binding``)` [C macro]
`void S48_UNSAFE_SHARED_BINDING_SET (`*s48_value* `shared_binding`, *s48_value* `value``)` [C macro]

type `S48_UNSAFE_EXTRACT_VALUE (`*s48_value* `bytev`, *type*`)` [C macro]
type `* S48_UNSAFE_EXTRACT_VALUE_POINTER (`*s48_value* `bytev`, *type*`)` [C macro]
`void S48_UNSAFE_SET_VALUE (`*s48_value* `bytev`, *type*, *type* `value``)` [C macro]

8 POSIX interface

This chapter describes Scheme48's interface to POSIX C calls. Scheme versions of most of the C functions in POSIX are provided. Both the interface and implementation are new and likely to change significantly in future releases. The implementation may also contain many bugs.

The POSIX bindings are available in several structures:

`posix-processes`
: `fork`, `exec`, and other process manipulation procedures

`posix-process-data`
: procedures for accessing information about processes

`posix-files`
: POSIX file system access procedures

`posix-i/o`
: pipes and various POSIX I/O controls

`posix-time`
: POSIX time operations

`posix-users`
: user and group manipulation procedures

`posix-regexps`
: POSIX regular expression construction and matching

`posix` all of the above

Scheme48's POSIX interface differs from scsh [Shivers 94; Shivers 96; Shivers *et al.* 04] in several ways. The interface here lacks scsh's high-level constructs and utilities such as the process notation, `awk` facility, and parsing utilities. Scheme48 uses disjoint types for some values that scsh leaves as symbols or simple integers; these include file types, file modes, and user & group ids. Many of the names and other interface details are different as well.

8.1 Processes

The procedures described in this section control the creation of subprocesses and the execution of programs. They exported by both the `posix-processes` and `posix` structures.

`fork` ⟶ *process id or* `#f` [procedure]
`fork-and-forget` *thunk* ⟶ *unspecified* [procedure]
: `Fork` creates a new child process. In the parent process, it returns the child's process id; in the child process, it returns `#f`. `Fork-and-forget` calls *thunk* in a new process; no process id is returned. `Fork-and-forget` uses an intermediate process to avoid creating a zombie.

`process-id?` *object* ⟶ *boolean* [procedure]
`process-id=?` pid_a pid_b ⟶ *boolean* [procedure]
`process-id->integer` *pid* ⟶ *integer* [procedure]
`integer->process-id` *integer* ⟶ *pid* [procedure]

`Process-id?` is the disjoint type predicate for process ids. `Process-id=?` tests whether two process ids are the same. `Process-id->integer` & `integer->process-id` convert between Scheme48's opaque process id type and POSIX process id integers.

`process-id-exit-status` *pid* ⟶ *integer or* `#f` [procedure]
`process-id-terminating-signal` *pid* ⟶ *signal or* `#f` [procedure]
`wait-for-child-process` *pid* ⟶ *unspecified* [procedure]

If the process identified by *pid* exited normally or is running, `process-id-exit-status` and `process-id-terminating-signal` will both return `#f`. If, however, it terminated abnormally, `process-id-exit-status` returns its exit status, and if it exited due to a signal then `process-id-terminating-signal` returns the signal due to which it exited. `Wait-for-child-process` blocks the current process until the process identified by *pid* has terminated. Scheme48 may reap child processes before the user requests their exit status, but it does not always do so.

`exit` *status* ⟶ *does not return* [procedure]

Terminates the current process with the integer *status* as its exit status.

`exec` *program argument ...* ⟶ *does not return* [procedure]
`exec-with-environment` *program env argument ...* ⟶ *does not return* [procedure]
`exec-file` *filename argument ...* ⟶ *does not return* [procedure]
`exec-file-with-environment` *filename env argument ...* ⟶ *does not return* [procedure]

These all replace the current program with a new one. They differ in how the program is found and what process environment the program should receive. `Exec` & `exec-with-environment` look up the program in the search path (the `PATH` environment variable), while `exec-file` & `exec-file-with-environment` execute a particular file. The environment is either inherited from the current process, in the cases of `exec` & `exec-file`, or explicitly specified, in the cases of `exec-with-environment` & `exec-file-with-environment`. *Program*, *filename*, & all *argument*s should be strings. *Env* should be a list of strings of the form `"name=value"`. When the new program is invoked, its arguments consist of the program name prepended to the remaining specified arguments.

`exec-with-alias` *name lookup? maybe-env arguments* ⟶ *does not return* [procedure]

General omnibus procedure that subsumes the other `exec` variants. *Name* is looked up in the search path if *lookup?* is true or used as an ordinary filename if it is false. *Maybe-env* is either `#f`, in which case the new program's environment should be inherited from the current process, or a list of strings of the above form for environments, which specifies the new program's environment. *Arguments* is a list of *all* of the program's arguments; `exec-with-alias` does *not* prepend *name* to that list (hence `-with-alias`).

8.2 Signals

There are two varieties of signals available, named & anonymous. A *named* signal is one for which there is provided a symbolic name, such as `kill` or `pipe`. Anonymous signals are those that the operating system provided but for which POSIX does not define a symbolic name, only a number, and which may not have meaning on other operating systems. Named signals preserve their meaning through heap image dumps; anonymous signals may not be dumped in heap images. (If they are, a warning is signalled, and they are replaced with a special token that denotes a non-portable signal.) Not all named signals are available from all operating systems, and there may be multiple names for a single operating system signal number.

`signal` *name* $\longrightarrow$ *signal* [syntax]
`name->signal` *symbol* $\longrightarrow$ *signal or* `#f` [procedure]
`integer->signal` *integer* $\longrightarrow$ *signal* [procedure]
`signal?` *object* $\longrightarrow$ *boolean* [procedure]
`signal-name` *signal* $\longrightarrow$ *symbol or* `#f` [procedure]
`signal-os-number` *signal* $\longrightarrow$ *integer* [procedure]
`signal=?` $signal_a$ $signal_b$ $\longrightarrow$ *boolean* [procedure]

`Signal` evaluates to the signal object with the known symbolic name *name*. It is an error if *name* is not recognized as any signal's name. `Name->signal` returns the signal corresponding with the given *name* or `#f` if no such signal is known. `Integer->signal` returns a signal, named or anonymous, with the given OS number. `Signal?` is the disjoint type predicate for signal objects. `Signal-name` returns the symbolic name of *signal* if it is a named signal or `#f` if it is an anonymous signal. `Signal-OS-number` returns the operating system's integer value of *signal*. `Signal=?` tests whether two signals are the same, *i.e.* whether their OS numbers are equal equal.

These are all of the symbols that POSIX defines.

`abrt` abnormal termination (as by `abort(3)`)

`alrm` timeout signal (as by `alarm(2)`)

`fpe` floating point exception

`hup` hangup on controlling terminal or death of controlling process

`ill` illegal instruction

`int` interrupt — interaction attention

`kill` termination signal, cannot be caught or ignored

`pipe` write was attempted on a pipe with no readers

`quit` interaction termination

`segv` segmentation violation — invalid memory reference

`term` termination signal

`usr1`
`usr2` for use by applications

`chld` child process stopped or terminated

`cont` continue if stopped

`stop` stop immediately, cannot be caught or ignored

`tstp` interactive stop

`ttin` read from control terminal attempted by a background process

`ttou` write to control terminal attempted by a background process

`bus` bus error — access to undefined portion of memory

There are also several other signals whose names are allowed to be passed to `signal` that are not defined by POSIX, but that are recognized by many operating systems.

`trap` trace or breakpoint trap

`iot` synonym for `abrt`

`emt`

`sys` bad argument to routine (SVID)

`stkflt` stack fault on coprocessor

`urg` urgent condition on socket (4.2 BSD)

`io` I/O now possible (4.2 BSD)

`poll` synonym for `io` (System V)

`cld` synonym for `chld`

`xcpu` CPU time limit exceeded (4.2 BSD)

`xfsz` file size limit exceeded (4.2 BSD)

`vtalrm` virtual alarm clock (4.2 BSD)

`prof` profile alarm clock

`pwr` power failure (System V)

`info` synonym for `pwr`

`lock` file lock lost

`winch` Window resize signal (4.3 BSD, Sun)

`unused`

8.2.1 Sending & receiving signals

`signal-process` *pid signal* $\longrightarrow$ *unspecified* [procedure]

Sends a signal represented by *signal* to the process identified by *pid*.

Signals received by the Scheme process can be obtained via one or more *signal queues*. Each signal queue has a list of monitored signals and a queue of received signals that have yet to be consumed from the queue. When the Scheme process receives a signal, that signal is added to the signal queues that are currently monitoring the signal received.

`make-signal-queue` *signal-list* ⟶ *signal-queue* [procedure]
`signal-queue?` *object* ⟶ *boolean* [procedure]
`signal-queue-monitored-signals` *signal-queue* ⟶ *signal-list* [procedure]
`dequeue-signal!` *signal-queue* ⟶ *signal* (*may block*) [procedure]
`maybe-dequeue-signal!` *signal-queue* ⟶ *signal or* `#f` [procedure]

`Make-signal-queue` returns a new signal queue that will monitor all of the signals in the given list. `Signal-queue?` is the disjoint type predicate for signal queues. `Signal-queue-monitored-signals` returns a freshly-allocated list of the signals currently monitored by *signal-queue*. `Dequeue-signal!` & `maybe-dequeue-signal!` both access the next signal ready to be read from *signal-queue*. If the signal queue is empty, `dequeue-signal!` will block until a signal is received, while `maybe-dequeue-signal!` will immediately return `#f`.

Note: There is a bug in the current system that causes an erroneous deadlock to occur if threads are blocked waiting for signals and no other threads are available to run. A workaround is to create a thread that sleeps for a long time, which prevents any deadlock errors (including real ones):

```
> ,open threads
> (spawn (lambda ()
           ;; Sleep for a year.
           (sleep (* 1000 60 60 24 365))))
```

`add-signal-queue-signal!` *signal-queue signal* ⟶ *unspecified* [procedure]
`remove-signal-queue-signal!` *signal-queue signal* ⟶ *unspecified* [procedure]

These add & remove signals from signal queues' list of signals to monitor. Note that `remove-signal-queue-signal!` also removes any pending signals from the queue, so `dequeue-signal!` & `maybe-dequeue-signal!` will only ever return signals that are on the queue's list of monitored signals when they are called.

8.3 Process environment

These procedures are exported by the structures `posix` & `posix-process-data`.

`get-process-id` ⟶ *process-id* [procedure]
`get-parent-process-id` ⟶ *process-id* [procedure]

These return the process id (see Section 8.1 [POSIX processes], page 130) of the current process or the current process's parent, respectively.

`get-user-id` ⟶ *user-id* [procedure]
`get-effective-user-id` ⟶ *user-id* [procedure]
`set-user-id!` *user-id* ⟶ *unspecified* [procedure]
`get-group-id` ⟶ *group-id* [procedure]
`get-effective-group-id` ⟶ *group-id* [procedure]
`set-group-id!` *group-id* ⟶ *unspecified* [procedure]

These access the original and effective user & group ids (see Section 8.4 [POSIX users and groups], page 135) of the current process. The effective ids may be set, but not the original ones.

`get-groups` ⟶ *group-id list* [procedure]
`get-login-name` ⟶ *string* [procedure]
`Get-groups` returns a list of the supplementary groups of the current process. `Get-login-name` returns a user name for the current process.

`lookup-environment-variable` *string* ⟶ *string or* `#f` [procedure]
`environment-alist` ⟶ *alist* [procedure]
`Lookup-environment-variable` looks up its argument in the environment list of the current process and returns the corresponding string, or `#f` if there is none. `Environment-alist` returns the entire environment as a list of (*`name-string`* . *`value-string`*) pairs.

8.4 Users and groups

User ids & *group ids* are boxed integers that represent Unix users & groups. Also, every user & group has a corresponding *user info* or *group info* record, which contains miscellaneous information about the user or group. The procedures in this section are exported by the structures `posix-users` & `posix`.

`user-id?` *object* ⟶ *boolean* [procedure]
`user-id=?` uid_a uid_b ⟶ *boolean* [procedure]
`user-id->integer` *uid* ⟶ *integer* [procedure]
`integer->user-id` *integer* ⟶ *uid* [procedure]
`group-id?` *object* ⟶ *boolean* [procedure]
`group-id=?` gid_a gid_b ⟶ *boolean* [procedure]
`group-id->integer` *gid* ⟶ *integer* [procedure]
`integer->group-id` *integer* ⟶ *gid* [procedure]
`User-id?` & `group-id?` are the disjoint type predicates for user & group ids. `User-id=?` & `group-id=?` test whether two user or group ids, respectively, are the same, *i.e.* whether their numbers are equal. `User-id->integer`, `group-id->integer`, `integer->user-id`, & `integer->group-id` convert between user or group ids and integers.

`user-id->user-info` *uid* ⟶ *user-info* [procedure]
`name->user-info` *string* ⟶ *user-info* [procedure]
`group-id->group-info` *gid* ⟶ *group-info* [procedure]
`name->group-info` *string* ⟶ *group-info* [procedure]
These provide access for the user or group info records that correspond with the given user or group ids or names.

`user-info?` *object* ⟶ *boolean* [procedure]
`user-info-name` *user-info* ⟶ *string* [procedure]
`user-info-id` *user-info* ⟶ *user-id* [procedure]
`user-info-group` *user-info* ⟶ *group-id* [procedure]
`user-info-home-directory` *user-info* ⟶ *string* [procedure]
`user-info-shell` *user-info* ⟶ *string* [procedure]
`group-info?` *object* ⟶ *boolean* [procedure]
`group-info-name` *group-info* ⟶ *string* [procedure]
`group-info-id` *group-info* ⟶ *group-id* [procedure]

`group-info-members` *group-info* ⟶ *user-id-list* [procedure]
`User-info?` & `group-info?` are the disjoint type predicates for user info & group info records. The others are accessors for the various data available in those records.

8.5 Host OS and machine identification

`host-name` ⟶ *string* [procedure]
`os-node-name` ⟶ *string* [procedure]
`os-release-name` ⟶ *string* [procedure]
`os-version-name` ⟶ *string* [procedure]
`machine-name` ⟶ *string* [procedure]
These procedures return strings that are intended to identify various aspects of the current operating system and physical machine. POSIX does not specify the format of the strings. These procedures are provided by both the structure `posix-platform-names` and the structure `posix`.

8.6 File system access

These procedures operate on the file system via the facilities defined by POSIX and offer more than standard & portable R5RS operations. All of these names are exported by the structures `posix-files` and `posix`.

`open-directory-stream` *filename* ⟶ *dir-stream* [procedure]
`directory-stream?` *object* ⟶ *boolean* [procedure]
`read-directory-stream` *dir-stream* ⟶ *filename or* `#f` [procedure]
`close-directory-stream` *dir-stream* ⟶ *unspecified* [procedure]
Directory streams are the low-level interface provided by POSIX to enumerate the contents of a directory. `Open-directory-stream` opens a new directory stream that will enumerate all of the files within the directory named by *filename*. `Directory-stream?` is the disjoint type predicate for directory streams. `Read-directory-stream` consumes the next filename from *dir-stream* and returns it, or returns `#f` if the stream has finished. Note that `read-directory-stream` will return only simple filenames, not full pathnames. `Close-directory-stream` closes *dir-stream*, removing any storage it required in the operating system. Closing an already closed directory stream has no effect.

`list-directory` *filename* ⟶ *string list* [procedure]
Returns the list of filenames in the directory named by *filename*. This is equivalent to opening a directory stream, repeatedly reading from it & accumulating the list of filenames, and closing the stream.

`working-directory` ⟶ *string* [procedure]
`set-working-directory!` *string* ⟶ *unspecified* [procedure]
These access the working directory's filename of the current process.

`open-file` *pathname file-options* [*file-mode*] ⟶ *port* [procedure]
Opens a port to the file named by the string *pathname*. *File-options* specifies various aspects of the port. The optional *file-mode* argument is used only if the file to be

opened does not already exist; it specifies the permissions to be assigned to the file if it is created. The returned port is an input port if the given options include `read-only`; otherwise `open-file` returns an output port. Because Scheme48 does not support combined input/output ports, `dup-switching-mode` can be used to open an input port for output ports opened with the `read-write` option.

File options are stored in a boxed mask representation. File option sets are created with `file-options` and tested with `file-options-on?`.

`file-options` *name ...* $\longrightarrow$ *file-options* [syntax]
`file-options-on?` $options_a$ $options_b$ $\longrightarrow$ *boolean* [procedure]

`File-options` evaluates to a file option set, suitable for passing to `open-file`, that includes all of the given named options. `File-options-on?` returns true if $options_a$ includes all of the options in $options_b$, or false if otherwise.

The following file option names are supported as arguments to the `file-options` syntax:

`create` create file if it does not already exist; a *file-mode* argument is required to be passed to `open-file` if the `create` option is specified

`exclusive` an error will be signalled if this option & `create` are both set and the file already exists

`no-controlling-tty` if the pathname being opened is a terminal device, the terminal will not become the controlling terminal of the process

`truncate` file is truncated

`append` written data to the newly opened file will be appended to the existing contents

`nonblocking` read & write operations will not block

`read-only` file may not be written to, only read from

`read-write` file may be both read from & written to

`write-only` file may not be read from, only written to

The last three are all mutually exclusive.

Examples:

```
(open-file "some-file.txt"
           (file-options create write-only)
           (file-mode read owner-write))
```

returns an output port that writes to a newly-created file that can be read from by anyone but written to only by the owner. Once the file `some-file.txt` exists,

```
(open-file "some-file.txt"
```

```
(file-options append write-only))
```

will open an output port that appends to the file.

`I/o-flags` & `set-i/o-flags!` (see Section 8.8 [POSIX I/O utilities], page 141) can be used to access the `append`, `nonblocking`, and read/write file options of ports, as well as modify the `append` & `nonblocking` options.

To keep port operations from blocking in the Scheme48 process, output ports are set to be nonblocking at the time of creation. (Input ports are managed using `select(2)`.) `Set-i/o-flags!` can be used to make an output port blocking, for example directly before forking, but care should be exercised, because the Scheme48 run-time system may be confused if an I/O operation blocks.

`set-file-creation-mask!` *file-mode* ⟶ *file-mode* [procedure]
: Sets the file creation mask to be *file-mode*. Bits set in *file-mode* are cleared in the modes of any files or directories subsequently created by the current process.

`link` *existing-pathname new-pathname* ⟶ *unspecified* [procedure]
`make-directory` *pathname file-mode* ⟶ *unspecified* [procedure]
`make-fifo` *pathname file-mode* ⟶ *unspecified* [procedure]
: `Link` creates a hard link for the file at *existing-pathname* at *new-pathname*. `Make-directory` creates a new directory at the locations specified by *pathname* with the the file mode *file-mode*. `Make-fifo` does similarly, but it creates a FIFO (first-in first-out) file instead of a directory.

`unlink` *pathname* ⟶ *unspecified* [procedure]
`remove-directory` *pathname* ⟶ *unspecified* [procedure]
`rename` *old-pathname new-pathname* ⟶ *unspecified* [procedure]
: `Unlink` removes a link at the location *pathname*. `Remove-directory` removes a directory at the location specified by *pathname*. The directory must be empty; an exception is signalled if it is not. `Rename` moves the file at the location *old-pathname* to the new location *new-pathname*.

`accessible?` *pathname access-mode more-modes . . .* ⟶ *boolean* [procedure]
`access-mode` *name* ⟶ *access mode* [syntax]
: `Accessible?` returns true if *pathname* is accessible by all of the given access modes. (There must be at least one access mode argument.) `Access-mode` evaluates to an access mode, suitable for passing to `accessible?`, from the given name. The allowed names are `read`, `write`, `execute`, & `exists`.

Information about files can be queried using the *file info* abstraction. Every file has a corresponding file info record, which contains various data about the file including its name, its type, its device & inode numbers, the number of links to it, its size in bytes, its owner, its group, its file mode, and its access times.

`get-file-info` *pathname* ⟶ *file-info* [procedure]
`get-file/link-info` *pathname* ⟶ *file-info* [procedure]
`get-port-info` *fd-port* ⟶ *file-info* [procedure]
: `Get-file-info` & `get-file/link-info` return a file info record for the files named by *pathname*. `Get-file-info` follows symbolic links, however, while `get-file/link`

info does not. `Get-port-info` returns a file info record for the file that *fd-port* is a port atop a file descriptor for. If *fd-port* does not read from or write to a file descriptor, an error is signalled.

`file-info?` *object* ⟶ *boolean* [procedure]
`file-info-name` *file-info* ⟶ *string* [procedure]
`file-info-device` *file-info* ⟶ *integer* [procedure]
`file-info-inode` *file-info* ⟶ *integer* [procedure]
`file-info-link-count` *file-info* ⟶ *integer* [procedure]
`file-info-size` *file-info* ⟶ *integer* [procedure]
`file-info-owner` *file-info* ⟶ *user-id* [procedure]
`file-info-group` *file-info* ⟶ *group-id* [procedure]
`file-info-mode` *file-info* ⟶ *file-mode* [procedure]
`file-info-last-access` *file-info* ⟶ *time* [procedure]
`file-info-last-modification` *file-info* ⟶ *time* [procedure]
`file-info-last-change` *file-info* ⟶ *time* [procedure]

Accessors for various file info record fields. The name is the string passed to `get-file-info` or `get-file/link-info`, if the file info record was created with either of those two, or the name of the file that the file descriptor of the port queried was created on, if the file info record was obtained with `get-port-info`.

`file-info-type` *file-info* ⟶ *file-type* [procedure]
`file-type` *name* ⟶ *file-type* [syntax]
`file-type?` *object* ⟶ *boolean* [procedure]
`file-type-name` *file-type* ⟶ *symbol* [procedure]

`File-info-type` returns the type of the file as a *file type* object. File types may be compared using `eq?`. `File-type` evaluates to a file type of the given name. The disjoint type predicate for file types is `file-type?`. `File-type-name` returns the symbolic name that represents *file-type*.

The valid file type names are:

`regular`
`directory`
`character-device`
`block-device`
`fifo`
`symbolic-link` (not required by POSIX)
`socket` (not required by POSIX)
`other`

File modes are boxed integers that represent POSIX file protection masks.

`file-mode` *permission-name* ... ⟶ *file-mode* [syntax]
`file-mode?` *object* ⟶ *boolean* [procedure]

`File-mode` evaluates to a file mode object that contains all of the specified permissions. `File-mode?` is the disjoint type predicate for file mode descriptor objects. These are all of the names, with their corresponding octal bit masks and meanings, allowed to be passed to `file-mode`:

Permission name	Octal mask	Description
`set-uid`	`#o4000`	set user id when executing
`set-gid`	`#o2000`	set group id when executing
`owner-read`	`#o0400`	read by owner
`owner-write`	`#o0200`	write by owner
`owner-exec`	`#o0100`	execute (or search) by owner
`group-read`	`#o0040`	read by group
`group-write`	`#o0020`	write by group
`group-exec`	`#o0010`	execute (or search) by group
`other-read`	`#o0004`	read by others
`other-write`	`#o0002`	write by others
`other-exec`	`#o0001`	execute (or search) by others

Also, several compound masks are supported for convenience:

Permission set name	Octal mask	Description
`owner`	`#o0700`	read, write, & execute by owner
`group`	`#o0070`	read, write, & execute by group
`other`	`#o0007`	read, write, & execute by others
`read`	`#o0444`	read by anyone
`write`	`#o0111`	write by anyone
`exec`	`#o0777`	read, write, & execute by anyone

`file-mode+` *file-mode ...* $\longrightarrow$ *file-mode* [procedure]
`file-mode-` *file-mode*$_a$ *file-mode*$_b$ $\longrightarrow$ *file-mode* [procedure]
`file-mode=?` *file-mode*$_a$ *file-mode*$_b$ $\longrightarrow$ *boolean* [procedure]
`file-mode<=?` *file-mode*$_a$ *file-mode*$_b$ $\longrightarrow$ *boolean* [procedure]
`file-mode>=?` *file-mode*$_a$ *file-mode*$_b$ $\longrightarrow$ *boolean* [procedure]

`File-mode+` returns a file mode that contains all of the permissions specified in any of its arguments. `File-mode-` returns a file mode that contains all of *file-mode*$_a$'s permissions not in *file-mode*$_b$. `File-mode=?` tests whether two file modes are the same. `File-mode<=?` returns true if each successive file mode argument has the same or more permissions as the previous one. `File-mode>=?` returns true if each successive file mode argument has the same or fewer permissions as the previous one.

`file-mode->integer` *file-mode* $\longrightarrow$ *integer* [procedure]
`integer->file-mode` *integer* $\longrightarrow$ *file-mode* [procedure]

These convert between file mode objects and Unix file mode masks as integers. The integer representations may or may not be the masks used by the underlying operating system.

8.7 Time

A *time* record contains an integer that represents a time as the number of seconds since the Unix epoch (00:00:00 GMT, January 1, 1970). These procedures for operating on time records are in the structures `posix-time` & `posix`.

`make-time` *seconds* $\longrightarrow$ *time* [procedure]
`current-time` $\longrightarrow$ *time* [procedure]
`time?` *object* $\longrightarrow$ *boolean* [procedure]

`time-seconds` *time* ⟶ *integer* [procedure]
: `Make-time` & `current-time` construct time records; `make-time` uses the number of seconds that is its argument, and `current-time` uses the current number of seconds since the epoch. `Time?` is the disjoint type predicate for time objects. `Time-seconds` returns the number of seconds recorded by *time*.

`time=?` $time_a$ $time_b$ ⟶ *boolean* [procedure]
`time<?` $time_a$ $time_b$ ⟶ *boolean* [procedure]
`time<=?` $time_a$ $time_b$ ⟶ *boolean* [procedure]
`time>?` $time_a$ $time_b$ ⟶ *boolean* [procedure]
`time>=?` $time_a$ $time_b$ ⟶ *boolean* [procedure]
: Various time comparators. `Time=?` returns true if its two arguments represent the same number of seconds since the epoch. `Time<?`, `time<=?`, `time>?`, & `time>=` return true if their arguments are monotonically increasing, monotonically non-decreasing, monotonically decreasing, or monotonically non-increasing, respectively.

`time->string` *time* ⟶ *string* [procedure]
: Returns a string representation of *time* in the format of "*DDD MMM HH:MM:SS YYYY*". For example,

```
(time->string (make-time 1234567890))
    ⇒ "Fri Feb 13 18:31:30 2009"
```

Note: The string has a newline suffix.

8.8 I/O utilities

These procedures for manipulating pipes and ports built on file descriptors are provided by the structures `posix-i/o` & `posix`.

`open-pipe` ⟶ [*input-port output-port*] [procedure]
: Creates a pipe and returns the two ends of the pipe as an input port & an output port.

A *file descriptor port* (or *fd-port*) is a port or a channel (see Section 4.5.4 [Channels], page 65) that reads from or writes to an OS file descriptor. File descriptor ports are returned by the standard Scheme procedures `open-input-file` & `open-output-file` as well as the procedures `open-file` & `open-pipe` from this POSIX interface.

`fd-port?` *port* ⟶ *boolean* [procedure]
`port->fd` *port* ⟶ *integer or* `#f` [procedure]
: `Fd-port?` returns true if *port* is a port that reads from or writes to a file descriptor, or false if not. `Port->fd` returns the file descriptor that *port* reads from or writes to, if it is a file descriptor port, or `#f` if it is not. It is an error to pass a value that is not a port to either of these procedures.

  Note: Channels may *not* be passed to these procedures. To access a channel's file descriptor, use `channel-os-index`; see Section 4.5.4 [Channels], page 65, for more details.

`remap-file-descriptors!` *fd-spec ...* $\longrightarrow$ *unspecified* [procedure]

Reassigns file descriptors to ports. Each *fd-spec* specifies what port is to be mapped to what file descriptor: the first port gets file descriptor `0`; the second, `1`; and so on. An *fd-spec* is either a port that reads from or writes to a file descriptor or `#f`; in the latter case, the corresponding file descriptor is not used. Any open ports not listed are marked *close-on-exec*. The same port may be moved to multiple new file descriptors.

For example,

```
(remap-file-descriptors! (current-output-port)
                         #f
                         (current-input-port))
```

moves the current output port to file descriptor `0` (*i.e.* `stdin`) and the current input port to file descriptor `2` (*i.e.* `stderr`). File descriptor `1` (`stdout`) is not mapped to anything, and all other open ports (including anything that had the file descriptor `1`) are marked *close-on-exec*.

`dup` *fd-port* $\longrightarrow$ *fd-port* [procedure]
`dup-switching-mode` *fd-port* $\longrightarrow$ *fd-port* [procedure]
`dup2` *fd-port fdes* $\longrightarrow$ *fd-port* [procedure]

These change *fd-port*'s file descriptor and return new ports that have the ports' old file descriptors. `Dup` uses the lowest unused file descriptor; `dup2` uses the one provided. `Dup-switching-mode` is the same as `dup` except that the returned port is an input port if the argument was an output port and vice versa. If any existing port uses the file descriptor passed to `dup2`, that port is closed.

`close-all-port` *port-or-channel ...* $\longrightarrow$*unspecified* [procedure]

Closes all ports or channels not listed as arguments.

`close-on-exec?` *channel* $\longrightarrow$ *boolean* [procedure]
`set-close-on-exec?!` *channel boolean* $\longrightarrow$ *unspecified* [procedure]

These access the boolean flag that specifies whether *channel* will be closed when a new program is `exec`'d.

`i/o-flags` *fd-port* $\longrightarrow$ *file-options* [procedure]
`set-i/o-flags!` *fd-port file-options* $\longrightarrow$ *unspecified* [procedure]

These access various file options (see Section 8.6 [POSIX file system access], page 136) for *fd-port*. The options that may be read are `append`, `nonblocking`, `read-only`, `read-write`, and `write-only`; only the `append` and `nonblocking` options can be written.

`port-is-a-terminal?` *port* $\longrightarrow$ *boolean* [procedure]
`port-terminal-name` *port* $\longrightarrow$ *string or* `#f` [procedure]

`Port-is-a-terminal?` returns true of *port* is a port that has an underlying file descriptor associated with a terminal. For such ports, `port-terminal-name` returns the name of the terminal; for all others, it returns `#f`.

Note: These procedures accept only ports, not channels.

8.9 Regular expressions

The procedures in this section provide access to POSIX regular expression matching. The regular expression syntax and semantics are far too complex to be described here.

Note: Because the C interface uses ASCII `NUL` bytes to mark the ends of strings, patterns & strings that contain `NUL` characters will not work correctly.

8.9.1 Direct POSIX regular expression interface

The first interface to regular expressions is a thin layer over the interface that POSIX provides. It is exported by the structures `posix-regexps` & `posix`.

`make-regexp` *string option ...* $\longrightarrow$ *regexp* [procedure]
`regexp?` *object* $\longrightarrow$ *boolean* [procedure]

`Make-regexp` creates a regular expression with the given string pattern. The arguments after *string* specify various options for the regular expression; see `regexp-option` below. The regular expression is not compiled until it is matched against a string, so any errors in the pattern string will not be reported until that point. `Regexp?` is the disjoint type predicate for regular expression objects.

`regexp-option` *name* $\longrightarrow$ *regexp-option* [syntax]

Evaluates to a regular expression option, suitable to be passed to `make-regexp`, with the given name. The possible option names are:

`extended` use the extended patterns

`ignore-case`
: ignore case differences when matching

`submatches`
: report submatches

`newline` treat newlines specially

`regexp-match` *regexp string start submatches? starts-line? ends-line?* $\longrightarrow$ *boolean or list of matches* [procedure]

`Regexp-match` matches *regexp* against the characters in *string*, starting at position *start*. If the string does not match the regular expression, `regexp-match` returns `#f`. If the string does match, then a list of match records is returned if *submatches?* is true or `#t` if *submatches?* is false. The first match record gives the location of the substring that matched *regexp*. If the pattern in *regexp* contained submatches, then the submatches are returned in order, with match records in the positions where submatches succeeded and `#f` in the positions where submatches failed.

Starts-line? should be true if *string* starts at the beginning of a line, and *ends-line?* should be true if it ends one.

`match?` *object* $\longrightarrow$ *boolean* [procedure]
`match-start` *match* $\longrightarrow$ *integer* [procedure]
`match-end` *match* $\longrightarrow$ *integer* [procedure]
`match-submatches` *match* $\longrightarrow$ *alist* [procedure]

`Match?` is the disjoint type predicate for match records. Match records contain three values: the beginning & end of the substring that matched the pattern and an association list of submatch keys and corresponding match records for any named submatches

that also matched. `Match-start` returns the index of the first character in the matching substring, and `match-end` gives the index of the first character after the matching substring. `Match-submatches` returns the alist of submatches.

8.9.2 High-level regular expression construction

This section describes a functional interface for building regular expressions and matching them against strings, higher-level than the direct POSIX interface. The matching is done using the POSIX regular expression package. Regular expressions constructed by procedures listed here are compatible with those in the previous section; that is, they satisfy the predicate `regexp?` from the `posix-regexps` structure. These names are exported by the structure `regexps`.

8.9.2.1 Character sets

Character sets may be defined using a list of characters and strings, using a range or ranges of characters, or by using set operations on existing character sets.

`set` *char-or-string ...* $\longrightarrow$ *char-set-regexp* [procedure]
`range` *low-char high-char* $\longrightarrow$ *char-set-regexp* [procedure]
`ranges` *low-char high-char ...* $\longrightarrow$ *char-set-regexp* [procedure]
`ascii-range` *low-char high-char* $\longrightarrow$ *char-set-regexp* [procedure]
`ascii-ranges` *low-char high-char ...* $\longrightarrow$ *char-set-regexp* [procedure]

`Set` returns a character set that contains all of the character arguments and all of the characters in all of the string arguments. `Range` returns a character set that contains all characters between *low-char* and *high-char*, inclusive. `Ranges` returns a set that contains all of the characters in the given set of ranges. `Range` & `ranges` use the ordering imposed by `char->integer`. `Ascii-range` & `ascii-ranges` are like `range` & `ranges`, but they use the ASCII ordering. `Ranges` & `ascii-ranges` must be given an even number of arguments. It is an error for a *high-char* to be less than the preceding *low-char* in the appropriate ordering.

`negate` *char-set* $\longrightarrow$ *char-set-regexp* [procedure]
`union` *char-set*$_a$ *char-set*$_b$ $\longrightarrow$ *char-set-regexp* [procedure]
`intersection` *char-set*$_a$ *char-set*$_b$ $\longrightarrow$ *char-set-regexp* [procedure]
`subtract` *char-set*$_a$ *char-set*$_b$ $\longrightarrow$ *char-set-regexp* [procedure]

Set operations on character sets. `Negate` returns a character set of all characters that are not in *char-set*. `Union` returns a character set that contains all of the characters in *char-set*$_a$ and all of the characters in *char-set*$_b$. `Intersection` returns a character set of all of the characters that are in both *char-set*$_a$ and *char-set*$_b$. `Subtract` returns a character set of all the characters in *char-set*$_a$ that are not also in *char-set*$_b$.

```
lower-case = (set "abcdefghijklmnopqrstuvwxyz")                    [character set]
lower-case = (set "abcdefghijklmnopqrstuvwxyz")                    [character set]
upper-case = (set "ABCDEFGHIJKLMNOPQRSTUVWXYZ")                    [character set]
alphabetic = (union lower-case upper-case)                         [character set]
numeric = (set "0123456789")                                       [character set]
alphanumeric = (union alphabetic numeric)                          [character set]
punctuation = (set "!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~")           [character set]
graphic = (union alphanumeric punctuation)                         [character set]
```

`printing` = (`union graphic` (`set #\space`)) [character set]
`control` = (`negate printing`) [character set]
`blank` = (`set #\space` (`ascii->char 9`)) `; ASCII 9 = TAB` [character set]
`whitespace` = (`union` (`set #\space`) (`ascii-range 9 13`)) [character set]
`hexdigit` = (`set "0123456789ABCDEF"`) [character set]
Predefined character sets.

8.9.2.2 Anchoring

`string-start` ⟶ *regexp* [procedure]
`string-end` ⟶ *regexp* [procedure]
`String-start` returns a regular expression that matches the beginning of the string being matched against; `string-end` returns one that matches the end.

8.9.2.3 Composite expressions

`sequence` *regexp* ... ⟶ *regexp* [procedure]
`one-of` *regexp* ... ⟶ *regexp* [procedure]
`Sequence` returns a regular expression that matches concatenation of all of its arguments; `one-of` returns a regular expression that matches any one of its arguments.

`text` *string* ⟶ *regexp* [procedure]
Returns a regular expression that matches exactly the characters in *string*, in order.

`repeat` *regexp* ⟶ *regexp* [procedure]
`repeat` *count regexp* ⟶ *regexp* [procedure]
`repeat` *min max regexp* ⟶ *regexp* [procedure]
`Repeat` returns a regular expression that matches zero or more occurrences of its *regexp* argument. With only one argument, the result will match *regexp* any number of times. With two arguments, *i.e.* one *count* argument, the returned regular expression will match *regexp* exactly that number of times. The final case will match from *min* to *max* repetitions, inclusive. *Max* may be `#f`, in which case there is no maximum number of matches. *Count* & *min* must be exact, non-negative integers; *max* should be either `#f` or an exact, non-negative integer.

8.9.2.4 Case sensitivity

Regular expressions are normally case-sensitive, but case sensitivity can be manipulated simply.

`ignore-case` *regexp* ⟶ *regexp* [procedure]
`use-case` *regexp* ⟶ *regexp* [procedure]
The regular expression returned by `ignore-case` is identical to its argument except that the case will be ignored when matching. The value returned by `use-case` is protected from future applications of `ignore-case`. The expressions returned by `use-case` and `ignore-case` are unaffected by any enclosing uses of these procedures.

By way of example, the following matches `"ab"`, but not `"aB"`, `"Ab"`, or `"AB"`:

```
(text "ab")
```

while

```
(ignore-case (text "ab"))
```

matches all of those, and

```
(ignore-case (sequence (text "a")
                       (use-case (text "b"))))
```

matches `"ab"` or `"Ab"`, but not `"aB"` or `"AB"`.

8.9.2.5 Submatches and matching

A subexpression within a larger expression can be marked as a submatch. When an expression is matched against a string, the success or failure of each submatch within that expression is reported, as well as the location of the substring matched by each successful submatch.

`submatch` *key regexp* ⟶ *regexp* [procedure]
`no-submatches` *regexp* ⟶ *regexp* [procedure]

`Submatch` returns a regular expression that is equivalent to *regexp* in every way except that the regular expression returned by `submatch` will produce a submatch record in the output for the part of the string matched by *regexp*. `No-submatches` returns a regular expression that is equivalent to *regexp* in every respect except that all submatches generated by *regexp* will be ignored & removed from the output.

`any-match?` *regexp string* ⟶ *boolean* [procedure]
`exact-match?` *regexp string* ⟶ *boolean* [procedure]
`match` *regexp string* ⟶ *match or* `#f` [procedure]

`Any-match?` returns `#t` if *string* matches *regexp* or contains a substring that does, or `#f` if otherwise. `Exact-match?` returns `#t` if *string* matches *regexp* exactly, or `#f` if it does not.

`Match` returns `#f` if *string* does not match *regexp*, or a match record if it does, as described in the previous section. Matching occurs according to POSIX. The match returned is the one with the lowest starting index in *string*. If there is more than one such match, the longest is returned. Within that match, the longest possible submatches are returned.

All three matching procedures cache a compiled version of *regexp*. Subsequent calls with the same input regular expression will be more efficient.

Here are some examples of the high-level regular expression interface:

```
(define pattern (text "abc"))

(any-match? pattern "abc")            ⇒ #t
(any-match? pattern "abx")            ⇒ #f
(any-match? pattern "xxabcxx")        ⇒ #t

(exact-match? pattern "abc")          ⇒ #t
(exact-match? pattern "abx")          ⇒ #f
(exact-match? pattern "xxabcxx")      ⇒ #f
```

```
(let ((m (match (sequence (text "ab")
                          (submatch 'foo (text "cd"))
                          (text "ef")))
         "xxabcdefxx"))
  (list m (match-submatches m)))
    ⇒ (#{Match 3 9} ((foo . #{Match 5 7})))

(match-submatches
 (match (sequence (set "a")
                  (one-of (submatch 'foo (text "bc"))
                          (submatch 'bar (text "BC"))))
        "xxxaBCd"))
    ⇒ ((bar . #{Match 4 6}))
```

8.10 C to Scheme correspondence

access	accessible?
chdir	set-working-directory!
close	close-input-port, close-output-port, close-channel, close-socket
closedir	close-directory-stream
creat	open-file
ctime	time->string
dup	dup, dup-switching-mode
dup2	dup2
exec[l|v][e|p|eps]	exec, exec-with-environment, exec-file, exec-file-with-environment, exec-with-alias
_exit	exit
fcntl	i/o-flags, set-i/o-flags!, close-on-exec?, set-close-on-exec?!
fork	fork, fork-and-forget
fstat	get-port-info
getcwd	working-directory
getegid	get-effective-group-id
getenv	lookup-environment-variable, environment-alist
geteuid	get-effective-user-id
getgid	get-group-id
getgroups	get-login-name
getpid	get-process-id

getppid	get-parent-process-id
getuid	get-user-id
isatty	port-is-a-terminal?
link	link
lstat	get-file/link-info
mkdir	make-directory
mkfifo	make-fifo
open	open-file
opendir	open-directory-stream
pipe	open-pipe
read	read-char, read-block
readdir	read-directory-stream
rename	rename
rmdir	remove-directory
setgid	set-group-id!
setuid	set-user-id!
stat	get-file-info
time	current-time
ttyname	port-terminal-name
umask	set-file-creation-mask!
uname	os-name, os-node-name, os-release-name, os-version-name, machine-name
unlink	unlink
waitpid	wait-for-child-process
write	write-char, write-block

9 Pre-Scheme: A low-level dialect of Scheme

Pre-Scheme [Kelsey 97] is a low-level dialect of Scheme, designed for systems programming with higher-level abstractions. For example, the Scheme48 virtual machine is written in Pre-Scheme. Pre-Scheme is a particularly interesting alternative to C for many systems programming tasks, because not only does it operate at about the same level as C, but it also may be run in a regular high-level Scheme development with no changes to the source, without resorting to low-level stack munging with tools such as gdb. Pre-Scheme also supports two *extremely* important high-level abstractions of Scheme: macros and higher-order, anonymous functions. Richard Kelsey's Pre-Scheme compiler, based on his PhD research on transformational compilation [Kelsey 89], compiles Pre-Scheme to efficient C, applying numerous intermediate source transformations in the process.

This chapter describes details of the differences between Scheme and Pre-Scheme, listings of the default environment and other packages available to Pre-Scheme, the operation of Richard Kelsey's Pre-Scheme compiler, and how to run Pre-Scheme code as if it were Scheme in a regular Scheme environment.

9.1 Differences between Pre-Scheme & Scheme

Pre-Scheme is often considered either a dialect of Scheme or a subset of Scheme. However, there are several very important fundamental differences between the semantics of Pre-Scheme & Scheme to detail.

There is no garbage collector in Pre-Scheme.
: All memory management is manual, as in C, although there are two levels to memory management, for higher- and lower-level purposes: pointers & addresses. Pointers represent higher-level data that are statically checked for type coherency, such as vectors of a certain element type, or strings. Addresses represent direct, low-level memory indices.

Pre-Scheme has no closures.
: `Lambda` expressions that would require full closures at run-time — *e.g.*, those whose values are stored in the heap — are not permitted in Pre-Scheme. However, the Pre-Scheme compiler can hoist many `lambda` expressions to the top level, removing the need of closures for them. (Closures would be much less useful in the absence of garbage collection, in any case.) If the Pre-Scheme compiler is unable to move a `lambda` to a place where it requires no closure, it signals an error to the user.

Tail call optimization is not universal.
: The Pre-Scheme compiler optimizes tail calls where it is possible — typically, just in local loops and top-level procedures that are not exported from the package, but there are other heuristics —, but it is not universal. Programmers may force tail call optimization with Pre-Scheme's `goto` special form (see Section 9.3.2 [Tail call optimization in Pre-Scheme], page 152), but, in situations where the compiler would not have optimized the tail call, this can make the generated code have to jump through many hoops to be a tail call — often necessitating code bloat, because the code of the tail-called procedure is

integrated into the caller's driver loop —; and, where the compiler would have otherwise optimized the tail call, `goto` has no effect anyway.

Types are strictly verified with Hindley-Milner type inference.
The types of Pre-Scheme programs are statically verified based on Hindley-Milner type inference, with some modifications specific to Pre-Scheme. Type information is *not* retained at run-time; any tagging must be performed explicitly.

Pre-Scheme does not support first-class continuations.
There is no `call-with-current-continuation` or other continuation manipulation interface. It has been suggested that downward-only continuations, based on C's `setjmp` & `longjmp`, might be implemented in the future, but this is not yet the case.[1]

The full numeric tower of R5RS is not supported by Pre-Scheme.
Pre-Scheme's only numeric types are fixnums and flonums, with precision determined by the architecture on which the Pre-Scheme code runs. Fixnums are translated to C as the `long` type; flonums are translated as the `float` type.

Top-level Pre-Scheme code is evaluated at compile-time.
Closures actually *are* available, as long as they may be eliminated before run-time. Code evaluated at compile-time also does not require satisfaction of strict static typing. Moreover, certain procedures, such as `vector-length`, are available only at compile-time.

9.2 Type specifiers

Although Pre-Scheme's static type system is based mostly on Hindley-Milner type inference, with as little explicit type information as possible, there are still places where it is necessary to specify types explicitly; for example, see Section 9.3.7 [Pre-Scheme access to C functions and macros], page 156. There are several different kinds of types with different syntax:

`type-name`
Symbols denote either record type or base types. Record types are defined with the `define-record-type` special form described later; the following base types are defined:

`integer` Fixed-size integers (fixnums). This type is translated into C as `long`. The actual size depends on the size of C's `long`, which on most architectures is 32 bits.

`float` Floating-point data. This type translates to C as `double`.

`null` Type which has no value. The `null` type translates to the C `void` type.

`unit` Type which has one value. Actually, this, too, translates to C's `void`, so that it has one value is not strictly true.

[1] It may be possible to use Pre-Scheme's C FFI to manually use `setjmp` & `longjmp`, but the author of this manual cannot attest to this working.

`boolean` Booleans translate to the C `char` type. `#t` is emitted as `TRUE`, and `#f`, as `FALSE`; these are usually the same as `1` & `0`, respectively.

`input-port`
`output-port`
I/O ports. On Unix, since Pre-Scheme uses `stdio`, these are translated to `FILE *`s, `stdio` file streams.

`char` Characters. The size of characters is dependent on the underlying C compiler's implementation of the `char` type.

`address` Simple addresses for use in Pre-Scheme's low-level memory manipulation primitives (see Section 9.4.4 [Low-level Pre-Scheme memory manipulation], page 157); see that section for more details.

`(=> (`*argument-type* `...)` *return-type* `...)`
The types of procedures, known as 'arrow' types.

`(^` *type*`)` The type of pointers that point to *type*. Note that these are distinct from the address type. Pointer types are statically verified to be coherent data, with no defined operations except for accessing offsets in memory from the pointer — *i.e.* operations such as `vector-ref` —; addresses simply index bytes, on which only direct dereferencing, but also arbitrary address arithmetic, is available. Pointers and addresses are *not* interchangeable, and and there is no way to convert between them, as that would break the type safety of Pre-Scheme pointers.

`(tuple` *type* `...)`
Multiple value types, internally used for argument & return types.

9.3 Standard environment

Pre-Scheme programs usually open the `prescheme` structure. There are several other structures built-in to Pre-Scheme as well, described in the next section. This section describes the `prescheme` structure.

9.3.1 Scheme bindings

Bindings for all the names specified here from R5RS Scheme are available in Pre-Scheme. The remainder of the sections after this one detail Pre-Scheme specifics that are not a part of Scheme.

`define` *name value* [syntax]
`define` (*name . argument-list*) *value* [syntax]
`if` *condition consequent* [*alternate*] [syntax]
`let` ((*name expression*) ...) *body* [syntax]
`let*` ((*name expression*) ...) *body* [syntax]
`and` *conjunct ...* [syntax]
`or` *disjunct ...* [syntax]
`cond` *cond-clause ...* [syntax]
`do` ((*name init-exp* [*step-exp*]) ...) (*test-exp* [*return-exp*]) *body* [syntax]
These special forms & macros are all unchanged from their R5RS specifications.

`define-syntax` *name transformer-expression* [*aux-names*] [syntax]
`let-syntax` ((*name transformer-expression*) ...) *body* [syntax]
`letrec-syntax` ((*name transformer-expression*) ...) *body* [syntax]

Pre-Scheme's macro facility is exactly the same as Scheme48's. *Transformer-expression* may be either a `syntax-rules` or an explicit renaming transformer, just as in Scheme48; in the latter case, it is evaluated either in a standard Scheme environment or however the `for-syntax` clause specified of the package in whose code the transformer appeared. For details on the extra *aux-names* operand to `define-syntax`, see Section 4.1.12 [Explicit renaming macros], page 48.

`not` *boolean* ⟶ *boolean* [procedure]
`eq?` $value_a$ $value_b$ ⟶ *boolean* [procedure]
`char=?` $char_a$ $char_b$ ⟶ *boolean* [procedure]
`char<?` $char_a$ $char_b$ ⟶ *boolean* [procedure]
`values` *value* ... ⟶ *values* [procedure]
`call-with-values` *producer consumer* ⟶ *values* [procedure]
`current-input-port` ⟶ *input-port* [procedure]
`current-output-port` ⟶ *output-port* [procedure]

These procedures are all unchanged from their R5RS specifications.

`+` *addend* ... ⟶ *integer* [procedure]
`-` *integer* ⟶ *integer* [procedure]
`-` *minuend subtrahend* ⟶ *integer* [procedure]
`*` *multiplicand* ... ⟶ *integer* [procedure]
`=` $integer_a$ $integer_b$ ⟶ *boolean* [procedure]
`<` $integer_a$ $integer_b$ ⟶ *boolean* [procedure]
`>` $integer_a$ $integer_b$ ⟶ *boolean* [procedure]
`<=` $integer_a$ $integer_b$ ⟶ *boolean* [procedure]
`>=` $integer_a$ $integer_b$ ⟶ *boolean* [procedure]
`min` $integer_1$ $integer_2$ ... ⟶ *integer* [procedure]
`max` $integer_1$ $integer_2$ ... ⟶ *integer* [procedure]
`abs` *integer* ⟶ *integer* [procedure]
`quotient` *divisor dividend* ⟶ *integer* [procedure]
`remainder` *divisor dividend* ⟶ *integer* [procedure]
`expt` *base exponent* ⟶ *integer* [procedure]

These numerical operations are all unchanged from their R5RS counterparts, except that they are applicable only to fixnums, not to flonums, and they always return fixnums.

9.3.2 Tail call optimization

`goto` *procedure argument* ... [syntax]

The Pre-Scheme compiler can be forced to optimize tail calls, even those it would not have otherwise optimized, by use of the `goto` special form, rather than simple procedure calls. In every respect other than tail call optimization, this is equivalent to calling *procedure* with the given arguments. Note, however, that uses of `goto` may cause code to blow up if the Pre-Scheme compiler had reason not to optimize the tail

call were it not for the `goto`: it may need to merge the tail-called procedure into the caller's code.

9.3.3 Bitwise manipulation

Pre-Scheme provides basic bitwise manipulation operators.

`bitwise-and` *integer*$_a$ *integer*$_b$ $\longrightarrow$ *integer* [procedure]
`bitwise-ior` *integer*$_a$ *integer*$_b$ $\longrightarrow$ *integer* [procedure]
`bitwise-xor` *integer*$_a$ *integer*$_b$ $\longrightarrow$ *integer* [procedure]
`bitwise-not` *integer* $\longrightarrow$ *integer* [procedure]

Bitwise boolean logical operations.

`shift-left` *integer count* $\longrightarrow$ *integer* [procedure]
`arithmetic-shift-right` *integer count* $\longrightarrow$ *integer* [procedure]
`logical-shift-right` *integer count* $\longrightarrow$ *integer* [procedure]

Three ways to shift bit strings: `shift-left` shifts *integer* left by *count*, `arithmetic-shift-right` shifts *integer* right by *count* arithmetically, and `logical-shift-right` shifts *integer* right by *count* logically.

9.3.4 Compound data manipulation

Pre-Scheme has somewhat lower-level vector & string facilities than Scheme, with more orientation towards static typing. It also provides a statically typed record facility, which translates to C structs, though not described here, as it is not in the `prescheme` structure; see Section 9.4.2 [Pre-Scheme record types], page 156.

`make-vector` *length init* $\longrightarrow$ *vector* [procedure]
`vector-length` *vector* $\longrightarrow$ *integer* [procedure]
`vector-ref` *vector index* $\longrightarrow$ *value* [procedure]
`vector-set!` *vector index value* $\longrightarrow$ *unit* [procedure]

Vectors in Pre-Scheme are almost the same as vectors in regular Scheme, but with a few differences. `Make-vector` initializes what it returns with null pointers (see below); it uses the *required* (unlike Scheme) *init* argument only to determine the type of the vector: vectors are statically typed; they can contain only values that have the same static type as *init*. `Vector-length` is available only at the top level, where calls to it can be evaluated at compile-time; vectors do not at run-time store their lengths. Vectors must also be explicitly deallocated.

Warning: As in C, there is *no* vector bounds checking at run-time.

`make-string` *length* $\longrightarrow$ *string* [procedure]
`string-length` *string* $\longrightarrow$ *integer* [procedure]
`string-ref` *string index* $\longrightarrow$ *char* [procedure]
`string-set!` *string index char* $\longrightarrow$ *unit* [procedure]

Strings in Pre-Scheme are the nearly same as strings in R5RS Scheme. The only three differences here are that `make-string` accepts exactly one argument, strings must be explicitly deallocated, and strings are `nul`-terminated: `string-length` operates by scanning for the first ASCII `nul` character in a string.

Warning: As in C, there is *no* string bounds checking at run-time.

`deallocate` *pointer* ⟶ *unit* [procedure]
Deallocates the memory pointed to by `pointer`. This is necessary at the end of a string, vector, or record's life, as Pre-Scheme data are not automatically garbage-collected.

`null-pointer` ⟶ *null-pointer* [procedure]
`null-pointer?` *pointer* ⟶ *boolean* [procedure]
`Null-pointer` returns the distinguished null pointer object. It corresponds with `0` in a pointer context or `NULL` in C. `Null-pointer?` returns true if *pointer* is a null pointer, or false if not.

9.3.5 Error handling

Pre-Scheme's method of error handling is similar to the most common one in C: error codes. There is an enumeration `errors` of some error codes commonly and portably encountered in Pre-Scheme.

`errors` [enumeration]

```
(define-enumeration errors
  (no-errors
   parse-error
   file-not-found
   out-of-memory
   invalid-port))
```

Each enumerand has the following meaning:

`(enum errors no-errors)`
Absence of error: success.

`(enum errors parse-error)`
Any kind of parsing error. The Scheme48 VM uses this when someone attempts to resume a malformed suspended heap image.

`(enum errors file-not-found)`
Used when an operation that operates on a file given a string filename found that the file for that filename was absent.

`(enum errors out-of-memory)`
When there is no more memory to allocate.

`(enum errors invalid-port)`
Unused.

`error-string` *error-status* ⟶ *string* [procedure]
Returns a string describing the meaning of the `errors` enumerand *error-status*.

`error` *message irritant* . . . [procedure]
Signals a fatal error with the given message & related irritants and halts the program. On Unix, the program's exit code is -1.

9.3.6 Input & output

Pre-Scheme's I/O facilities are somewhat different from Scheme's, given the low level and the static type strictness. There is no exception mechanism in Pre-Scheme; everything is maintained by returning a status token, as in C. Pre-Scheme's built-in I/O facilities are buffered.[2] (see Section 9.4.4 [Low-level Pre-Scheme memory manipulation], page 157, for two other I/O primitives, `read-block` & `write-block`, for reading & writing blocks of direct memory.)

`open-input-file` *filename* ⟶ [*port status*] [procedure]
`open-output-file` *filename* ⟶ [*port status*] [procedure]
`close-input-port` *input-port* ⟶ *status* [procedure]
`close-output-port` *output-port* ⟶ *status* [procedure]

`Open-input-file` & `open-output-file` open ports for the given filenames. They each return two values: the newly open port and an `errors` enumerand status. Users of these procedures should always check the error status before proceeding to operate with the port. `Close-input-port` & `close-output-port` close their port arguments and return the `errors` enumerand status of the closing.

`read-char` *input-port* ⟶ [*char eof? status*] [procedure]
`peek-char` *input-port* ⟶ [*char eof? status*] [procedure]
`read-integer` *input-port* ⟶ [*integer eof? status*] [procedure]

`Read-char` reads & consumes a single character from its *input-port* argument. `Peek-char` reads, but does not consume, a single character from *input-port*. `Read-integer` parses an integer literal, including sign. All of these also return two other values: whether or not the file is at the end and any `errors` enumerand status. If any error occurred, the first two values returned should be ignored. If *status* is `(enum errors no-errors)`, users of these three procedures should then check *eof?*; it is true if *input-port* was at the end of the file with nothing more left to read and false otherwise. Finally, if both *status* is `(enum errors no-errors)` and *eof?* is false, the first value returned may be safely used.

`write-char` *char output-port* ⟶ *status* [procedure]
`newline` *output-port* ⟶ *status* [procedure]
`write-string` *string output-port* ⟶ *status* [procedure]
`write-integer` *integer output-port* ⟶ *status* [procedure]

These all write particular elements to their *output-port* arguments. `Write-char` writes individual characters. `Newline` writes newlines (line-feed, or ASCII codepoint 10, on Unix). `Write-string` writes the contents of *string*. `Write-integer` writes an ASCII representation of *integer* to port, suitable to be read by `read-integer`. These all return an `errors` enumerand status. If it is `no-errors`, the write succeeded.

`force-output` *output-port* ⟶ *status* [procedure]

Forces all buffered output in *output-port*. *Status* tells whether or not the operation was successful.

[2] Scheme48's VM does not use Pre-Scheme's built-in I/O facilities to implement channels (see Section 4.5.4 [Channels], page 65) — it builds its own lower-level facilities that are still OS-independent, but, because they're written individually for different OSs, they integrate better as low-level I/O channels with the OS. On Unix, the Scheme48 VM uses file descriptors; Pre-Scheme's built-in I/O uses `stdio`. Scheme48's VM uses Pre-Scheme's built-in I/O only to read heap images.

9.3.7 Access to C functions and macros

`external` *c-name ps-type* $\longrightarrow$ *procedure* [syntax]

Special form for accessing C functions & macros. Calls in Pre-Scheme to the resulting procedure are compiled to calls in C to the function or macro named by *c-name*, which should be a string. *PS-type* is the Pre-Scheme type (see Section 9.2 [Pre-Scheme type specifiers], page 150) that the procedure should have, which is necessary for type inference.

9.4 More Pre-Scheme packages

Along with the `prescheme` structure, there are several other structures built-in to Pre-Scheme.

9.4.1 Floating point operation

Since Pre-Scheme's strict static type system would not permit overloading of the arithmetic operators for integers & floats, it provides a different set of operators for floats. These names are all exported by the `ps-flonums` structure.

`fl+` *augend addend ...* $\longrightarrow$ *float* [procedure]
`fl-` *float* $\longrightarrow$ *float* [procedure]
`fl-` *minuend subtrahend* $\longrightarrow$ *float* [procedure]
`fl*` *multiplier multiplicand ...* $\longrightarrow$ *float* [procedure]
`fl/` *divisor dividend* $\longrightarrow$ *float* [procedure]
`fl=` $float_a$ $float_b$ $\longrightarrow$ *boolean* [procedure]
`fl<` $float_a$ $float_b$ $\longrightarrow$ *boolean* [procedure]
`fl>` $float_a$ $float_b$ $\longrightarrow$ *boolean* [procedure]
`fl<=` $float_a$ $float_b$ $\longrightarrow$ *boolean* [procedure]
`fl>=` $float_a$ $float_b$ $\longrightarrow$ *boolean* [procedure]

All of these operations `flop` correspond as floating point variations of their *op* integer equivalents.

9.4.2 Record types

The `ps-record-types` structure defines the following special form for introducing record types. Pre-Scheme record types are translated to C as structs.

`define-record-type` [syntax]

```
(define-record-type type type-descriptor
  (constructor argument-field-tag ...)
  (field-tag1 field-type-spec1
    field-accessor1 [field-modifier1])
  (field-tag2 field-type-spec2
    field-accessor2 [field-modifier2])
  ...
  (field-tagn field-type-specn
    field-accessorn [field-modifiern])
```

Defines a record type. *Type* is mangled to the C struct type name (*type-descriptor-name* is unused unless running Pre-Scheme as Scheme). *Constructor* is defined to

construct a record of the new type and initialize the fields *argument-type-field* ... with its arguments, respectively. If it cannot allocate a sufficient quantity of memory, *constructor* returns a null pointer. The initial values of fields that are not passed to the constructor are undefined. For each field $field_i$ specified,

- $field_i$ is specified to have the type $field\text{-}type\text{-}spec_i$;
- $field\text{-}accessor_i$ is defined to be a procedure of one argument, a record of type *type-name*, that returns the value of the field $field_i$ of that record — its type is defined to be `(=> (`*type-name*`)` $field\text{-}type\text{-}spec_i$`)`; and
- if present, $field\text{-}modifier_i$ is defined to be a procedure of two arguments, a record of type *type-name* and a value of type *field-type-spec*, that assigns the value of the field $field_i$ in its first argument to be the value of its second argument; its type is `(=> (`*type-name field-type-spec*`) unit)`.

Records must be deallocated explicitly when their lifetime has expired with `deallocate`.

9.4.3 Multiple return values

Pre-Scheme support multiple return values, like in Scheme. The only difference is that one cannot operate on multiple return values as lists, since Pre-Scheme does not have lists. Multiple return values are implemented in C as returning in C the first value and passing pointers to the remaining values, which the function returning multiple values assigns. The `prescheme` structure exports the two multiple return value primitives, `call-with-values` and `values`, but the `ps-receive` structure exports this macro for more conveniently binding multiple return values.

`receive` *formals producer body* [syntax]

Binds the `lambda` parameter list *formals* to the multiple values that *producer* returns, and evaluates *body* with the new variables bound.

```
(receive formals
         producer
  body)
    ≡
(call-with-values
    (lambda () producer)
  (lambda formals
    body))
```

9.4.4 Low-level memory manipulation

Pre-Scheme is a low-level language. It provides very low-level, direct memory manipulation. 'Addresses' index a flat store of sequences of bytes. While Pre-Scheme 'pointers' are statically checked for data coherency, allow no arbitrary arithmetic, and in general are high-level abstract data to some extent, addresses are much lower-level, have no statically checked coherency — the values an address represents are selected by what operation used to read or write from it —, permit arbitrary address arithmetic, and are a much more concrete interface into direct memory. The `ps-memory` structure exports these direct memory manipulation primitives.

`allocate-memory` *size* ⟶ *address* [procedure]
`deallocate-memory` *address* ⟶ *unit* [procedure]

`Allocate-memory` reserves a sequence of *size* bytes in the store and returns an address to the first byte in the sequence. `Deallocate-memory` releases the memory at *address*, which should have been the initial address of a contiguous byte sequence, as `allocate-memory` would return, not an offset address from such an initial address.

`unsigned-byte-ref` *address* ⟶ *unsigned-byte* [procedure]
`unsigned-byte-set!` *address unsigned-byte* ⟶ *unit* [procedure]
`word-ref` *address* ⟶ *word* [procedure]
`word-set!` *address word* ⟶ *unit* [procedure]
`flonum-ref` *address* ⟶ *float* [procedure]
`flonum-set!` *address float* ⟶ *unit* [procedure]

Procedures for reading from & storing to memory. `Unsigned-byte-ref` & `unsigned-byte-set!` access & store the first unsigned byte at *address*. `Word-ref` & `word-set!` access & store the first word — Pre-Scheme integer — beginning at *address*. `Flonum-ref` & `flonum-set!` access & store 64-bit floats beginning at *address*..

Bug: `Flonum-ref` & `flonum-set!` are unimplemented in the Pre-Scheme-as-Scheme layer (see Section 9.7 [Running Pre-Scheme as Scheme], page 163).

`address?` *value* ⟶ *boolean* [procedure]

Disjoint type predicate for addresses.

Note: `Address?` is available *only* at the top level, where code is evaluated at compile-time. Do not use this in any place where it may be called at run-time.

`null-address` ⟶ *address* [constant]

The null address. This is somewhat similar to the null pointer, except that it is an address.

Note: One acquires the null *pointer* by calling the *procedure* `null-pointer`, whereas the constant value of the *binding* named `null-address` is the null *address*.

`null-address?` *address* ⟶ *boolean* [procedure]

`Null-address?` returns true if *address* is the null address and false if not.

`address+` *address increment* ⟶ *address* [procedure]
`address-` *address decrement* ⟶ *address* [procedure]
`address-difference` $address_a$ $address_b$ ⟶ *integer* [procedure]

Address arithmetic operators. `Address+` adds *increment* to *address*; `address-` subtracts *decrement* from *address*; and `address-difference` returns the integer difference between $address_a$ and $address_b$. For any $address_p$ & $address_q$, (`address+` $address_p$ (`address-difference` $address_p$ $address_q$)) is equal to $address_q$.

`address=` $address_a$ $address_b$ ⟶ *boolean* [procedure]
`address<` $address_a$ $address_b$ ⟶ *boolean* [procedure]
`address>` $address_a$ $address_b$ ⟶ *boolean* [procedure]
`address<=` $address_a$ $address_b$ ⟶ *boolean* [procedure]
`address>=` $address_a$ $address_b$ ⟶ *boolean* [procedure]

Address comparators.

`integer->address` *integer* ⟶ *address* [procedure]
`address->integer` *address* ⟶ *integer* [procedure]
Integers and addresses, although not the same type, may be converted to and from each other; `integer->address` & `address->integer` perform this conversion. Note that Pre-Scheme *pointers* may not be converted to addresses or integers, and the converse is also true.

`copy-memory!` *source-address target-address count* ⟶ *unit* [procedure]
Copies *count* bytes starting at *source-address* to *target-address*. This is similar to C's `memcpy`.

`memory-equal?` $address_a$ $address_b$ *count* ⟶ *boolean* [procedure]
Compares the two sequences of *count* bytes starting at addresses $address_a$ & $address_b$. It returns true if every byte is equal and false if not.

`char-pointer->string` *address size* ⟶ *string* [procedure]
`char-pointer->nul-terminated-string` *address* ⟶ *string* [procedure]
`Char-pointer->string` returns a string with *size* bytes from the contiguous sequence of bytes starting at *address*. `Char-pointer->nul-terminated-string` does similarly, but it returns a string whose contents include every byte starting at *address* until, but not including, the first 0 byte, *i.e.* ASCII nul character, following *address*.

`read-block` *port address count* ⟶ [*count-read eof? status*] [procedure]
`write-block` *port address count* ⟶ *status* [procedure]
`Read-block` attempts to read *count* bytes from *port* into memory starting at *address*. `Write-block` attempts to write *count* bytes to *port* from the contiguous sequence in memory starting at *address*. `Read-block` returns three values: the number of bytes read, whether or not the read went to the end of the file, and the error status (see Section 9.3.5 [Pre-Scheme error handling], page 154). `Write-block` returns the error status.

9.5 Invoking the Pre-Scheme compiler

Richard Kelsey's Pre-Scheme compiler is a whole-program compiler based on techniques from his research in transformational compilation [Kelsey 89]. It compiles the restricted dialect of Scheme to efficient C, and provides facilities for programmer direction in several optimizations.

9.5.1 Loading the compiler

There is a script, a Scheme48 command program (see Section 2.4.10 [Command programs], page 19), that comes with Scheme48 to load the Pre-Scheme compiler, which is in the file `ps-compiler/load-ps-compiler.scm`. It must be loaded from the `ps-compiler/` directory, from Scheme48's main distribution, into the `exec` package, after having loaded `../scheme/prescheme/interface.scm` & `../scheme/prescheme/package-defs.scm` into the `config` package. The Pre-Scheme compiler takes some time to load, so it may be easier to load it once and dump a heap image of the suspended command processor after having loaded everything; see Section 2.4.11 [Image-building commands], page 20.

To load the Pre-Scheme compiler and dump an image to the file `ps-compiler.image` that contains `prescheme-compiler` in the user package, send this sequence of commands to the command processor while in the `ps-compiler/` directory of Scheme48's distribution:

```
,config ,load ../scheme/prescheme/interface.scm
,config ,load ../scheme/prescheme/package-defs.scm
,exec ,load load-ps-compiler.scm
,in prescheme-compiler prescheme-compiler
,user (define prescheme-compiler ##)
,dump ps-compiler.image "(Pre-Scheme)"
```

9.5.2 Calling the compiler

After having loaded the Pre-Scheme compiler, the `prescheme-compiler` structure is the front end to the compiler that exports the `prescheme-compiler` procedure.

`prescheme-compiler` *structure-spec config-filenames init-name c-filename command ...* [procedure]

Invokes the Pre-Scheme compiler. *Config-filenames* contain module descriptions (see Chapter 3 [Module system], page 23) for the components of the program. *Structure-spec* may be a symbol or a list of symbols, naming the important structure or structures. All structures that it relies/they rely on are traced in the packages' `open` clauses. Modules that are not traced in the dependency graph with root vertices of the given structure[s] are omitted from the output. *C-filename* is a string naming the file to which the C code generated by the Pre-Scheme compiler should be emitted. *Init-name* is the name for an initialization routine, generated automatically by the Pre-Scheme compiler to initialize some top-level variables. The *command* arguments are used to control certain aspects of the compilation. The following commands are defined:

`(copy (`*structure copyable-procedure*`) ...)`
: Specifies that each the body of each *copyable-procedure* from the respective *structure* (from one of *config-filenames*) may be integrated & duplicated.

`(no-copy (`*structure uncopyable-procedure*`) ...)`
: Specifies that the given procedures may not be integrated.

`(shadow ((`*proc-structure procedure*`) (`*var-structure variable-to-shadow*`) ...) ...)`
: Specifies that, in *procedure* from *proc-structure*, the global variables *variable-to-shadow* from their respective *var-structure*s should be shadowed with local variables, which are more likely to be kept in registers for faster operation on them.

`(integrate (`*client-procedure integrable-procedure*`) ...)`
: Forces *integrable-procedure* to be integrated in *client-procedure*.

  Note: The `integrate` command operates on the global program, not on one particular module; each *client-procedure* and *integrable-procedure* is chosen from all variables defined in the entirety of the program, across all modules. It is advised that there be only one of each.

`(header` *`header-line ...`*`)`

Each *header-line* is added to the top of the generated C file, after a cpp inclusion of `<stdio.h>` and `"prescheme.h"`.

The command arguments to `prescheme-compiler` are optional; they are used only to optimize the compiled program at the programmer's request.

9.6 Example Pre-Scheme compiler usage

The `ps-compiler/compile-vm.scm`, `ps-compiler/compile-gc.scm`, and `ps-compiler/compile-vm-no-gc.scm` files give examples of running the Pre-Scheme compiler. They are Scheme48 command programs (see Section 2.4.10 [Command programs], page 19), to be loaded into the `exec` package after having already loaded the Pre-Scheme compiler. `compile-vm.scm` & `compile-vm-no-gc.scm` generate a new `scheme48vm.c` in the `scheme/vm/` directory — `compile-vm.scm` includes the garbage collector, while `compile-vm-no-gc.scm` does not[3] —, and `compile-gc.scm` generates a new `scheme48heap.c`, `scheme48read-image.c`, & `scheme48write-image.c` in the `scheme/vm/` directory.

Here is a somewhat simpler example. It assumes a pre-built image with the Pre-Scheme compiler loaded is in the `ps-compiler.image` file in the current directory (see Section 9.5 [Invoking the Pre-Scheme compiler], page 159, where there is a description of how to dump an image with the Pre-Scheme compiler loaded).

```
% ls
hello.scm               packages.scm            ps-compiler.image
% cat hello.scm
(define (main argc argv)
  (if (= argc 2)
      (let ((out (current-output-port)))
        (write-string "Hello, world, " out)
        (write-string (vector-ref argv 1) out)
        (write-char #\! out)
        (newline out)
        0)
      (let ((out (current-error-port)))
        (write-string "Usage: " out)
        (write-string (vector-ref argv 0) out)
        (write-string " <user>" out)
        (newline out)
        (write-string "  Greets the world & <user>." out)
        (newline out)
        -1)))
% cat packages.scm
(define-structure hello (export main)
  (open prescheme)
  (files hello))
% scheme48 -i ps-compiler.image
```

[3] The actual distribution of Scheme48 separates the garbage collector and the main virtual machine.

```
heap size 3000000 is too small, using 4770088
Welcome to Scheme 48 1.3 (Pre-Scheme)
Copyright (c) 1993-2005 by Richard Kelsey and Jonathan Rees.
Please report bugs to scheme-48-bugs@s48.org.
Get more information at http://www.s48.org/.
Type ,? (comma question-mark) for help.
> (prescheme-compiler 'hello '("packages.scm") 'hello-init "hello.c")
packages.scm
 hello.scmChecking types
 main : ((integer **char) -> integer)
In-lining single-use procedures
Call Graph:
<procedure name>
  <called non-tail-recursively>
  <called tail-recursively>
main  (exported)
Merging forms
Translating
 main
#{Unspecific}
> ,exit
% cat hello.c
#include <stdio.h>
#include "prescheme.h"

long main(long, char**);


long main(long argc_0X, char **argv_1X)
{
  FILE * out_3X;
  FILE * out_2X;
 {  if ((1 == argc_0X)) {
    out_2X = stdout;
    ps_write_string("Hello, world, ", out_2X);
    ps_write_string((*(argv_1X + 1)), out_2X);
    { long ignoreXX;
    PS_WRITE_CHAR(33, out_2X, ignoreXX) }
    { long ignoreXX;
    PS_WRITE_CHAR(10, out_2X, ignoreXX) }
    return 0;}
  else {
    out_3X = stderr;
    ps_write_string("Usage: ", out_3X);
    ps_write_string((*(argv_1X + 0)), out_3X);
    ps_write_string(" <user>", out_3X);
    { long ignoreXX;
```

```
        PS_WRITE_CHAR(10, out_3X, ignoreXX) }
        ps_write_string("  Greets the world & <user>.", out_3X);
        { long ignoreXX;
        PS_WRITE_CHAR(10, out_3X, ignoreXX) }
        return -1;}}
    }
    %
```

9.7 Running Pre-Scheme as Scheme

To facilitate the operation of Pre-Scheme systems within a high-level Scheme development environment, Scheme48 simply defines the `prescheme`, `ps-memory`, `ps-record-types`, `ps-flonums`, and `ps-receive` structures in terms of Scheme; Pre-Scheme structures can be loaded as regular Scheme structures because of this. Those structures and the interfaces they implement are defined in the files `scheme/prescheme/interface.scm` and `scheme/prescheme/package-defs.scm` from the main Scheme48 distribution; simply load these files into the config package (see Section 2.4.6 [Module commands], page 15) before loading any Pre-Scheme configuration files.

The Pre-Scheme emulation layer in Scheme has some shortcomings:

- No more than sixteen megabytes can be allocated at once.
- More than thirty-two or sixty-four or so allocations result in addresses overflowing bignums, which deallocations does not affect.
- Flonum memory access is unimplemented. (Flonum arithmetic works, though.)
- The layer is very slow.

References

[Cejtin *et al.* 95]
Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-Order Distributed Objects. In *ACM Transactions on Programming Languages and Systems*, vol. 17, pp. 704–739, ACM Press, September 1995.

[Clinger 91]
William D. Clinger. Hygienic Macros through Explicit Renaming. In *Lisp Pointers*, IV(4): 25-28, December 1991.

[Donald 92]
Bruce Donald and Jonathan A. Rees. Program Mobile Robots in Scheme! In *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, 2681-2688.

[Friedman 00]
Daniel Friedman and Erik Hilsdale. *Writing Macros in Continuation-Passing Style*. Worksop on Scheme and Functional Programming, September 2000.

[Kelsey 89]
Richard Kelsey. *Compilation by Program Transformation*. PhD thesis, Yale University, 1989.

[Kelsey 97]
Richard Kelsey. *Pre-Scheme: A Scheme Dialect for Systems Programming*. June 1997.

[Museme] Franklyn Turbak and Dan Winship. *Museme: a multi-user simulation environment for Scheme*.
`http://www.bloodandcoffee.net/campbell/code/museme.tar.gz`

[Rees 96] Jonathan A. Rees. *A Security Kernel based on the Lambda-Calculus*. PhD thesis, AI Memo 1564, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1996.

[Reppy 99]
John Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[Shivers 94]
Olin Shivers. *A Scheme Shell*. Tech Report 635, Massachusetts Institute of Technology, Laboratory for Computer Science, 1994.

[Shivers 96]
Olin Shivers. *A Universal Scripting Framework, or Lambda: the Ultimate "Little Language". Concurrency and Parallelism, Programming, Networking, and Security*, pp. 254-265, 1996, Joxan Jaffar and Roland H. C. Yap (eds).

[Shivers *et al.* 04]
Olin Shivers, Brian D. Carlstrom, Martin Gasbichler, and Michael Sperber. *Scsh Reference Manual*, for scsh release 0.6.6
`http://www.scsh.net/docu/docu.html`

[SRFI 1] Olin Shivers. *SRFI 1: List Library* Scheme Requests for Implementation, 1999.
`http://srfi.schemers.org/srfi-1/`

[SRFI 7] Richard Kelsey. *SRFI 7: Feature-Based Program Configuration Language* Scheme Requests for Implementation, 1999.
`http://srfi.schemers.org/srfi-7/`

[SRFI 9] Richard Kelsey. *SRFI 9: Defining Record Types* Scheme Requests for Implementation, 1999.
`http://srfi.schemers.org/srfi-9/`

[SRFI 22] Martin Gasbichler and Michael Sperber *SRFI 22: Running Scheme Scripts on Unix* Scheme Requests for Implementation, 2002.
`http://srfi.schemers.org/srfi-22/`

[SRFI 34] Richard Kelsey and Michael Sperber. *SRFI 34: Exception Handling for Programs.* Scheme Requests for Implementation, 2002.
`http://srfi.schemers.org/srfi-34/`

[SRFI 35] Richard Kelsey and Michael Sperber. *SRFI 35: Conditions.* Scheme Requests for Implementation, 2002.
`http://srfi.schemers.org/srfi-35/`

Concept index

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Binding index

&

*

+

,

-

:

<

=

>

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Structure index

R

S

T

U

W