The Incomplete

Scheme 48 Reference Manual

for release 1.9.2

Richard Kelsey Jonathan Rees Mike Sperber
Marcus Crestani Robert Ransom Roderic Morris
Marcel Turino Martin Gasbichler

A line may take us hours, yet if it does not seem a moment's thought
All our stitching and unstitching has been as nought.

Yeats
Adam's Curse

Acknowledgements

Thanks to Scheme 48's users for their suggestions, bug reports, and forbearance. Thanks also to Deborah Tatar for providing the Yeats quotation.

Contents

Chapter 1

Introduction

Scheme 48 is an implementation of the Scheme programming language as described in the Revised^5 Report on the Algorithmic Language Scheme [6]. It is based on a compiler and interpreter for a virtual Scheme machine. Scheme 48 tries to be faithful to the Revised^5 Scheme Report, providing neither more nor less in the initial user environment. (This is not to say that more isn't available in other environments; see below.)

Scheme 48 is under continual development. Please report bugs, especially in the VM, especially core dumps, to scheme-48-bugs@s48.org. Include the version number x.yy from the "Welcome to Scheme 48 x.yy" greeting message in your bug report. It is a goal of this project to produce a bullet-proof system; we want no bugs and, especially, no crashes. (There are a few known bugs, listed in the `doc/todo.txt` file that comes with the distribution.)

Send a message to scheme-48-request@s48.org with subject 'subscribe' to be put on a mailing list for announcements, discussion, bug reports, and bug fixes.

The name 'Scheme 48' commemorates our having written the original version in forty-eight hours, on August 6th and 7th, 1986.

Chapter 2

User's guide

This chapter details Scheme 48's user interface: its command-line arguments, command processor, debugger, and so forth.

2.1 Command line arguments

A few command line arguments are processed by Scheme 48 as it starts up.

`scheme48` [`-i` *image*] [`-h` *heapsize*] [`-a` *argument* ...]

`-i` ***image*** specifies a heap image file to resume. This defaults to a heap image that runs a Scheme command processor. Heap images are created by the `,dump` and `,build commands`, for which see below.

`-h` ***heapsize*** specifies how much space should be reserved for allocation. *Heapsize* is in words (where one word = 4 bytes), and covers both semispaces, only one of which is in use at any given time (except during garbage collection). Cons cells are currently 3 words, so if you want to make sure you can allocate a million cons cells, you should specify `-h 6000000` (actually somewhat more than this, to account for the initial heap image and breathing room). The default heap size is 3000000 words. The system will use a larger heap if the specified (or default) size is less than the size of the image being resumed.

`-a` ***argument ...*** is only useful with images built using `,build`. The arguments are passed as a list of OS strings (see section 5.15) to the procedure specified in the `,build` command. For example:

```
> ,open os-strings
> (define (f xs)
    (write (map os-string->string xs))
    (newline)
    0)                                        ;must return an integer
```

```
> ,build f foo.image
> ,exit
% scheme48vm -i foo.image -a mumble "foo x" -h 5000000
("mumble" "foo x" "-h" "5000000")
%
```

`-I` ***image argument ...*** is equivalent to `-i` *image* `-a` *argument* .... On most Unix-like systems, a heap image can be made executable with the following Bourne shell commands:

```
% (echo '#!/s48/install/prefix/lib/scheme48-1.9.2/scheme48vm -I'
   cat original.image) >new.image
% chmod +x new.image
```

The usual definition of the `s48` or `scheme48` command is actually a shell script that starts up the Scheme 48 virtual machine with a `-i` *imagefile* specifying the development environment heap image and a `-o` *vm-executable* specifying the location of the virtual-machine executable (the executable is needed for loading external code on some versions of Unix; see section 8.4 for more information). The file `go` in the Scheme 48 installation source directory is an example of such a shell script.

2.2 Command processor

When you invoke the default heap image, a command processor starts running. The command processor acts as both a read-eval-print loop, reading expressions, evaluating them, and printing the results, and as an interactive debugger and data inspector. See Chapter 3 for a description of the command processor.

2.3 Editing

We recommend running Scheme 48 under GNU Emacs or XEmacs using the `cmuscheme48` command package. This is in the Scheme 48 distribution's `emacs/` subdirectory and is included in XEmacs's `scheme` package. It is a variant of the `cmuscheme` library, which comes to us courtesy of Olin Shivers, formerly of CMU. You might want to put the following in your Emacs init file (`.emacs`):

```
(setq scheme-program-name "scheme48")
(autoload 'run-scheme
          "cmuscheme48"
          "Run an inferior Scheme process."
          t)
```

The Emacs function `run-scheme` can then be used to start a process running the program `scheme48` in a new buffer. To make the `autoload` and `(require ...)` forms work, you will also need to put the directory containing `cmuscheme` and related files in your emacs load-path:

```
(setq load-path
  (append load-path '("scheme-48-directory/emacs")))
```

Further documentation can be found in the files `emacs/cmuscheme48.el` and `emacs/comint.el`.

2.4 Performance

If you want to generally have your code run faster than it normally would, enter `inline-values` mode before loading anything. Otherwise calls to primitives (like `+` and `cons`) and in-line procedures (like `not` and `cadr`) won't be open-coded, and programs will run more slowly.

The system doesn't start in `inline-values` mode by default because the Scheme report permits redefinitions of built-in procedures. With this mode set, such redefinitions don't work according to the report, because previously compiled calls may have in-lined the old definition, leaving no opportunity to call the new definition.

`Inline-values` mode is controlled by the `inline-values` switch. `,set inline-values` and `,unset inline-values` turn it on and off.

2.5 Disassembler

The `,dis` command prints out the disassembled byte codes of a procedure.

```
> ,dis cons
cons
  0 (protocol 2)
  2 (pop)
  3 (make-stored-object 2 pair)
  6 (return)
>
```

The current byte codes are listed in the file `scheme/vm/interp/arch.scm`. A somewhat out-of-date description of them can be found in [5].

The command argument is optional; if unsupplied it defaults to the current focus object (`##`).

The disassembler can also be invoked on continuations and templates.

2.6 Module system

This section gives a brief description of modules and related entities. For detailed information, including a description of the module configuration language, see chapter 4.

A *module* is an isolated namespace, with visibility of bindings controlled by module descriptions written in a special configuration language. A module may be instantiated as a *package*, which is an environment in which code can be evaluated. Most modules are instantiated only once and so have a unique package. A *structure* is a subset of the bindings in a package. Only by being included in a structure can a binding be made visible in other packages. A structure has two parts, the package whose bindings are being exported and the set of names that are to be exported. This set of names is called an *interface*. A module then has three parts:

- a set of structures whose bindings are to be visible within the module
- the source code to be evaluated within the module
- a set of exported interfaces

Instantiating a module produces a package and a set of structures, one for each of the exported interfaces.

The following example uses `define-structure` to create a module that implements simple cells as pairs, instantiates this module, and binds the resulting structure to `cells`. The syntax `(export` *name* ...`)` creates an interface containing *name* .... The `open` clause lists structures whose bindings are visible within the module. The `begin` clause contains source code.

```
(define-structure cells (export make-cell
                                cell-ref
                                cell-set!)
  (open scheme)
  (begin (define (make-cell x)
           (cons 'cell x))
         (define cell-ref cdr)
         (define cell-set! set-cdr!)))
```

Cells could also have been implemented using the record facility described in section 5.9 and available in structure `define-record-type`.

```
(define-structure cells (export make-cell
                                cell-ref
                                cell-set!)
  (open scheme define-record-types)
  (begin (define-record-type cell :cell
           (make-cell value)
           cell?
           (value cell-ref cell-set!))))
```

With either definition the resulting structure can be used in other modules by including `cells` in an `open` clause.

The command interpreter is always operating within a particular package. Initially this is a package in which only the standard Scheme bindings are visible. The bindings of other structures can be made visible by using the `,open` command described in section 3.4 below.

Note that this initial package does not include the configuration language. Module code needs to be evaluated in the configuration package, which can be done by using the ,config command:

```
> ,config (define-structure cells ...)
> ,open cells
> (make-cell 4)
'(cell . 4)
> (define c (make-cell 4))
> (cell-ref c)
4
```

2.7 Library

A number of useful utilities are either built in to Scheme 48 or can be loaded from an external library. These utilities are not visible in the user environment by default, but can be made available with the `open` command. For example, to use the `tables` structure, do

```
> ,open tables
>
```

If the utility is not already loaded, then the `,open` command will load it. Or, you can load something explicitly (without opening it) using the `load-package` command:

```
> ,load-package queues
> ,open queues
```

When loading a utility, the message "Note: optional optimizer not invoked" is innocuous. Feel free to ignore it.

See also the package system documentation, in chapter 4.

Not all of the the libraries available in Scheme 48 are described in this manual. All are listed in files `rts-packages.scm`, `comp-packages.scm`, `env-packages.scm`, and `more-packages.scm` in the `scheme` directory of the distribution, and the bindings they export are listed in `interfaces.scm` and `more-interfaces.scm` in the same directory.

Chapter 3

Command processor

This chapter details Scheme 48's command processor, which incorporates both a read-eval-print loop and an interactive debugger. At the `>` prompt, you can type either a Scheme form (expression or definition) or a command beginning with a comma. In inspection mode (see section 3.8) the prompt changes to `:` and commands no longer need to be preceded by a comma; input beginning with a letter or digit is assumed to be a command, not an expression. In inspection mode the command processor prints out a menu of selectable components for the current object of interest.

3.1 Current focus value and

The command processor keeps track of a current *focus value*. This value is normally the last value returned by a command. If a command returns multiple values the focus object is a list of the values. The focus value is not changed if a command returns no values or a distinguished 'unspecific' value. Examples of forms that return this unspecific value are definitions, uses of `set!`, and `(if #f 0)`. It prints as `#{Unspecific}`.

The reader used by the command processor reads `##` as a special expression that evaluates to the current focus object.

```
> (list 'a 'b)
'(a b)
> (car ##)
'a
> (symbol->string ##)
"a"
> (if #f 0)
#{Unspecific}
> ##
"a"
>
```

3.2 Command levels

If an error, keyboard interrupt, or other breakpoint occurs, or the `,push` command is used, the command processor invokes a recursive copy of itself, preserving the dynamic state of the program when the breakpoint occurred. The recursive invocation creates a new *command level.* The command levels form a stack with the current level at the top. The command prompt indicates the number of stopped levels below the current one: `>` or `:` for the base level and *n*`>` or *n*`:` for all other levels, where *n* is the command-level nesting depth. The `levels` setting described below can be used to disable the automatic pushing of new levels.

The command processor's evaluation package and the value of the current focus value are local to each command level. They are preserved when a new level is pushed and restored when it is discarded. The settings of all other settings are shared by all command levels.

⟨eof⟩
: Discards the current command level and resumes running the level down. ⟨eof⟩ is usually control-`D` at a Unix shell or control-`C` control-`D` using the Emacs `cmuscheme48` library.

`,pop`
: The same as ⟨eof⟩.

`,proceed [`*exp* `...]`
: Proceed after an interrupt or error, resuming the next command level down, delivering the values of *exp ...* to the continuation. Interrupt continuations discard any returned values. `,Pop` and `,proceed` have the same effect after an interrupt but behave differently after errors. `,Proceed` restarts the erroneous computation from the point where the error occurred (although not all errors are proceedable) while `,pop` (and ⟨eof⟩) discards it and prompts for a new command.

`,push`
: Pushes a new command level on above the current one. This is useful if the `levels` setting has been used to disable the automatic pushing of new levels for errors and interrupts.

`,reset [`*number*`]`
: Pops down to a given level and restarts that level. *Number* defaults to zero, `,reset` restarts the command processor, discarding all existing levels.

Whenever moving to an existing level, either by sending an ⟨eof⟩ or by using `,reset` or the other commands listed above, the command processor runs all of the `dynamic-wind` "after" thunks belonging to stopped computations on the discarded level(s).

3.3 Logistical commands

`,load` *filename* ...
: Loads the named Scheme source file(s). Easier to type than `(load "`*filename*`")` because you don't have to shift to type the parentheses or quote marks. (However, it is still possible to specify a filename as a Scheme string literal, with quote marks—you'll need this for filenames containing whitespace.) Also, it works in any package, unlike `(load "`*filename*`")`, which will work only in packages in which the variable `load` is defined appropriately.

`,exit [`*exp*`]`
: Exits back out to shell (or executive or whatever invoked Scheme 48 in the first place). *Exp* should evaluate to an integer. The integer is returned to the calling program. The default value of *exp* is zero, which, on Unix, is generally interpreted as success.

3.4 Module commands

There are many commands related to modules. Only the most commonly used module commands are described here; documentation for the rest can be found in section 4.8. There is also a brief description of modules, structures, and packages in section 2.6 below.

`,open` *structure* ...
: Makes the bindings in the *structure*s visible in the current package. The packages associated with the *structure*s will be loaded if this has not already been done (the `ask-before-loading` setting can be used to disable the automatic loading of packages).

`,config [`*command*`]`
: Executes *command* in the `config` package, which includes the module configuration language. For example, use

```
,config ,load filename
```

to load a file containing module definitions. If no *command* is given, the `config` package becomes the execution package for future commands.

`,user [`*command*`]`
: This is similar to the `,config`. It moves to or executes a command in the user package (which is the default package when the Scheme 48 command processor starts).

3.5 Debugging commands

`,preview`
: Somewhat like a backtrace, but because of tail recursion you see less than

you might in debuggers for some other languages. The stack to display is chosen as follows:

1. If the current focus object is a continuation or a thread, then that continuation or thread's stack is displayed.
2. Otherwise, if the current command level was initiated because of a breakpoint in the next level down, then the stack at that breakpoint is displayed.
3. Otherwise, there is no stack to display and a message is printed to that effect.

One line is printed out for each continuation on the chosen stack, going from top to bottom.

`,run` *exp*
: Evaluate *exp*, printing the result(s) and making them (or a list of them, if *exp* returns multiple results) the new focus object. The `,run` command is useful in inspection mode (see section 3.8 below) and when writing command programs (see section 3.9 below).

`,trace` *name* `...`
: Start tracing calls to the named procedure or procedures. With no arguments, displays all procedures currently traced. This affects the binding of *name*, not the behavior of the procedure that is its current value. *Name* is redefined to be a procedure that prints a message, calls the original value of *name*, prints another message, and finally passes along the value(s) returned by the original procedure.

`,untrace` *name* `...`
: Stop tracing calls to the named procedure or procedures. With no argument, stop tracing all calls to all procedures.

`,condition`
: The `,condition` command displays the condition object describing the error or interrupt that initiated the current command level. The condition object becomes the current focus value. This is particularly useful in conjunction with the inspector. For example, if a procedure is passed the wrong number of arguments, do `,condition` followed by `,inspect` to inspect the procedure and its arguments.

`,bound?` *name*
: Display the binding of *name*, if there is one, and otherwise prints '`Not bound`'.

`,expand` *form* Show macro expansion of *form*, if any, fully expanding all macros in *form*.

`,where` *procedure*
: Display name of file containing *procedure*'s source code.

3.6 Profiling commands

After loading the `profile-commands` structure via

```
,load-package profile-commands
```

Scheme 48 provides a basic profiler, with support for flat and call-graph profiles. In addition to call- and runtimes the heap memory usage is estimated as well. By adding the byte-code optimizer `profiler-instrumentation` to the definition of a structure via a clause

```
  (optimize profiler-instrumentation)
```

in the structure definition, code will by instrumented with calls to the profiler. The call-times to instrumented code can then be measured exactly. Without instrumentation, the profiler is solely based on sampling and therefore has only limited accuracy. Any evaluation can be profiled with the following command:

`,profile` *command*
: Evaluate *command* under the profiler, printing the profiling result and making the return value(s) of *command* the new focus object.

The output of the profiler is explained on the basis of the following piece of senseless code:

```
(define (c x)
  (if (= x 0)
      0
      (+ 1 (c (- x 1)))))

(define (b x)
  (let ((y (- x 1)))
    (if (> y 0)
        (begin
          (c y)
          (a y)
          (+ 1 (a y)))
        0)))

(define (a x)
  (let ((y (- x 1)))
    (if (> y 0)
        (begin
          (c y)
          (b y)
          (+ 1 (b y)))
        0)))

(define (main x)
  (+ 1 (a x)))
```

The profiler produces the following output:

```
> ,profile (main 21)

** Samples:        60 (approx. one per 40ms)
** Interrupt time: 50ms
** Real run time:  2430ms
** Total memory:   22k
** GC runs:        0

** Flat result (times in ms):

   time  cumu  self   mem  calls  ms/call  name
 61.66%  1498  1498    2k  37+51       17  c in "x.scm"
  25.0%  2430   608   10k    436        6  b in "x.scm"
 13.33%  2430   324   10k    442        5  a in "x.scm"
   0.0%  2430     0    0k      1     2430  main in "x.scm"

** Tree result (times in ms):

i    time self child  mem    calls  name
0  100.0%    0  2430   0k        0  <profiler> [0]
             0  2430   0k      1/1    main in "x.scm" [1]
=========================================================
                           441/442    b in "x.scm" <cycle 0> [3]
           324   891  10k    1/442    main in "x.scm" [1]
1  100.0%  324   891  10k      442  a in "x.scm" <cycle 0> [2]
           891     0   1k    22/37    c in "x.scm" [4]
                           436/436    b in "x.scm" <cycle 0> [3]
=========================================================
                           436/436    a in "x.scm" <cycle 0> [2]
2  100.0%  608   608  10k      436  b in "x.scm" <cycle 0> [3]
           608     0   1k    15/37    c in "x.scm" [4]
                           441/442    a in "x.scm" <cycle 0> [2]
=========================================================
             0  2430   0k      1/1    <profiler> [0]
3  100.0%    0  2430   0k        1  main in "x.scm" [1]
           324   891  10k    1/442    a in "x.scm" <cycle 0> [2]
=========================================================
           608     0   1k    15/37    b in "x.scm" <cycle 0> [3]
           891     0   1k    22/37    a in "x.scm" <cycle 0> [2]
4  61.66% 1498     0   2k    37+51  c in "x.scm" [4]
=========================================================
           932  1498  20k      1/1    main in "x.scm" [1]
0  100.0%  932  1498  20k    1+877  <cycle 0 as a whole>
           324   891  10k      441    a in "x.scm" <cycle 0> [2]
           608   608  10k      436     b in "x.scm" <cycle 0> [3]
             0     0   2k    37/37     c in "x.scm" [4]
```

The formatting of the output is based on the output of GNU gprof [1].
First, general information about the profiling is shown:

- `samples`: total number of samples taken and the average time between two samples
- `interrupt time`: theoretical interrupt time, the time between two samples
- `real run time`: the total run time of the evaluation, simple measurement from the beginning to the end of the evaluation
- `total memory`: the total amount of memory used by the program, here in kilobytes
- `GC runs`: the number of times the garbage collector was running while evaluating

The flat profile gives overall statistics for each procedure in the program:

- `time`: percentage of time the procedure has in the program (based on "`self`")
- `cumu`: total cumulative time the procedure was running
- `self`: total time the procedure was running itself (without children)
- `mem`: approximated memory usage of the procedure
- `calls`: the number of non-recursive calls to the procedure. Recursive calls are displayed after an optionally appended "+" to former number (see procedure `c` in example output above).
- `ms/call`: estimate of the time per call (with children), based on non-recursive calls and "`cumu`".
- `name`: the name of the procedure and the file it is defined in

The tree result displays profiling information depending on the caller of a particular procedure. As in gprof output, for each procedure there is an *entry*. Entries are separated by lines of `=`s. Each entry displays the callers (indented), the procedure itself and the procedure called (indented). The meaning of the fields depends on the type of the entry, for example:

[1]part of the GNU Binary Utilities: http://www.gnu.org/software/binutils/

```
i    time self  child  mem    calls   name
           324    891  10k    1/442      main [1]
1  100.0%  324    891  10k      442   a <cycle 0> [2]
           891      0   1k    22/37      c [4]
```

Every procedure has a unique index number `i` (here "1"), based on the position in the call-graph. It is used to quickly find the corresponding entry. The number is appended to the procedure name (in brackets). The meaning of the "cycle" suffix is explained below.

Callers (here `main`):

- `self`: time spent directly in `a` when called by `main`
- `child`: time spent in children of `a` when called by `main`
- `mem`: memory usage of `a` when called by `main`
- `calls`: two numbers: number of calls to `a` from `main` and total number of non-recursive calls to `a`

Primary line (here `a`):

- `time`: total percentage of time `a` was running
- `self`: total time spent directly in `a`
- `child`: total time spent in children of `a`
- `mem`: total memory usage of `a`
- `calls`: the total number of non-recursive calls to `a`. Recursive calls are displayed after an optional `+`.

Called (here `c`):

- `self`: time spent directly in `c` when called by `a`
- `child`: time spent in children of `c` when called by `a`
- `mem`: memory usage of `c` when called by `a`
- `calls`: two numbers: number of calls to `c` from `a` and total number of non-recursive calls to `c`

A special treatment is needed for mutual-recursive procedures. In the example, `a` calls `b` and `b` calls `a`. This is called a *cycle*. If `a` would call another procedure `d` and `d` would call `a`, all three procedures form a cycle.

Cycles have a separate entry in the call graph. This entry consists of the callers into the cycle, the primary line of the cycle, the member procedures of the cycle and the external procedures called by the cycle:

```
              932   1498  20k       1/1      main [1]
 0  100.0%    932   1498  20k     1+877   <cycle 0 as a whole>
              324    891  10k       441      a <cycle 0> [2]
              608    608  10k       436      b <cycle 0> [3]
                0      0   2k     37/37      c [4]
```

Callers (here `main`):

- the meaning of the fields are the same as above. The member procedures of the cycle are seen as a whole.

Primary line:

- `time`: total percentage of time any procedure of the cycle was running
- `self`: total time spent directly in a procedure of the cycle
- `child`: total time spent in external children of `a`
- `mem`: sum of memory usage of all cycle member procedures
- `calls`: the total number of external calls to the `cycle` and the total number of calls in the cycle internally

Member procedures:

- the meaning of the fields are the same as above, except…
- `calls`: the number of calls to the member procedure from within the cycle

External procedures (here `c`):

- the meaning of the fields are the same as with the member procedures, except…
- `calls`: the number of calls to the external procedure from the cycle and the total number of non-recursive calls to the external procedure

In the current implementation, there are some issues that need to be considered: in the default configuration, samples are taken every 50 milliseconds. Procedures with a by-call run time shorter than the interrupt time are likely to be profiled inaccurately or may not be seen at all, if they have not been instrumented.

The default interrupt time can be set with the `profiler-interrupt-time` command processor setting, see 3.7. For example,

```
,set profiler-interrupt-time 200
```

sets the default profiler sampling interrupt time to 200ms.

Because the interrupt is shared with the thread system, profiling may cause performance issues with multi-threaded programs. Also, programs with deep recursion (resulting in large continuations) can cause the profiler to use a significant percentage of the total run time.

By default, call times are measured by the sampling process. When code had been instrumented, call times of those procedures will be measured exactly. When only instrumented code should be considered in the output of the profiler, the command processor setting `profiler-measure-noninstr` can be set to `off`.

Scheme 48 optimizes tail calls. Thus, the profiler cannot "see" tail calls and in some situations, the output of the profiler may show that procedures directly call their "grandchildren".

3.6.1 How the profiler works

The profiler is based on taking samples of the current stack of continuations, the *call-stack*. Therefore it schedules the alarm interrupt at a regular interval specified by the `profiler-interrupt-time` command processor setting. Because the thread system uses the alarm interrupt as well, the profiler interrupt handler calls the handler of the thread system after processing the sample.

The profiler interrupt handler first captures the current continuation and follows the continuation-stack down to the continuation of the `profile-thunk` procedure, which called the thunk to be profiled. Each continuation with the corresponding code template is stored in a stack of `stackentry` records.

This `*cur-stack*` is compared from bottom to top with the `*last-stack*`, the stack captured while the prior sampling interrupt. The statistics gathered are stored in `profinfo` records.

For each stack-entry in the `*cur-stack*`, several possibilities exist in conjunction to the entry at the same depth in the `*last-stack*`:

- there is no such entry: the continuation must be the result of a new call
- both entries are the same continuation: nothing changed, the procedure is still running
- entries are not the same continuation: the following entries up in the stack must have changed. If the templates of the current entries are the same, the continuation counts as the still same call to the procedure (but at another expression in the procedure). A such situation only counts as a new call if the continuation object has changed, but the continuation program-counter, arguments and the template stayed the same.

As we ascend in the call-stack, changes in the lower levels indicate that all entries above have to be new calls. This way, the profiler gathers information about how often a template is called by another template.

Additionally, every time a procedure is seen, we record this "occurrence". If the procedure is a the top of the stack, it is currently running. Based on these numbers, the average interrupt time and the number of calls, we can estimate the self and child times.

While running, the profiler collects information about heap memory usage as well. After comparing the current with the last stack, the profiler knows which procedures finished, which procedures are new and which procedure called the new procedures. This information is used to distribute the difference in heap memory usage between the prior and the current sample. The profiler uses the `memory-status` primitive for retrieving available heap space and the number of garbage collector runs (`gc-count`). Difference in used memory between two samples is distributed by the following principles:

- if `gc-count` increased, distribution cannot be done
- if the same continuation is still at top, credit its template with all used memory between the two samplings
- if there are stackentry-templates gone or new, distribute the used memory *equally* between them:
  - gone templates could have used some memory before "returning"
  - new templates could have used some memory while running
  - the caller of new templates could have used some memory between the calling of gone and new procedures

The byte-code optimizer `profiler-instrumentation` adds a call to the "profile-count" procedure of the profiler structure. This procedure is responsible for the exact call-times measurement.

After collection of the data, procedures are numbered and cycles are detected by ascending in the call tree. Time is being propagated from top to bottom, remembering self and child times. Most of the time, cycles are being considered as one procedure.

3.7 Settings

There are a number of settings that control the behavior of the command processor; most of them are booleans. They can be set using the `,set` and `,unset` commands.

`,set` *setting* `[on | off | literal | ?]`
: This sets the value of setting *setting*. For a boolean setting, the second argument must be `on` or `off`; it then defaults to `on`. Otherwise, the value must be a literal, typically a positive number. If the second argument is `?` the value of *setting* is is displayed and not changed. Doing `,set ?` will display a list of the setting and their current values.

`,unset` *setting*
: `,unset` *setting* is the same as `,set` *setting* `off`.

The settings are as follows:

`batch` (boolean)
: In 'batch mode' any error or interrupt that comes up will cause Scheme 48 to exit immediately with a non-zero exit status. Also, the command processor doesn't print prompts. Batch mode is off by default.

`levels` (boolean)
: Enables or disables the automatic pushing of a new command level when an error, interrupt, or other breakpoint occurs. When enabled (the default), breakpoints push a new command level, and ⟨eof⟩ (see above) or `,reset` is required to return to top level. The effects of pushed command levels include:

  - a longer prompt
  - retention of the continuation in effect at the point of error
  - confusion among some newcomers

  With `levels` disabled one must issue a `,push` command immediately following an error in order to retain the error continuation for debugging purposes; otherwise the continuation is lost as soon as the focus object changes. If you don't know anything about the available debugging tools, then `levels` might as well be disabled.

`break-on-warnings` (boolean)
: Enter a new command level when a warning is produced, just as when an error occurs. Normally warnings only result in a displayed message and the program does not stop executing.

`ask-before-loading` (boolean)
: If on, the system will ask before loading modules that are arguments to the `,open` command. `Ask-before-loading` is off by default.

```
> ,set ask-before-loading
will ask before loading modules
> ,open random
Load structure random (y/n)? y
>
```

`load-noisily` (boolean)
: When on, the system will print out the names of modules and files as they are loaded. `Load-noisily` is off by default.

```
> ,set load-noisily
will notify when loading modules and files
> ,open random
[random /usr/local/lib/scheme48/big/random.scm]
>
```

`inline-values` (boolean)
: This controls whether or not the compiler is allowed to substitute variables' values in-line. When `inline-values` mode is on, some Scheme procedures will be substituted in-line; when it is off, none will. Section 2.4 has more information.

`inspector-menu-limit` (positive integer)
: This controls how many items the displayed portion of the inspector menu contains at most. (See Section 3.8.)

`inspector-writing-depth` (positive integer)
: This controls the maximum depth to which the inspector prints values. (See Section 3.8.)

`inspector-writing-length` (positive integer)
: This controls the maximum length to which the inspector prints values. (See Section 3.8.)

`condition-writing-depth` (positive integer)
: This controls the maximum depth to which conditions are printed.

`condition-writing-length` (positive integer)
: This controls the maximum length to which conditions are printed.

`profiler-interrupt-time` (positive integer)
: This controls the time between two profiler sampling interrupts (in milliseconds, see section 3.6).

`profiler-measure-noninstr` (boolean)
: When this flag is enabled, call-times will be measured by the sampling process. When it is disabled, call-times will only be measured when the procedure has been instrumented (see section 3.6).

`trace-writing-length` (positive integer)
: This controls the maximum length to which tracing prints procedure calls.

3.8 Inspection mode

There is a data inspector available via the `,inspect` and `,debug` commands. The inspector is particularly useful with procedures, continuations, and records. The command processor can be taken out of inspection mode by using the `q` command. When in inspection mode, input that begins with a letter or digit is read as a command, not as an expression. To see the value of a variable or number, do `(begin` *exp*`)` or use the `,run` *exp* command.

In inspection mode the command processor prints out a menu of selectable components for the current focus object. To inspect a particular component, just type the corresponding number in the menu. That component becomes the new focus object. For example:

```
> ,inspect '(a (b c) d)
(a (b c) d)

[0] a
[1] (b c)
[2] d
: 1
(b c)

[0] b
[1] c
:
```

When a new focus object is selected the previous one is pushed onto a stack. You can pop the stack, reverting to the previous object, with the `u` command, or use the `stack` command to move to an earlier object.

Commands useful when in inspection mode:

- `u` (up) pop object stack
- `m` (more) print more of a long menu
- `(...)` evaluate a form and select result
- `q` quit
- `template` select a closure or continuation's template (Templates are the static components of procedures; these are found inside of procedures and continuations, and contain the quoted constants and top-level variables referred to by byte-compiled code.)
- `d` (down) move to the next continuation (current object must be a continuation)
- `menu` print the selection menu for the focus object

Multiple selection commands (`u`, `d`, and menu indexes) may be put on a single line.

All ordinary commands are available when in inspection mode. Similarly, the inspection commands can be used when not in inspection mode. For example:

```
> (list 'a '(b c) 'd)
'(a (b c) d)
> ,1
'(b c)
> ,menu
[0] b
[1] c
>
```

If the current command level was initiated because of a breakpoint in the next level down, then `,debug` will invoke the inspector on the continuation at the point of the error. The `u` and `d` (up and down) commands then make the inspected-value stack look like a conventional stack debugger, with continuations playing the role of stack frames. `D` goes to older or deeper continuations (frames), and `u` goes back up to more recent ones.

3.9 Command programs

The `exec` package contains procedures that are used to execute the command processor's commands. A command `,`*foo* is executed by applying the value of the identifier *foo* in the `exec` package to the (suitably parsed) command arguments.

`,exec [`*command*`]`
: Evaluate *command* in the `exec` package. For example, use

  `,exec ,load` *filename*

  to load a file containing commands. If no *command* is given, the `exec` package becomes the execution package for future commands.

The required argument types are as follows:

- filenames should be strings
- other names and identifiers should be symbols
- expressions should be s-expressions
- commands (as for `,config` and `,exec` itself) should be lists of the form `(`*command-name argument* `...)` where *command-name* is a symbol.

For example, the following two commands are equivalent:

```
,config ,load my-file.scm

,exec (config '(load "my-file.scm"))
```

The file `scheme/vm/load-vm.scm` in the source directory contains an example of an `exec` program.

3.10 Building images

`,dump` *filename* `[`*identification*`]`
: Writes the current heap out to a file, which can then be run using the virtual machine. The new image file includes the command processor. If present, *identification* should be a string (written with double quotes); this string will be part of the greeting message as the image starts up.

`,build` *exp filename* `[`*option ...*`]`

Like `,dump`, except that the image file contains the value of *exp*, which should be a procedure of one argument, instead of the command processor. When *filename* is resumed, that procedure will be invoked on the VM's `-a` arguments, which are passed as a list of OS strings (see section 5.15. The procedure should return an integer, which is returned to the program that invoked the VM. The command processor and debugging system are not included in the image (unless you go to some effort to preserve them, such as retaining a continuation).

If `no-warnings` appears as an *option* after the file name, no warnings about undefined external bindings (see Section 8.2) will be printed upon resuming the image. This is useful when the definitions of external bindings appear in shared objects that are only loaded after the resumption of the image.

Doing `,flush` before building an image will reduce the amount of debugging information in the image, making for a smaller image file, but if an error occurs, the error message may be less helpful. Doing `,flush source maps` before loading any programs used in the image will make it still smaller. See section 3.11 for more information.

3.11 Resource query and control

`,time` *exp*

Measure execution time.

`,collect`

Invoke the garbage collector. Ordinarily this happens automatically, but the command tells how much space is available before and after the collection.

`,keep` *kind*

`,flush` *kind*

These control the amount of debugging information retained after compiling procedures. This information can consume a fair amount of space. *kind* is one of the following:

- `maps` - environment maps (local variable names, for inspector)
- `source` - source code for continuations (displayed by inspector)
- `names` - procedure names (as displayed by `write` and in error messages)
- `files` - source file names

These commands refer to future compilations only, not to procedures that already exist. To have any effect, they must be done before programs are loaded. The default is to keep all four types.

`,flush`

The flush command with no argument deletes the database of names of initial procedures. Doing `,flush` before a `,build` or `,dump` will make the resulting image significantly smaller, but will compromise the information content of many error messages.

3.12 Threads

Each command level has its own set of threads. These threads are suspended when a new level is entered and resumed when the owning level again becomes the current level. A thread that raises an error is not resumed unless explicitly restarted using the `,proceed` command. In addition to any threads spawned by the user, each level has a thread that runs the command processor on that level. A new command-processor thread is started if the current one dies or is terminated. When a command level is abandoned for a lower level, or when a level is restarted using `,reset`, all of the threads on that level are terminated and any `dynamic-wind` "after" thunks are run.

The following commands are useful when debugging multithreaded programs:

`,resume [`*number*`]`

Pops out to a given level and resumes running all threads at that level. *Number* defaults to zero.

`,threads`

Invokes the inspector on a list of the threads running at the next lower command level.

`,exit-when-done [`*exp*`]`

Waits until all user threads have completed and then exits back out to shell (or executive or whatever invoked Scheme 48 in the first place). *Exp* should evaluate to an integer which is then returned to the calling program.

3.13 Quite obscure

`,go` *exp*

This is like `,exit` *exp* except that the evaluation of *exp* is tail-recursive with respect to the command processor. This means that the command processor itself can probably be GC'ed, should a garbage collection occur in the execution of *exp*. If an error occurs Scheme 48 will exit with a non-zero value.

`,translate` *from to*

For `load` and the `,load` command (but not for `open-{in|out}put-file`), file names beginning with the string *from* will be changed so that the initial *from* is replaced by the string *to*. E.g.

```
,translate /usr/gjc/ /zu/gjc/
```

will cause `(load "/usr/gjc/foo.scm")` to have the same effect as `(load "/zu/gjc/foo.scm")`.

`,from-file` *filename* *form* `... ,end`
: This is used by the `cmuscheme48` Emacs library to indicate the file from which the *forms* came. *Filename* is then used by the command processor to determine the package in which the *forms* are to be evaluated.

Chapter 4

Module system

This chapter describes Scheme 48's module system. The module system is unique in the extent to which it supports both static linking and rapid turnaround during program development. The design was influenced by Standard ML modules[7] and by the module system for Scheme Xerox[4]. It has also been shaped by the needs of Scheme 48, which is designed to run both on workstations and on relatively small (less than 1 Mbyte) embedded controllers.

Except where noted, everything described here is implemented in Scheme 48, and exercised by the Scheme 48 implementation and some application programs.

Unlike the Common Lisp package system, the module system described here controls the mapping of names to denotations, not the mapping of strings to symbols.

4.1 Introduction

The module system supports the structured division of a corpus of Scheme software into a set of modules. Each module has its own isolated namespace, with visibility of bindings controlled by module descriptions written in a special *configuration language.*

A module may be instantiated multiple times, producing several *packages*, just as a lambda-expression can be instantiated multiple times to produce several different procedures. Since single instantiation is the normal case, we will defer discussion of multiple instantiation until a later section. For now you can think of a package as simply a module's internal environment mapping names to denotations.

A module exports bindings by providing views onto the underlying package. Such a view is called a *structure* (terminology from Standard ML). One module may provide several different views. A structure is just a subset of the package's bindings. The particular set of names whose bindings are exported is the structure's *interface.*

A module imports bindings from other modules by either *opening* or *ac-*

cessing some structures that are built on other packages. When a structure is opened, all of its exported bindings are visible in the client package.

For example:

```
(define-structure foo (export a c cons)
  (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) (+ a (c w)))))
```

This configuration defines two structures, `foo` and `bar`. `foo` is a view on a package in which the `scheme` structure's bindings (including `define` and `+`) are visible, together with bindings for `a`, `b`, and `c`. `foo`'s interface is `(export a c cons)`, so of the bindings in its underlying package, `foo` only exports those three. Similarly, structure `bar` consists of the binding of `d` from a package in which both `scheme`'s and `foo`'s bindings are visible. `foo`'s binding of `cons` is imported from the Scheme structure and then re-exported.

A module's body, the part following `begin` in the above example, is evaluated in an isolated lexical scope completely specified by the package definition's `open` and `access` clauses. In particular, the binding of the syntactic operator `define-structure` is not visible unless it comes from some opened structure. Similarly, bindings from the `scheme` structure aren't visible unless they become so by `scheme` (or an equivalent structure) being opened.

4.2 The configuration language

The configuration language consists of top-level defining forms for modules and interfaces. Its syntax is given in figure 4.1.

A `define-structure` form introduces a binding of a name to a structure. A structure is a view on an underlying package which is created according to the clauses of the `define-structure` form. Each structure has an interface that specifies which bindings in the structure's underlying package can be seen via that structure in other packages.

An `open` clause specifies which structures will be opened up for use inside the new package. At least one structure must be specified or else it will be impossible to write any useful programs inside the package, since `define`, `lambda`, `cons`, etc. will be unavailable. Packages typically include `scheme`, which exports all bindings appropriate to Revised^5 Scheme, in an `open` clause. For building structures that export structures, there is a `defpackage` package that exports the operators of the configuration language. Many other structures, such as record and hash table facilities, are also available in the Scheme 48 implementation.

```
⟨configuration⟩ ⟶ ⟨definition⟩*
 ⟨definition⟩ ⟶  (define-structure ⟨name⟩ ⟨interface⟩ ⟨clause⟩*)
               |  (define-structures ((⟨name⟩ ⟨interface⟩)*) ⟨clause⟩*)
               |  (define-interface ⟨name⟩ ⟨interface⟩)
               |  (define-syntax ⟨name⟩ ⟨transformer-spec⟩)
 ⟨clause⟩ ⟶  (open ⟨structure⟩*)
           |  (access ⟨name⟩*)
           |  (begin ⟨program⟩)
           |  (files ⟨filespec⟩*)
           |  (optimize ⟨optimize-spec⟩*)
           |  (for-syntax ⟨clause⟩*)
 ⟨interface⟩ ⟶  (export ⟨item⟩*)
              |  ⟨name⟩
              |  (compound-interface ⟨interface⟩*)
 ⟨item⟩ ⟶  ⟨name⟩
                 |  (⟨name⟩ ⟨type⟩)
                 |  ((⟨name⟩*) ⟨type⟩)
 ⟨structure⟩ ⟶  ⟨name⟩
              |  (modify ⟨structure⟩ ⟨modifier⟩*)
              |  (subset ⟨structure⟩ (⟨name⟩*))
              |  (with-prefix ⟨structure⟩ ⟨name⟩)
 ⟨modifier⟩ ⟶  (expose ⟨name⟩*)
             |  (hide ⟨name⟩*)
             |  (rename (⟨name⟩₀ ⟨name⟩₁)*)
             |  (alias (⟨name⟩₀ ⟨name⟩₁)*)
             |  (prefix ⟨name⟩)
```

Figure 4.1: The configuration language.

The `modify`, `subset`, and `prefix` forms produce new views on existing structures by renaming or hiding exported names. `Subset` returns a new structure that exports only the listed names from its ⟨structure⟩ argument. `With-prefix` returns a new structure that adds ⟨prefix⟩ to each of the names exported by the ⟨structure⟩ argument. For example, if structure `s` exports `a` and `b`, then

```
(subset s (a))
```

exports only `a` and

```
(with-prefix s p/)
```

exports `a` as `p/a` and `b` as `p/b`.

Both `subset` and `with-prefix` are simple macros that expand into uses of `modify`, a more general renaming form. In a `modify` structure specification the ⟨command⟩s are applied to the names exported by ⟨structure⟩ to produce a new set of names for the ⟨structure⟩'s bindings. `Expose` makes only the listed names visible. `Hide` makes all but the listed names visible. `Rename` makes each

⟨name⟩$_0$ visible as ⟨name⟩$_1$ name and not visible as ⟨name⟩$_0$, while `alias` makes each ⟨name⟩$_0$ visible as both ⟨name⟩$_0$ and ⟨name⟩$_1$. `Prefix` adds ⟨name⟩ to the beginning of each exported name. The modifiers are applied from right to left. Thus

```
(modify scheme (prefix foo/) (rename (car bus))))
```

makes `car` available as `foo/bus`.

The package's body is specified by `begin` and/or `files` clauses. `begin` and `files` have the same semantics, except that for `begin` the text is given directly in the package definition, while for `files` the text is stored somewhere in the file system. The body consists of a Scheme program, that is, a sequence of definitions and expressions to be evaluated in order. In practice, we always use `files` in preference to `begin`; `begin` exists mainly for expository purposes.

A name's imported binding may be lexically overridden or *shadowed* by defining the name using a defining form such as `define` or `define-syntax`. This will create a new binding without having any effect on the binding in the opened package. For example, one can do `(define car 'chevy)` without affecting the binding of the name `car` in the `scheme` package.

Assignments (using `set!`) to imported and undefined variables are not allowed. In order to `set!` a top-level variable, the package body must contain a `define` form defining that variable. Applied to bindings from the `scheme` structure, this restriction is compatible with the requirements of the Revised^5 Scheme report.

It is an error for two of a package's opened structures to export two different bindings for the same name. However, the current implementation does not check for this situation; a name's binding is always taken from the structure that is listed first within the `open` clause. This may be fixed in the future.

File names in a `files` clause can be symbols, strings, or lists (Maclisp-style "namelists"). A "`.scm`" file type suffix is assumed. Symbols are converted to file names by converting to upper or lower case as appropriate for the host operating system. A namelist is an operating-system-independent way to specify a file obtained from a subdirectory. For example, the namelist `(rts record)` specifies the file `record.scm` in the `rts` subdirectory.

If the `define-structure` form was itself obtained from a file, then file names in `files` clauses are interpreted relative to the directory in which the file containing the `define-structure` form was found. You can't at present put an absolute path name in the `files` list.

4.3 Interfaces

`define-interface`

An interface can be thought of as the type of a structure. In its basic form it is just a list of variable names, written `(export` *name* `...)`. However, in place of a name one may write `(`*name type*`)`, indicating the type of *name*'s binding.

The type field is optional, except that exported macros must be indicated with type `:syntax`.

Interfaces may be either anonymous, as in the example in the introduction, or they may be given names by a `define-interface` form, for example

```
(define-interface foo-interface (export a c cons))
(define-structure foo foo-interface ...)
```

In principle, interfaces needn't ever be named. If an interface had to be given at the point of a structure's use as well as at the point of its definition, it would be important to name interfaces in order to avoid having to write them out twice, with risk of mismatch should the interface ever change. But they don't.

Still, there are several reasons to use `define-interface`:

1. It is important to separate the interface definition from the package definitions when there are multiple distinct structures that have the same interface — that is, multiple implementations of the same abstraction.
2. It is conceptually cleaner, and often useful for documentation purposes, to separate a module's specification (interface) from its implementation (package).
3. Our experience is that configurations that are separated into interface definitions and package definitions are easier to read; the long lists of exported bindings just get in the way most of the time.

The `compound-interface` operator forms an interface that is the union of two or more component interfaces. For example,

```
(define-interface bar-interface
  (compound-interface foo-interface (export mumble)))
```

defines `bar-interface` to be `foo-interface` with the name `mumble` added.

4.4 Macros

Hygienic macros, as described in [2, 3], are implemented. Structures may export macros; auxiliary names introduced into the expansion are resolved in the environment of the macro's definition.

For example, the `scheme` structure's `delay` macro is defined by the rewrite rule

`(delay` *exp*`)` $\Longrightarrow$ `(make-promise (lambda ()` *exp*`))`.

The variable `make-promise` is defined in the `scheme` structure's underlying package, but is not exported. A use of the `delay` macro, however, always accesses the correct definition of `make-promise`. Similarly, the `case` macro expands into uses of `cond`, `eqv?`, and so on. These names are exported by `scheme`, but their correct bindings will be found even if they are shadowed by definitions in the client package.

4.5 Higher-order modules

There are `define-module` and `define` forms for defining modules that are intended to be instantiated multiple times. But these are pretty kludgey — for example, compiled code isn't shared between the instantiations — so we won't describe them yet. If you must know, figure it out from the following grammar.

```
⟨definition⟩ ⟶  (define-module (⟨name⟩ (⟨name⟩ ⟨interface⟩)*)
                    ⟨definition⟩*
                    ⟨name⟩)
              |  (define ⟨name⟩ (⟨name⟩ ⟨name⟩*))
```

4.6 Compiling and linking

Scheme 48 has a static linker that produces stand-alone heap images from module descriptions. The programmer specifies a particular procedure in a particular structure to be the image's startup procedure (entry point), and the linker traces dependency links as given by `open` and `access` clauses to determine the composition of the heap image.

There is not currently any provision for separate compilation; the only input to the static linker is source code. However, it will not be difficult to implement separate compilation. The unit of compilation is one module (not one file). Any opened or accessed structures from which macros are obtained must be processed to the extent of extracting its macro definitions. The compiler knows from the interface of an opened or accessed structure which of its exports are macros. Except for macros, a module may be compiled without any knowledge of the implementation of its opened and accessed structures. However, inter-module optimization may be available as an option.

The main difficulty with separate compilation is resolution of auxiliary bindings introduced into macro expansions. The module compiler must transmit to the loader or linker the search path by which such bindings are to be resolved. In the case of the `delay` macro's auxiliary `make-promise` (see example above), the loader or linker needs to know that the desired binding of `make-promise` is the one apparent in `delay`'s defining package, not in the package being loaded or linked.

4.7 Semantics of configuration mutation

During program development it is often desirable to make changes to packages and interfaces. In static languages it may be necessary to recompile and re-link a program in order for such changes to be reflected in a running system. Even in interactive Common Lisp implementations, a change to a package's exports often requires reloading clients that have already mentioned names whose bindings change. 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 only be reflected by re-`read`ing all such references.

The Scheme 48 development environment supports rapid turnaround in modular program development by allowing mutations to a program's configuration, and giving a clear semantics to such mutations. The rule is that variable bindings in a running program are always resolved according to 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 `b`, which is free in the definition of `d`, has no binding in `bar`'s package. Suppose that `b` was supposed to be exported by `foo`, but was omitted from `foo-interface` by mistake. 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 procedure `d` is called.

Similarly, it is also 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 `foo` that exports a name `x`, and `foo`'s implementation obtains the binding of `x` as an import from some other structure `bar`. Then *C* will see the binding from `bar`. If one then alters `foo` so that it shadows `bar`'s binding of `x` with a definition of its own, then procedures in *C* that reference `x` will automatically see `foo`'s definition instead of the one from `bar` that they saw earlier.

This semantics might appear to require a large amount of computation on every variable reference: The specified behavior requires scanning the package's list of opened structures, examining their interfaces, on every variable reference, not just at compile time. However, the development environment uses caching with cache invalidation to make variable references fast.

4.8 Command processor support

While it is possible to use the Scheme 48 static linker for program development, it is far more convenient to use the development environment, which supports rapid turnaround for program changes. The programmer interacts with the development environment through a *command processor*. The command processor is like the usual Lisp read-eval-print loop in that it accepts Scheme forms to evaluate. However, all meta-level operations, such as exiting the Scheme system or requests for trace output, are handled by *commands,* which are lexically distinguished from Scheme forms. This arrangement is borrowed from the Symbolics Lisp Machine system, and is reminiscent of non-Lisp debuggers. Commands are a little easier to type than Scheme forms (no parentheses, so you don't have to shift), but more importantly, making them distinct from Scheme forms ensures that programs' namespaces aren't cluttered with inappropriate bindings. Equivalently, the command set is available for use regardless of what bindings happen to be visible in the current program. This is especially important in conjunction with the module system, which puts strict controls on visibility of bindings.

The Scheme 48 command processor supports the module system with a variety of special commands. For commands that require structure names, these names are resolved in a designated configuration package that is distinct from the current package for evaluating Scheme forms given to the command processor. The command processor interprets Scheme forms in a particular current package, and there are commands that move the command processor between different packages.

Commands are introduced by a comma (`,`) and end at the end of line. The command processor's prompt consists of the name of the current package followed by a greater-than (`>`).

`,open` *structure**

The `,open` command opens new structures in the current package, as if the package's definition's `open` clause had listed *structure*. As with `open` clauses the visible names can be modified, as in

```
,open (subset foo (bar baz))
```

which only makes the `bar` and `baz` bindings from structure `foo` visible.

`,config`

The `,config` command sets the command processor's current package to be the current configuration package. Forms entered at this point are interpreted as being configuration language forms, not Scheme forms.

`,config` *command*

This form of the `,config` command executes another command in the current configuration package. For example,

```
,config ,load foo.scm
```

interprets configuration language forms from the file `foo.scm` in the current configuration package.

`,config-package-is` *struct-name*

The `,config-package-is` command designates a new configuration package for use by the `,config` command and resolution of *struct-name*s for other commands such as `,in` and `,open`. See section 4.9 for information on making new configuration packages.

`,in` *struct-name*

The `,in` command moves the command processor to a specified structure's underlying package. For example:

```
user> ,config
config> (define-structure foo (export a)
          (open scheme))
config> ,in foo
foo> (define a 13)
foo> a
13
```

In this example the command processor starts in a package called `user`, but the `,config` command moves it into the configuration package, which has the name `config`. The `define-structure` form binds, in `config`, the name `foo` to a structure that exports `a`. Finally, the command `,in foo` moves the command processor into structure `foo`'s underlying package.

A package's body isn't executed (evaluated) until the package is *loaded*, which is accomplished by the `,load-package` command.

`,in` *struct-name command*

This form of the `,in` command executes a single command in the specified package without moving the command processor into that package. Example:

```
,in mumble (cons 1 2)
,in mumble ,trace foo
```

`,user [`*command*`]`

This is similar to the `,config` and `,in` commands. It moves to or executes a command in the user package (which is the default package when the Scheme 48 command processor starts).

`,user-package-is` *name*

The `,user-package-is` command designates a new user package for use by the `,user` command.

`,load-package` *struct-name*

The `,load-package` command ensures that the specified structure's underlying package's program has been loaded. This consists of (1) recursively ensuring that the packages of any opened or accessed structures are loaded, followed by (2) executing the package's body as specified by its definition's `begin` and `files` forms.

`,reload-package` *struct-name*

This command re-executes the structure's package's program. It is most useful if the program comes from a file or files, when it will update the package's bindings after mutations to its source file.

`,load` *filespec* `...`

The `,load` command executes forms from the specified file or files in the current package. `,load` *filespec* is similar to `(load "`*filespec*`")` except that the name `load` needn't be bound in the current package to Scheme's `load` procedure.

`,for-syntax [`*command*`]`

This is similar to the `,config` and `,in` commands. It moves to or executes a command in the current package's "package for syntax," which is the package in which the forms *f* in `(define-syntax` *name f*`)` are evaluated.

`,new-package [`*struct-name* `...]`

The `,new-package` command creates a new package and moves the command processor to it. With no arguments, only the standard Scheme bindings are visible in the new package. Otherwise, the structures specified as command arguments (and not the `scheme` structure) are opened in the new package.

`,structure` *name interface*

The `,structure` command defines *name* in the configuration package to be a structure with interface *interface* based on the current package.

4.9 Configuration packages

It is possible to set up multiple configuration packages. The default configuration package opens the following structures:

- `module-system`, which exports `define-structure` and the other configuration language keywords, as well as standard types and type constructors (`:syntax`, `:value`, `proc`, etc.).
- `built-in-structures`, which exports structures that are built into the initial Scheme 48 image; these include `scheme`, `threads`, `tables`, and `records`.
- `more-structures`, which exports additional structures that are available in the development environment. A complete listing can be found in the definition of `more-structures-interface` at the end of the file `scheme/packages.scm`.

Note that it does not open `scheme`.

You can define additional configuration packages by making a package that opens `module-system` and, optionally, `built-in-structures`, `more-structures`, or other structures that export structures and interfaces.

For example:

```
> ,config (define-structure foo (export)
            (open module-system
                  built-in-structures
                  more-structures))
> ,in foo
foo> (define-structure x (export a b)
       (open scheme)
       (files x))
foo>
```

Unfortunately, the above example does not work. The problem is that every environment in which `define-structure` is used must also have a way to create "syntactic towers". A new syntactic tower is required whenever a new environment is created for compiling the source code in the package associated with a new structure. The environment's tower is used at compile time for evaluating the *macro-source* in

`(define-syntax` *name macro-source*`)`
`(let-syntax ((`*name macro-source*`) ...)` *body*`)`

and so forth. It is a "tower" because that environment, in turn, has to say what environment to use if `macro-source` itself contains a use of `let-syntax`.

The simplest way to provide a tower maker is to pass on the one used by an existing configuration package. The special form `export-syntactic-tower-maker` creates an interface that exports a configuration package's tower. The following example uses `export-syntactic-tower-maker` and the `,structure` command to obtain a tower maker and create a new configuration environment.

```
> ,config ,structure t (export-syntactic-tower-maker)
> ,config (define-structure foo (export)
            (open module-system
                  t
                  built-in-structures
                  more-structures))
```

Before Scheme 48 1.9, `export-syntactic-tower-maker` was named `export-reflective-tower-maker`; this name is still supported for backwards compatibility.

4.10 Discussion

This module system was not designed as the be-all and end-all of Scheme module systems; it was only intended to help us organize the Scheme 48 system. Not only does the module system help avoid name clashes by keeping different subsystems in different namespaces, it has also helped us to tighten up and generalize Scheme 48's internal interfaces. Scheme 48 is unusual among Lisp implementations in admitting many different possible modes of operation. Examples of such multiple modes include the following:

- Linking can be either static or dynamic.

- The development environment (compiler, debugger, and command processor) can run either in the same address space as the program being developed or in a different address space. The environment and user program may even run on different processors under different operating systems[8].

- The virtual machine can be supported by either of two implementations of its implementation language, Prescheme.

The module system has been helpful in organizing these multiple modes. By forcing us to write down interfaces and module dependencies, the module system helps us to keep the system clean, or at least to keep us honest about how clean or not it is.

The need to make structures and interfaces second-class instead of first-class results from the requirements of static program analysis: it must be possible for the compiler and linker to expand macros and resolve variable bindings before the program is executed. Structures could be made first-class (as in FX[10]) if a type system were added to Scheme and the definitions of exported macros were defined in interfaces instead of in module bodies, but even in that case types and interfaces would remain second-class.

The prohibition on assignment to imported bindings makes substitution a valid optimization when a module is compiled as a block. The block compiler first scans the entire module body, noting which variables are assigned. Those that aren't assigned (only `define`d) may be assumed never assigned, even if they are exported. The optimizer can then perform a very simple-minded analysis to determine automatically that some procedures can and should have their calls compiled in line.

The programming style encouraged by the module system is consistent with the unextended Scheme language. Because module system features do not generally show up within module bodies, an individual module may be understood by someone who is not familiar with the module system. This is a great aid to code presentation and portability. If a few simple conditions are met (no name conflicts between packages, and use of `files` in preference to `begin`), then a multi-module program can be loaded into a Scheme implementation that does not support the module system. The Scheme 48 static linker satisfies these conditions, and can therefore run in other Scheme implementations. Scheme 48's bootstrap process, which is based on the static linker, is therefore nonincestuous. This contrasts with most other integrated programming environments, such as Smalltalk-80, where the system can only be built using an existing version of the system itself.

Like ML modules, but unlike Scheme Xerox modules, this module system is compositional. That is, structures are constructed by single syntactic units that compose existing structures with a body of code. In Scheme Xerox, the set of modules that can contribute to an interface is open-ended — any module can contribute bindings to any interface whose name is in scope. The module system implementation is a cross-bar that channels definitions from modules to interfaces. The module system described here has simpler semantics and makes dependencies easier to trace. It also allows for higher-order modules, which Scheme Xerox considers unimportant.

Chapter 5

Libraries

Use the `,open` command (section 3.4) or the module language (chapter 2.6) to open the structures described below.

5.1 General utilities

These are in the `big-util` structure.

- `(atom?` *value*`)` → *boolean*

`(atom?` *x*`)` is the same as `(not (pair?` *x*`))`.

- `(null-list?` *list*`)` → *boolean*

Returns true for the empty list, false for a pair, and signals an error otherwise.

- `(neq?` *value value*`)` → *boolean*

`(neq?` *x y*`)` is the same as `(not (eq?` *x y*`))`.

- `(n=` *number number*`)` → *boolean*

`(n=` *x y*`)` is the same as `(not (=` *x y*`))`.

- `(identity` *value*`)` → *value*
- `(no-op` *value*`)` → *value*

These both just return their argument. `No-op` is guaranteed not to be compiled in-line, `identity` may be.

- `(memq?` *value list*`)` → *boolean*

Returns true if *value* is in *list*, false otherwise.

- `(any?` *predicate list*`)` → *boolean*

Returns true if *predicate* is true for any element of *list*.

- `(every?` *predicate list*`)` → *boolean*

Returns true if *predicate* is true for every element of *list*.

- `(any` *predicate list*`)` → *value*
- `(first` *predicate list*`)` → *value*

`Any` returns some element of *list* for which *predicate* is true, or false if there are none. `First` does the same except that it returns the first element for which *predicate* is true.

- `(filter` *predicate list*`)` → *list*
- `(filter!` *predicate list*`)` → *list*

Returns a list containing all of the elements of *list* for which *predicate* is true. The order of the elements is preserved. `Filter!` may reuse the storage of *list*.

- `(filter-map` *procedure list*`)` → *list*

The same as `filter` except the returned list contains the results of applying *procedure* instead of elements of *list*. `(filter-map` *p l*`)` is the same as `(filter identity (map` *p l*`))`.

- `(partition-list` *predicate list*`)` → *list list*
- `(partition-list!` *predicate list*`)` → *list list*

The first return value contains those elements *list* for which *predicate* is true, the second contains the remaining elements. The order of the elements is preserved. `Partition-list!` may reuse the storage of the *list*.

- `(remove-duplicates` *list*`)` → *list*

Returns its argument with all duplicate elements removed. The first instance of each element is preserved.

- `(delq` *value list*`)` → *list*
- `(delq!` *value list*`)` → *list*
- `(delete` *predicate list*`)` → *list*

All three of these return *list* with some elements removed. `Delq` removes all elements `eq?` to *value*. `Delq!` does the same and may modify the list argument. `Delete` removes all elements for which *predicate* is true. Both `delq` and `delete` may reuse some of the storage in the list argument, but won't modify it.

- `(reverse!` *list*`)` → *list*

Destructively reverses *list*.

- `(concatenate-symbol` *value* ...`)` → *symbol*

Returns the symbol whose name is produced by concatenating the `display`ed representations of *value* ....

```
(concatenate-symbol 'abc "-" 4) ⟹ 'abc-4
```

5.2 Pretty-printing

These are in the `pp` structure.

- `(p` *value*`)`
- `(p` *value output-port*`)`
- `(pretty-print` *value output-port position*`)`

Pretty-print *value* The current output port is used if no port is specified. *Position* is the starting offset. *Value* will be pretty-printed to the right of this column.

5.3 Bitwise integer operations

These functions use the two's-complement representation for integers. There is no limit to the number of bits in an integer. They are in the structures `bitwise` and `big-scheme`.

- `(bitwise-and` *integer integer* ...`)` $\rightarrow$ *integer*
- `(bitwise-ior` *integer integer* ...`)` $\rightarrow$ *integer*
- `(bitwise-xor` *integer integer* ...`)` $\rightarrow$ *integer*
- `(bitwise-not` *integer*`)` $\rightarrow$ *integer*

These perform various logical operations on integers on a bit-by-bit basis. '`ior`' is inclusive OR and '`xor`' is exclusive OR.

- `(arithmetic-shift` *integer bit-count*`)` $\rightarrow$ *integer*

Shifts the integer by the given bit count, which must be an integer, shifting left for positive counts and right for negative ones. Shifting preserves the integer's sign.

- `(bit-count` *integer*`)` $\rightarrow$ *integer*

Counts the number of bits set in the integer. If the argument is negative a bitwise NOT operation is performed before counting.

5.4 Byte vectors

These are homogeneous vectors of small integers ($0 \leq i \leq 255$). The functions that operate on them are analogous to those for vectors. They are in the structure `byte-vectors`.

- `(byte-vector?` *value*`)` $\rightarrow$ *boolean*
- `(make-byte-vector` *k fill*`)` $\rightarrow$ *byte-vector*
- `(byte-vector` *b* ...`)` $\rightarrow$ *byte-vector*
- `(byte-vector-length` *byte-vector*`)` $\rightarrow$ *integer*
- `(byte-vector-ref` *byte-vector k*`)` $\rightarrow$ *integer*
- `(byte-vector-set!` *byte-vector k b*`)`
- `(byte-vector=?` *byte-vector byte-vector*`)` $\rightarrow$ *boolean*

5.5 Sparse vectors

These are vectors that grow as large as they need to. That is, they can be indexed by arbitrarily large nonnegative integers. The implementation allows for arbitrarily large gaps by arranging the entries in a tree. They are in the structure `sparse-vectors`.

- `(make-sparse-vector)` → *sparse-vector*
- `(sparse-vector-ref` *sparse-vector k*`)` → *value*
- `(sparse-vector-set!` *sparse-vector k value*`)`
- `(sparse-vector->list` *sparse-vector*`)` → *list*

`Make-sparse-vector`, `sparse-vector-ref`, and `sparse-vector-set!` are analogous to `make-vector`, `vector-ref`, and `vector-set!`, except that the indices passed to `sparse-vector-ref` and `sparse-vector-set!` can be arbitrarily large. For indices whose elements have not been set in a sparse vector, `sparse-vector-ref` returns `#f`.

`Sparse-vector->list` is for debugging: It returns a list of the consecutive elements in a sparse vector from 0 to the highest element that has been set. Note that the list will also include all the `#f` elements for the unset elements.

5.6 Cells

These hold a single value and are useful when a simple indirection is required. The system uses these to hold the values of lexical variables that may be `set!`.

- `(cell?` *value*`)` → *boolean*
- `(make-cell` *value*`)` → *cell*
- `(cell-ref` *cell*`)` → *value*
- `(cell-set!` *cell value*`)`

5.7 Queues

These are ordinary first-in, first-out queues. The procedures are in structure `queues`.

- `(make-queue)` → *queue*
- `(queue?` *value*`)` → *boolean*
- `(queue-empty?` *queue*`)` → *boolean*
- `(list->queue` *values*`)` → *queue*
- `(enqueue!` *queue value*`)`
- `(enqueue-many!` *queue list*`)`
- `(queue-head-or-value` *queue value*`)` → *value*
- `(queue-head-or-thunk` *queue thunk*`)` → *value*
- `(queue-head` *queue*`)` → *value*
- `(maybe-queue-head` *queue*`)` → *value*
- `(dequeue-or-value!` *queue value*`)` → *value*

- (dequeue-or-thunk! *queue thunk*) → *value*
- (dequeue! *queue*) → *value*
- (maybe-dequeue! *queue*) → *value*
- (empty-queue! *queue*)

`Make-queue` creates an empty queue, `queue?` is a predicate for identifying queues, and `queue-empty?` tells you if a queue is empty. `List->queue` returns a queue containing *values*, preserving their order. `Enqueue!` adds one value to the queue; `enqueue-many!` adds a list of values to the queue. `Queue-head-or-value`, `queue-head-or-thunk`, `queue-head`, and `maybe-queue-head` return the first value in *queue* if it is not empty; if the queue is empty, `queue-head-or-value` returns *value*, `queue-head-or-thunk` tail-calls *thunk*, `queue-head` raises an error, and `maybe-queue-head` returns `#f`. `Dequeue-or-value!`, `dequeue-or-thunk!`, `dequeue!`, and `maybe-dequeue!` remove a value from the queue if one is available; if the queue is empty, `dequeue-or-value!` returns *value*, `dequeue-or-thunk!` tail-calls *thunk*, `dequeue!` raises an error, and `maybe-dequeue!` returns `#f`. `Empty-queue!` removes all values from *queue*.

`(Dequeue-or-value! q value)` is more efficient than, but otherwise equivalent to:

```
(ensure-atomicity
 (if (queue-empty? q)
     value
     (dequeue! q)))
```

Because `queue-head` and `dequeue!` raise exceptions if they are called on an empty queue, they *must not* be called with a proposal already active unless `queue-empty?` has returned `#f` with the same proposal active.

The following procedures are not used in the Scheme 48 system, and are *very* slow. These operations may be removed from the `queues` structure in a future revision.

- (queue-length *queue*) → *integer*
- (queue->list *queue*) → *values*
- (delete-from-queue! *queue value*) → *boolean*
- (on-queue? *queue value*) → *boolean*

`Queue-length` returns the number of values in *queue*. `Queue->list` returns the values in *queue* as a list, in the order in which the values were added. `Delete-from-queue!` removes the first instance of *value* from *queue*, using `eqv?` for comparisons. `Delete-from-queue!` returns `#t` if it removes an element and `#f` if it does not. `On-queue?` returns `#t` if *value* is in the *queue* (using `eqv?` for comparisons) and `#f` if it is not.

5.8 Arrays

These provide N-dimensional, zero-based arrays and are in the structure `arrays`. The array interface is derived from one invented by Alan Bawden.

- `(make-array` *value dimension*$_0$ ...`)` → *array*
- `(array` *dimensions element*$_0$ ...`)` → *array*
- `(copy-array` *array*`)` → *array*

`Make-array` makes a new array with the given dimensions, each of which must be a non-negative integer. Every element is initially set to *value*. `Array` Returns a new array with the given dimensions and elements. *Dimensions* must be a list of positive integers, The number of elements should be the equal to the product of the dimensions. The elements are stored in row-major order.

```
(make-array 'a 2 3) → {Array 2 3}

(array '(2 3) 'a 'b 'c 'd 'e 'f)
    → {Array 2 3}
```

`Copy-array` returns a copy of *array*. The copy is identical to the *array* but does not share storage with it.

- `(array?` *value*`)` → *boolean*

Returns `#t` if *value* is an array.

- `(array-ref` *array index*$_0$ ...`)` → *value*
- `(array-set!` *array value index*$_0$ ...`)`
- `(array->vector` *array*`)` → *vector*
- `(array-shape` *array*`)` → *list*

`Array-ref` returns the specified array element and `array-set!` replaces the element with *value*.

```
(let ((a (array '(2 3) 'a 'b 'c 'd 'e 'f)))
  (let ((x (array-ref a 0 1)))
    (array-set! a 'g 0 1)
    (list x (array-ref a 0 1))))
    → '(b g)
```

`Array->vector` returns a vector containing the elements of *array* in row-major order. `Array-shape` returns the dimensions of the array as a list.

- `(make-shared-array` *array linear-map dimension*$_0$ ...`)` → *array*

`Make-shared-array` makes a new array that shares storage with *array* and uses *linear-map* to map indexes to elements. *Linear-map* must accept as many arguments as the number of *dimension*s given and must return a list of non-negative integers that are valid indexes into *array*.

```
(array-ref (make-shared-array a f i0 i1 ...)
           j0 j1 ...)
```

is equivalent to

```
(apply array-ref a (f j0 j1 ...))
```

As an example, the following function makes the transpose of a two-dimensional array:

```
(define (transpose array)
  (let ((shape (array-shape array)))
    (make-shared-array array
                       (lambda (x y)
                         (list y x))
                       (cadr shape)
                       (car shape))))

(array->vector
  (transpose
    (array '(2 3) 'a 'b 'c 'd 'e 'f)))
      → '(a d b e c f)
```

5.9 Records

New types can be constructed using the `define-record-type` macro from the `define-record-types` structure The general syntax is:

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

This makes the following definitions:

- *type-name* type
- (*constructor-name field-init* ...) → *type-name*
- (*predicate-name value*) → *boolean*
- (*accessor-name type-name*) → *value*
- (*modifier-name type-name value*)

Type-name is the record type itself, and can be used to specify a print method (see below). *Constructor-name* is a constructor that accepts values for the fields whose tags are specified. *Predicate-name* is a predicate that returns `#t` for elements of the type and `#f` for everything else. The *accessor-name*s retrieve the values of fields, and the *modifier-name*'s update them. *Tag* is used in printing instances of the record type and the *field-tag*s are used in the inspector and to match constructor arguments with fields. If *tag* is not specified, *type-name* is used instead.

- (`define-record-discloser` *type discloser*)

`Define-record-discloser` determines how records of type *type* are printed. *Discloser* should be procedure which takes a single record of type *type* and returns a list whose car is a symbol. The record will be printed as the value returned by *discloser* with curly braces used instead of the usual parenthesis.

For example

```
(define-record-type pare :pare
  (kons x y)
  pare?
  (x kar set-kar!)
  (y kdr))
```

defines `kons` to be a constructor, `kar` and `kdr` to be accessors, `set-kar!` to be a modifier, and `pare?` to be a predicate for a new type of object. The type itself is named `:pare`. `Pare` is a tag used in printing the new objects.

By default, the new objects print as `#{Pare}`. The print method can be modified using `define-record-discloser`:

```
(define-record-discloser :pare
  (lambda (p) `(pare ,(kar p) ,(kdr p))))
```

will cause the result of `(kons 1 2)` to print as `#{Pare 1 2}`.

`Define-record-resumer` (section 8.7.6) can be used to control how records are stored in heap images.

5.9.1 Low-level access to records

Records are implemented using primitive objects exactly analogous to vectors. Every record has a record type (which is another record) in the first slot. Note that use of these procedures, especially `record-set!`, breaks the record abstraction described above; caution is advised.

These procedures are in the structure `records`.

- `(make-record` *n value*`)` → *record*
- `(record` *value ...*`)` → *record-vector*
- `(record?` *value*`)` → *boolean*
- `(record-length` *record*`)` → *integer*
- `(record-type` *record*`)` → *value*
- `(record-ref` *record i*`)` → *value*
- `(record-set!` *record i value*`)`

These the same as the standard `vector-` procedures except that they operate on records. The value returned by `record-length` includes the slot holding the record's type. `(record-type` *x*`)` is equivalent to `(record-ref` *x* `0)`.

5.9.2 Record types

Record types are themselves records of a particular type (the first slot of `:record-type` points to itself). A record type contains four values: the name of the record type, a list of the names its fields, and procedures for disclosing and resuming records of that type. Procedures for manipulating them are in the structure `record-types`.

- `(make-record-type` *name field-names*`)` → *record-type*
- `(record-type?` *value*`)` → *boolean*
- `(record-type-name` *record-type*`)` → *symbol*
- `(record-type-field-names` *record-type*`)` → *symbols*

- `(record-constructor` *record-type field-names*`)` → *procedure*
- `(record-predicate` *record-type*`)` → *procedure*
- `(record-accessor` *record-type field-name*`)` → *procedure*
- `(record-modifier` *record-type field-name*`)` → *procedure*

These procedures construct the usual record-manipulating procedures. `Record-constructor` returns a constructor that is passed the initial values for the fields specified and returns a new record. `Record-predicate` returns a predicate that return true when passed a record of type *record-type* and false otherwise. `Record-accessor` and `record-modifier` return procedures that reference and set the given field in records of the appropriate type.

- `(define-record-discloser` *record-type discloser*`)`
- `(define-record-resumer` *record-type resumer*`)`

`Record-types` is the initial exporter of `define-record-discloser` (re-exported by `define-record-types` described above) and `define-record-resumer` (re-exported by `external-calls` (section 8.7.6)).

The procedures described in this section can be used to define new record-type-defining macros.

```
(define-record-type pare :pare
  (kons x y)
  pare?
  (x kar set-kar!)
  (y kdr))
```

is (semantically) equivalent to

```
(define :pare (make-record-type 'pare '(x y)))
(define kons (record-constructor :pare '(x y)))
(define kar (record-accessor :pare 'x))
(define set-kar! (record-modifier :pare 'x))
(define kdr (record-accessor :pare 'y))
```

The "(semantically)" above is because `define-record-type` adds declarations, which allows the type checker to detect some misuses of records, and uses more efficient definitions for the constructor, accessors, and modifiers. Ignoring the declarations, which will have to wait for another edition of the manual, what the above example actually expands into is:

```
(define :pare (make-record-type 'pare '(x y)))
(define (kons x y) (record :pare x y))
(define (kar r) (checked-record-ref r :pare 1))
```

```
(define (set-kar! r new)
  (checked-record-set! r :pare 1 new))
(define (kdr r) (checked-record-ref r :pare 2))
```

`Checked-record-ref` and `Checked-record-set!` are low-level procedures that check the type of the record and access or modify it using a single VM instruction.

5.10 Finite record types

The structure `finite-types` has two macros for defining 'finite' record types. These are record types for which there are a fixed number of instances, all of which are created at the same time as the record type itself. The syntax for defining an enumerated type is:

```
(define-enumerated-type tag type-name
  predicate-name
  vector-of-instances-name
  name-accessor
  index-accessor
  (instance-name ...))
```

This defines a new record type, bound to *type-name*, with as many instances as there are *instance-name*'s. *Vector-of-instances-name* is bound to a vector containing the instances of the type in the same order as the *instance-name* list. *Tag* is bound to a macro that when given an *instance-name* expands into an expression that returns corresponding instance. The name lookup is done at macro expansion time. *Predicate-name* is a predicate for the new type. *Name-accessor* and *index-accessor* are accessors for the name and index (in *vector-of-instances*) of instances of the type.

```
(define-enumerated-type color :color
  color?
  colors
  color-name
  color-index
  (black white purple maroon))

(color-name (vector-ref colors 0)) → black
(color-name (color white))         → white
(color-index (color purple))       → 2
```

Finite types are enumerations that allow the user to add additional fields in the type. The syntax for defining a finite type is:

```
(define-finite-type tag type-name
  (field-tag ...)
```

  predicate-name
  vector-of-instances-name
  name-accessor
  index-accessor
  (*field-tag* *accessor-name* [*modifier-name*])
  ...
  ((*instance-name* *field-value* ...)
   ...))

The additional fields are specified exactly as with `define-record-type`. The field arguments to the constructor are listed after the *type-name*; these do not include the name and index fields. The form ends with the names and the initial field values for the instances of the type. The instances are constructed by applying the (unnamed) constructor to these initial field values. The name must be first and the remaining values must match the *field-tag*s in the constructor's argument list.

```
(define-finite-type color :color
  (red green blue)
  color?
  colors
  color-name
  color-index
  (red   color-red)
  (green color-green)
  (blue  color-blue)
  ((black    0   0   0)
   (white  255 255 255)
   (purple 160  32 240)
   (maroon 176  48  96)))

(color-name (color black))          → black
(color-name (vector-ref colors 1))  → white
(color-index (color purple))        → 2
(color-red (color maroon))          → 176
```

5.11 Sets over finite types

The structure `enum-sets` has a macro for defining types for sets of elements of finite types. These work naturally with the finite types defined by the `finite-types` structure, but are not tied to them. The syntax for defining such a type is:

`(define-enum-set-type` *id* *type-name* *predicate* *constructor*
   element-syntax *element-predicate* *all-elements* *element-index-ref*`)`

This defines *id* to be syntax for constructing sets, *type-name* to be a value representing the type, *predicate* to be a predicate for those sets, and *constructor* a procedure for constructing one from a list.

Element-syntax must be the name of a macro for constructing set elements from names (akin to the *tag* argument to `define-enumerated-type`). *Element-predicate* must be a predicate for the element type, *all-elements* a vector of all values of the element type, and *element-index-ref* must return the index of an element within the *all-elements* vector.

- `(enum-set->list` *enum-set*`)` → *list*
- `(enum-set-member?` *enum-set enumerand*`)` → *boolean*
- `(enum-set=?` *enum-set enum-set*`)` → *boolean*
- `(enum-set-subset?` *enum-set enum-set*`)` → *boolean*
- `(enum-set-union` *enum-set enum-set*`)` → *enum-set*
- `(enum-set-intersection` *enum-set enum-set*`)` → *enum-set*
- `(enum-set-difference` *enum-set enum-set*`)` → *enum-set*
- `(enum-set-negation` *enum-set*`)` → *enum-set*

`Enum-set->list` converts a set into a list of its elements. `Enum-set-member?` tests for membership. `Enum-set=?` tests two sets of equal type for equality. (If its arguments are not of the same type, `enum-set=?` raises an exception.) `Enum-set-subset?` tests, for two sets of equal type, if the first set is a subset of the second one. `Enum-set-union` computes the union of two sets of equal type, `enum-set-intersection` computes the intersection, `enum-set-difference` computes the intersection, and `enum-set-negation` computes the complement of a set.

Here is an example. Given an enumerated type:

```
(define-enumerated-type color :color
  color?
  colors
  color-name
  color-index
  (red blue green))
```

we can define sets of colors:

```
(define-enum-set-type color-set :color-set
                      color-set?
                      make-color-set
  color color? colors color-index)

> (enum-set->list (color-set red blue))
(#{Color red} #{Color blue})
> (enum-set->list (enum-set-negation (color-set red blue)))
(#{Color green})
> (enum-set-member? (color-set red blue) (color blue))
#t
```

5.12 Hash tables

These are generic hash tables, and are in the structure `tables`. Strictly speaking they are more maps than tables, as every table has a value for every possible key (for that type of table). All but a finite number of those values are `#f`.

- `(make-table)` → *table*
- `(make-symbol-table)` → *symbol-table*
- `(make-string-table)` → *string-table*
- `(make-integer-table)` → *integer-table*
- `(make-table-maker` *compare-proc hash-proc*`)` → *procedure*
- `(make-table-immutable!` *table*`)`

The first four functions listed make various kinds of tables. `Make-table` returns a table whose keys may be symbols, integer, characters, booleans, or the empty list (these are also the values that may be used in `case` expressions). As with `case`, comparison is done using `eqv?`. The comparison procedures used in symbol, string, and integer tables are `eq?`, `string=?`, and `=`.

`Make-table-maker` takes two procedures as arguments and returns a nullary table-making procedure. *Compare-proc* should be a two-argument equality predicate. *Hash-proc* should be a one argument procedure that takes a key and returns a non-negative integer hash value. If `(`*compare-proc x y*`)` returns true, then `(= (`*hash-proc x*`) (`*hash-proc y*`))` must also return true. For example, `make-integer-table` could be defined as `(make-table-maker = abs)`.

`Make-table-immutable!` prohibits future modification to its argument.

- `(table?` *value*`)` → *boolean*
- `(table-ref` *table key*`)` → *value or* `#f`
- `(table-set!` *table key value*`)`
- `(table-walk` *procedure table*`)`
- `(table->entry-list` *table*`)` → *list of pairs*

`Table?` is the predicate for tables. `Table-ref` and `table-set!` access and modify the value of *key* in *table*. `Table-walk` applies *procedure*, which must accept two arguments, to every associated key and non-`#f` value in `table`. `Table->entry-list` returns a list with the values of the table.

- `(default-hash-function` *value*`)` → *integer*
- `(string-hash` *string*`)` → *integer*
- `(symbol-hash` *symbol*`)` → *integer*

`Default-hash-function` is the hash function used in the tables returned by `make-table`, `string-hash` is the one used by `make-string-table`, and `symbol-hash` is the one used by `make-symbol-table`.

5.13 Port extensions

These procedures are in structure `extended-ports`.

- `(make-string-input-port` *string*`)` → *input-port*
- `(make-string-output-port)` → *output-port*
- `(string-output-port-output` *string-output-port*`)` → *string*

`Make-string-input-port` returns an input port that that reads characters from the supplied string. An end-of-file object is returned if the user reads past the end of the string. `Make-string-output-port` returns an output port that saves the characters written to it. These are then returned as a string by `string-output-port-output`.

```
(read (make-string-input-port "(a b)"))
    → '(a b)

(let ((p (make-string-output-port)))
  (write '(a b) p)
  (let ((s (string-output-port-output p)))
    (display "c" p)
    (list s (string-output-port-output p))))
    → '("(a b)" "(a b)c")
```

- `(limit-output` *output-port n procedure*`)`

Procedure is called on an output port. Output written to that port is copied to *output-port* until *n* characters have been written, at which point `limit-output` returns. If *procedure* returns before writing *n* characters, then `limit-output` also returns at that time, regardless of how many characters have been written.

- `(make-tracking-input-port` *input-port*`)` → *input-port*
- `(make-tracking-output-port` *output-port*`)` → *output-port*
- `(current-row` *port*`)` → *integer or* `#f`
- `(current-column` *port*`)` → *integer or* `#f`
- `(fresh-line` *output-port*`)`

`Make-tracking-input-port` and `make-tracking-output-port` return ports that keep track of the current row and column and are otherwise identical to their arguments. Closing a tracking port does not close the underlying port. `Current-row` and `current-column` return *port*'s current read or write location. They return `#f` if *port* does not keep track of its location. `Fresh-line` writes a newline character to *output-port* if `(current-row` *port*`)` is not 0.

```
(define p (make-tracking-output-port (open-output-file "/tmp/temp")))
(list (current-row p) (current-column p))
    → '(0 0)
(display "012" p)
(list (current-row p) (current-column p))
    → '(0 3)
(fresh-line p)
(list (current-row p) (current-column p))
    → '(1 0)
```

```
(fresh-line p)
(list (current-row p) (current-column p))
    → '(1 0)
```

5.14 Fluid bindings

These procedures implement dynamic binding and are in structure `fluids`. A *fluid* is a cell whose value can be bound dynamically. Each fluid has a top-level value that is used when the fluid is unbound in the current dynamic environment.

- `(make-fluid` *value*`)` → *fluid*
- `(fluid` *fluid*`)` → *value*
- `(let-fluid` *fluid value thunk*`)` → *value(s)*
- `(let-fluids` *fluid*$_0$ *value*$_0$ *fluid*$_1$ *value*$_1$ *. . . thunk*`)` → *value(s)*

`Make-fluid` returns a new fluid with *value* as its initial top-level value. `Fluid` returns `fluid`'s current value. `Let-fluid` calls `thunk`, with *fluid* bound to *value* until `thunk` returns. Using a continuation to throw out of the call to `thunk` causes *fluid* to revert to its original value, while throwing back in causes *fluid* to be rebound to *value*. `Let-fluid` returns the value(s) returned by *thunk*. `Let-fluids` is identical to `let-fluid` except that it binds an arbitrary number of fluids to new values.

```
(let* ((f (make-fluid 'a))
       (v0 (fluid f))
       (v1 (let-fluid f 'b
             (lambda ()
               (fluid f))))
       (v2 (fluid f)))
  (list v0 v1 v2))
  → '(a b a)

(let ((f (make-fluid 'a))
      (path '())
      (c #f))
  (let ((add (lambda ()
               (set! path (cons (fluid f) path)))))
    (add)
    (let-fluid f 'b
      (lambda ()
        (call-with-current-continuation
          (lambda (c0)
            (set! c c0)))
        (add)))
    (add)
    (if (< (length path) 5)
        (c)
        (reverse path))))
  → '(a b a b a)
```

5.15 OS strings

On common operating systems such as Unix and Windows, various parameters to OS functionality—such as file names, user names, command-line arguments etc.—appear as text in most contexts, but are really byte sequences: On Unix, the byte sequence may be interpreted as text through some locale-determined encoding. On Windows, such parameters are typically represented as sequences of UTF-16 code units. In both cases, not every such byte sequence has a string equivalent: On Unix, a byte sequence encoding a file name using Latin-1 often cannot be decoded using UTF-8. On Windows, unpaired UTF-16 surrogates are admissible in encodings, and no lossless text decoding for them exists.

For representing such string-like parameters, Scheme 48 uses an abstraction called *OS strings*. An OS string is created from either a string or a NUL-terminated byte sequence stored in a byte vector, and has an associated text codec (see section 6.6.1) that is able to convert from one representation to the other. The exact meaning of a NUL-terminated byte sequence is dependent on this text codec. However, only codecs for encodings that are a conservative extension of ASCII (such as ASCII itself, Latin-1, or UTF-8) should be used here, to allow a minimal set of portable file names. (The Windows port uses a special synthetic encoding called UTF-8of16 compatible with UTF-8 but capable of encoding even invalid UTF-16 internally, but uses the UTF-8 codec at the Scheme level.)

Most procedures accepting OS strings also accept strings or byte vectors, which are then used to construct a OS string. In the headers of the specifications of these procedures, such arguments occur as *os-string-thing*. The standard Scheme procedures such as `open-input-file` that take file names all accept *os-string-thing* arguments. OS strings are in the `os-strings` structure.

- `(os-string?` *value*`)` → *boolean*
- `(make-os-string` *text-codec string/byte-vector*`)` → *os-string*

The `os-string?` predicate returns `#t` if its argument is an OS string, `#f` otherwise.

The `make-os-string` procedure creates an OS string from a text codec and a byte vector or string that defines its contents. If the argument is a byte vector, it does not matter if it is NUL-terminated or not.

- `(string->os-string` *string*`)` → *os-string*
- `(byte-vector->os-string` *byte-vector*`)` → *os-string*
- `(x->os-string` *os-string-thing*`)` → *os-string*

These procedures create an OS string from a string, a byte-vector (whose last value should be 0), and an *os-string-thing* argument, respectively, always using the standard OS-string text codec (see below).

- `(os-string->byte-vector` *os-string*`)` $\rightarrow$ *byte-vector*
- `(os-string->string` *os-string*`)` $\rightarrow$ *string*

These procedures yield the contents of an OS string. For an OS string created from a string, `os-string->string` will return a string with the same contents; for an OS string created from a byte vector, `os-string->byte-vector` will return a byte vector with the same contents. For the other cases, data loss as determined by the text codec is possible.

- `(os-string-text-codec` *os-string*`)` $\rightarrow$ *text-codec*

This procedure returns the text codec of the OS string.

- `(os-string=?` *os-string os-string*`)` $\rightarrow$ *boolean*

This procedure returns `#t` if its arguments denote the same byte sequence, `#f` otherwise.

- `(x->os-byte-vector` *os-string-thing*`)` $\rightarrow$ *byte-vector*
- `(string->os-byte-vector` *string*`)` $\rightarrow$ *byte-vector*

These are convenience procedures: The first is the composition of `x->os-string` and `os-string->byte-vector`, and the second is the composition of `string->os-string` and `os-string->byte-vector`.

- `(current-os-string-text-codec)` $\rightarrow$ *text-codec*
- `(call-with-os-string-text-codec` *text-codec thunk*`)` $\rightarrow$ *value(s)*

The `current-os-string-text-codec` returns the current text codec used for creating new OS strings. The initial default is determined by the operating system. (On Unix, this is the text codec determined by the locale. On Windows, this is UTF-8.) The `call-with-os-string-text-codec` procedure dynamically binds the current text codec to *text-codec* during the invocation of *thunk*.

5.16 Shell commands

Structure `c-system-function` provides access to the C `system()` function.

- `(have-system?)` $\rightarrow$ *boolean*
- `(system` *os-string-thing*`)` $\rightarrow$ *integer*

`Have-system?` returns true if the underlying C implementation has a command processor. `(System` *string*`)` passes *string* to the C `system()` function and returns the result.

```
(begin
  (system "echo foo > test-file")
  (call-with-input-file "test-file" read))
→ 'foo
```

5.17 Sockets

Structure `sockets` provides access to TCP/IP sockets for interprocess and network communication.

- `(open-socket)` → *socket*
- `(open-socket` *port-number*`)` → *socket*
- `(socket-port-number` *socket*`)` → *integer*
- `(close-socket` *socket*`)`
- `(socket-accept` *socket*`)` → *input-port output-port*
- `(get-host-name)` → *string*

`Open-socket` creates a new socket. If no *port-number* is supplied the system picks one at random. `Socket-port-number` returns a socket's port number. `Close-socket` closes a socket, preventing any further connections. `Socket-accept` accepts a single connection on *socket*, returning an input port and an output port for communicating with the client. If no client is waiting `socket-accept` blocks until one appears. `Get-host-name` returns the network name of the machine.

- `(socket-client` *host-name port-number*`)` → *input-port output-port*

`Socket-client` connects to the server at *port-number* on the machine named *host-name*. `Socket-client` blocks until the server accepts the connection.

The following simple example shows a server and client for a centralized UID service.

```
(define (id-server)
  (let ((socket (open-socket)))
    (display "Waiting on port ")
    (display (socket-port-number socket))
    (newline)
    (let loop ((next-id 0))
      (call-with-values
        (lambda ()
          (socket-accept socket))
        (lambda (in out)
          (display next-id out)
          (close-input-port in)
          (close-output-port out)
          (loop (+ next-id 1)))))))

(define (get-id machine port-number)
  (call-with-values
    (lambda ()
      (socket-client machine port-number))
    (lambda (in out)
      (let ((id (read in)))
        (close-input-port in)
        (close-output-port out)
        id))))
```

5.18 Profiling

The profiler can be used programmatically from within the code and its results can be processed dynamically.

The structure `profiler` offers basic data structures and functions. All functions need a `profile-data` record argument that can be created with `make-empty-profile-data`.

- `(make-empty-profile-data)` $\rightarrow$ *profile-data*
- `(profile-thunk` *profile-data thunk [interrupt-time [with-non-instr?]]*`)` $\rightarrow$ *value*

`profile-thunk` executes a thunk under the profiler. The data is stored in the *profile-data* record passed as argument. Optionally the timeout for the profiler interrupt can be passed (in milliseconds). The return value of `profile-thunk` is the return value of `thunk`. By default, non-instrumented code will be profiled by the sampling process. By passing `with-non-instr? = #f` only instrumented functions will be profiled.

After profiling data, can be retrieved with the following accessors:

- `(profile-data-starttime` *profile-data*`)` $\rightarrow$ *number*
- `(profile-data-endtime` *profile-data*`)` $\rightarrow$ *number*
- `(profile-data-runtime` *profile-data*`)` $\rightarrow$ *number*
- `(profile-data-memoryuse` *profile-data*`)` $\rightarrow$ *number*
- `(profile-data-gcruns` *profile-data*`)` $\rightarrow$ *number*
- `(profile-data-samples` *profile-data*`)` $\rightarrow$ *number*
- `(profile-data-interrupttime` *profile-data*`)` $\rightarrow$ *number*

Times are in milliseconds, memory usage in bytes.

The following functions produce the same output as the `,profile` command (see section 3.6). They all take an optional *port* argument. Default is the `current-output-port`.

- `(profile-display` *profile-data [port]*`)`
- `(profile-display-overview` *profile-data [port]*`)`
- `(profile-display-flat` *profile-data [port]*`)`
- `(profile-display-tree` *profile-data [port]*`)`

`profile-display` prints the full output of the profiler. The other `profile-display-...` functions only display the respective part of the output.

The single fields in the flat profile can be retrieved with the following accessors:

- `(profile-function-calls` *profile-data names*`)`
- `(profile-function-reccalls` *profile-data names*`)`
- `(profile-function-nonreccalls` *profile-data names*`)`
- `(profile-function-occurs` *profile-data names*`)`

- `(profile-function-hist` *profile-data names*`)`
- `(profile-function-memoryuse` *profile-data names*`)`
- `(profile-function-timeshare` *profile-data names*`)`
- `(profile-function-time-cumulative` *profile-data names*`)`
- `(profile-function-time-self` *profile-data names*`)`

Here *names* is the list of names specifying the function, optionally with it's module. For example, `names = ("dynamic-wind", "wind")` would specify the `dynamic-wind` function in the module `wind`, if it was seen while profiling. If two or more functions match, the first one is used. If no function matches, the functions return `#{Unspecific}`.

The argument *names* can also be a plain string, as in `(profile-display-function-flat prof-data "module")`. This will display all flat function profiles that have "module" either as name or module.

The meanings of the fields that the functions return are as follows:

- `calls`: total number of calls (recursive and non-recursive) to the function
- `reccalls`: total number of recursive calls to the function
- `nonreccalls`: total number of non-recursive calls to the function
- `occurs`: number of times the function was seen on stack while profiling
- `hist`: number of times the function was seen running while profiling
- `memoryuse`: bytes of memory used by the function
- `timeshare`: percentage of time used by the function itself (number from 0 to 1)
- `time-cumulative`: total time in ms the function was on call-stack
- `time-self`: total time in ms the function actively running

The following shows a short example of the usage of the profiler interface, where `main` is the function to be profiled:

```
(define prof-data (make-empty-profile-data))

(profile-thunk prof-data (lambda () (main 22)))
(display "Samples: ")
(display (profile-data-samples prof-data))
(newline)

(profile-display-overview prof-data)
(profile-display-flat prof-data (current-output-port))
(profile-display-tree prof-data)

; print only function "a"
```

```
(profile-display-function-flat prof-data '("a"))

; print only function "a" in file "x.scm"
(profile-display-function-flat prof-data '("a" "x.scm"))

; print all profiled functions in file "x.scm"
(profile-display-function-flat prof-data "x.scm")

; print percentage of time "a" was running
(display (* (profile-function-timeshare prof-data '("a")) 100))
```

5.19 Macros for writing loops

`Iterate` and `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` and `reduce` are in structure `reduce`.

5.19.1 `Iterate`

The syntax of `iterate` is:

```
(iterate loop-name
         ((sequence-type  element-variable  sequence-data ...)
          ...)
         ((state-variable  initial-value)
          ...)
  body-expression
  [final-expression])
```

`Iterate` steps the *element-variable*s in parallel through the sequences, while each *state-variable* has the corresponding *initial-value* for the first iteration and have later values supplied by *body-expression*. If any sequence has reached its limit the value of the `iterate` expression is the value of *final-expression*, if present, or the current values of the *state-variable*s, returned as multiple values. If no sequence has reached its limit, *body-expression* is evaluated and either calls *loop-name* with new values for the *state-variable*s, or returns some other value(s).

The *loop-name* and the *state-variable*s and *initial-value*s behave exactly as in named-`let`. The named-`let` expression

```
(let loop-name ((state-variable initial-value) ...)
  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 `define`s):

```
(iterate loop-name
         ()
         ((state-variable initial-value) ...)
  (let () body ...))
```

The *sequence-type*s are keywords (they are actually macros of a particular form; it is easy to add additional types of sequences). Examples are `list*` which walks down the elements of a list and `vector*` which does the same for vectors. For each iteration, each *element-variable* is bound to the next element of the sequence. The *sequence-data* gives the actual list or vector or whatever.

If there is a *final-expression*, it is evaluated when the end of one or more sequences is reached. If the *body-expression* does not call *loop-name* the *final-expression* is not evaluated. The *state-variable*s are visible in *final-expression* but the *sequence-variable*s are not.

The *body-expression* and the *final-expression* are in tail-position within the `iterate`. Unlike named-`let`, the behavior 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).

5.19.2 Reduce

If an `iterate` expression is not meant to terminate before a sequence has reached its end, *body-expression* will always end with a tail call to *loop-name*. `Reduce` is a 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*. The *body-expression* returns new values of the *state-variable*s instead of passing them to *loop-name*. Thus *body-expression* must return as many values as there are state variables. By special dispensation, if there are no state variables then *body-expression* may return any number of values, all of which are ignored.

The syntax of `reduce` is:

```
(reduce ((sequence-type element-variable sequence-data ...)
         ...)
        ((state-variable initial-value)
         ...)
  body-expression
  [final-expression])
```

The value(s) returned by an instance of `reduce` is the value(s) returned by the *final-expression*, 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-var* and a wrapper for the *body-expression* that calls the *loop-var*.

```
(iterate loop
         ((sequence-type element-variable sequence-data ...)
          ...)
         ((state-variable  initial-value)
          ...)
  (call-with-values (lambda ()
                      body-expression)
                    loop)
  [final-expression])
```

5.19.3 Sequence types

The predefined sequence types are:

- `(list*` *elt-var list*`)` syntax
- `(list-spine*` *elt-var list*`)` syntax
- `(list-spine-cycle-safe*` *elt-var list on-cycle-thunk*`)` syntax
- `(vector*` *elt-var vector*`)` syntax
- `(string*` *elt-var string*`)` syntax
- `(count*` *elt-var start* `[`*end* `[`*step*`]])` syntax
- `(bits*` *elt-var i* `[`*size*`])` syntax
- `(input*` *elt-var input-port read-procedure*`)` syntax
- `(stream*` *elt-var procedure initial-data*`)` syntax

For lists, vectors, and strings the element variable is bound to the successive elements of the list or vector, or the characters in the string.

For `list-spine*` the element variable is bound to the successive pairs in the spine of the list. `List-spine-cycle-safe*` is similar, but calls *on-cycle-thunk* with no arguments and with the continuation of the loop macro at an unspecified time if *list* contains a cycle.

For `count*` the element variable 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. 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)` doesn't terminate because it begins past the *end* value and `(count* i 0 1 2)` doesn't terminate because it skips over the *end* value.

For `bits*`, the element variable is bound to a sequence of representations of successive bit-fields of i, from least to most significant. If *size* is present, it must be a positive exact integer, and the element variable is bound to a sequence of *size*-bit integers. If *size* is omitted, `bits*` iterates through single bits, and the element variable is bound to a sequence of booleans. `#t` represents 1, and `#f` represents 0.

For `input*` the elements are the results of successive applications of *read-procedure* to *input-port*. The sequence ends when *read-procedure* returns an end-of-file object.

For a stream, the *procedure* takes the current data value as an argument and returns two values, the next value of the sequence and a new data value. If the new data is `#f` then the previous element was the last one. For example,

```
  (list* elt my-list)
```

is the same as

```
  (stream* elt list->stream my-list)
```

where `list->stream` is

```
  (lambda (list)
    (if (null? list)
        (values 'ignored #f)
        (values (car list) (cdr list))))
```

5.19.4 Synchronous sequences

When using the sequence types described above, a loop terminates when any of its sequences reaches its end. To help detect bugs it is useful to have sequence types that check to see if two or more sequences end on the same iteration. For this purpose there is second set of sequence types called synchronous sequences. These are identical to the ones listed above 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 a non-synchronous sequence no further checking is done.

The synchronous sequences are:

- `(list%` *elt-var list*`)` syntax
- `(list-spine%` *elt-var list*`)` syntax
- `(list-spine-cycle-safe%` *elt-var list on-cycle-thunk*`)` syntax
- `(vector%` *elt-var vector*`)` syntax
- `(string%` *elt-var string*`)` syntax
- `(count%` *elt-var start end* `[`*step*`])` syntax
- `(input%` *elt-var input-port read-procedure*`)` syntax
- `(stream%` *elt-var procedure initial-data*`)` syntax

Note that the synchronous `count%` must have an *end*, unlike the nonsynchronous `count*`.

5.19.5 Examples

Gathering the indexes of list elements that answer true to some predicate.

```
(lambda (my-list predicate)
  (reduce ((list* elt my-list)
           (count* i 0))
```

```
          ((hits '()))
    (if (predicate elt)
        (cons i hits)
        hits)
    (reverse hits))
```

Looking for the index of an element of a list.

```
(lambda (my-list predicate)
  (iterate loop
           ((list* elt my-list)
            (count* i 0))
           ()                                ; no state
    (if (predicate elt)
        i
        (loop))))
```

Reading one line.

```
(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)
        ; no newline at end of file
        (list->string (reverse chars)))))
```

Counting the lines in a file. We can't use `count*` because we need the value of the count after the loop has finished.

```
(define (line-count name)
  (call-with-input-file name
    (lambda (in)
      (reduce ((input* l in read-line))
              ((i 0))
        (+ i 1)))))
```

5.19.6 Defining sequence types

The sequence types are object-oriented macros similar to enumerations. A non-synchronous sequence macro needs to supply three values: `#f` to indicate that it isn't synchronous, a list of state variables and their initializers, and the code for one iteration. The first two methods are CPS'ed: they take another macro and argument to which to pass their result. The `sync` method gets no additional arguments. The `state-vars` method is passed a list of names which will be bound to the arguments to the sequence. The final method, for the step, 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.

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 final-exp)
     (if (null? list-var)
         final-exp
         (let ((value-var (car list-var))
               (list-var (cdr list-var)))
           loop-body)))))
```

Synchronized sequences are the same, except that they need to provide a termination test to be used when some other synchronized method terminates the loop.

```
(define-syntax list%
  (syntax-rules (sync done)
    ((list% sync (next more))
     (next #t more))
    ((list% done (start-list) (list-var))
     (null? list-var))
    ((list% stuff ...)
     (list* stuff ...))))
```

5.19.7 Expanded code

The expansion of

```
  (reduce ((list* x '(1 2 3)))
          ((r '()))
    (cons x r))
```

is

```
  (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 inefficiencies in this code are the `final` and `continue` procedures, both of which could be substituted in-line. The macro expander could do the substitution for `continue` when there is no explicit proceed variable, as in this case, but not in general.

5.20 Sorting lists and vectors

(This section, as the libraries it describes, was written mostly by Olin Shivers for the draft of SRFI 32.)

The sort libraries in Scheme 48 include

- vector insert sort (stable)
- vector heap sort
- vector quick sort (with regular comparisons and with median-of-3 pivot picking)
- vector merge sort (stable)
- pure and destructive list merge sort (stable)
- stable vector and list merge
- miscellaneous sort-related procedures: vector and list merging, sorted predicates, vector binary search, vector and list delete-equal-neighbor procedures.
- a general, non-algorithmic set of procedure names for general sorting and merging

5.20.1 Design rules

What vs. how There are two different interfaces: "what" (simple) and "how" (detailed).

Simple you specify semantics: datatype (list or vector), mutability, and stability.

Detailed you specify the actual algorithm (quick, heap, insert, merge). Different algorithms have different properties, both semantic and pragmatic, so these exports are necessary.

It is necessarily the case that the specifications of these procedures make statements about execution "pragmatics." For example, the sole distinction between heap sort and quick sort—both of which are provided by this library—-is one of execution time, which is not a "semantic" distinction. Similar resource-use statements are made about "iterative" procedures, meaning that they can execute on input of arbitrary size in a constant number of stack frames.

Consistency across procedure signatures The two interfaces share common procedure signatures wherever possible, to facilitate switching a given call from one procedure to another.

Less-than parameter first, data parameter after These procedures uniformly observe the following parameter order: the data to be sorted comes after the comparison procedure. That is, we write

`(sort <` *list*`)`

not

`(sort` *list* `<)`

Ordering, comparison procedures and stability These routines take a $<$ comparison procedure, not a $\leq$ comparison procedure, and they sort into increasing order. The difference between a $<$ spec and a $\leq$ spec comes up in two places:

- the definition of an ordered or sorted data set, and
- the definition of a stable sorting algorithm.

We say that a data set (a list or vector) is *sorted* or *ordered* if it contains no adjacent pair of values $\ldots x, y \ldots$ such that $y < x$.

In other words, scanning across the data never takes a "downwards" step.

If you use a $\leq$ procedure where these algorithms expect a $<$ procedure, you may not get the answers you expect. For example, the `list-sorted?` procedure will return false if you pass it a $\leq$ comparison procedure and an ordered list containing adjacent equal elements.

A "stable" sort is one that preserves the pre-existing order of equal elements. Suppose, for example, that we sort a list of numbers by comparing their absolute values, i.e., using comparison procedure

```
(lambda (x y) (< (abs x) (abs y)))
```

If we sort a list that contains both 3 and -3:

$$\ldots 3, \ldots, -3 \ldots$$

then a stable sort is an algorithm that will not swap the order of these two elements, that is, the answer is guaranteed to to look like

$$\ldots 3, -3 \ldots$$

not

$$\ldots -3, 3 \ldots$$

Choosing $<$ for the comparison procedure instead of $\leq$ affects how stability is coded. Given an adjacent pair x, y, `(<` y x`)` means "x should be moved in front of x"—otherwise, leave things as they are. So using a $\leq$ procedure where a $<$ procedure is expected will *invert* stability.

This is due to the definition of equality, given a $<$ comparator:

```
(and (not (< x y))
     (not (< y x)))
```

The definition is rather different, given a $\leq$ comparator:

```
(and (<= x y)
     (<= y x))
```

A "stable" merge is one that reliably favors one of its data sets when equal items appear in both data sets. *All merge operations in this library are stable*, breaking ties between data sets in favor of the first data set—elements of the first list come before equal elements in the second list.

So, if we are merging two lists of numbers ordered by absolute value, the stable merge operation `list-merge`

```
(list-merge (lambda (x y) (< (abs x) (abs y)))
            '(0 -2 4 8 -10) '(-1 3 -4 7))
```

reliably places the 4 of the first list before the equal-comparing -4 of the second list:

```
(0 -1 -2 4 -4 7 8 -10)
```

Some sort algorithms *will not work correctly* if given a $\leq$ when they expect a $<$ comparison (or vice-versa).

In short, if your comparison procedure f answers true to `(`f `x x)`, then

- using a stable sorting or merging algorithm will not give you a stable sort or merge,
- `list-sorted?` may surprise you.

Note that you can synthesize a $<$ procedure from a $\leq$ procedure with

```
(lambda (x y) (not (<= y x)))
```

if need be.

Precise definitions give sharp edges to tools, but require care in use. "Measure twice, cut once."

All vector operations accept optional subrange parameters The vector operations specified below all take optional `start`/`end` arguments indicating a selected subrange of a vector's elements. If a `start` parameter or `start`/`end` parameter pair is given to such a procedure, they must be exact, non-negative integers, such that

$$0 \leq start \leq end \leq \texttt{(vector-length } vector\texttt{)}$$

where *vector* is the related vector parameter. If not specified, they default to 0 and the length of the vector, respectively. They are interpreted to select the range [*start*, *end*), that is, all elements from index *start* (inclusive) up to, but not including, index *end*.

Required vs. allowed side-effects `List-sort!` and `List-stable-sort!` are allowed, but not required, to alter their arguments' cons cells to construct the result list. This is consistent with the what-not-how character of the group of procedures to which they belong (the `sorting` structure).

The `list-delete-neighbor-dups!`, `list-merge!` and `list-merge-sort!` procedures, on the other hand, provide specific algorithms, and, as such, explicitly commit to the use of side-effects on their input lists in order to guarantee their key algorithmic properties (e.g., linear-time operation).

5.20.2 Procedure specification

Structure name	Functionality
`sorting`	General sorting for lists and vectors
`sorted`	Sorted predicates for lists and vectors
`list-merge-sort`	List merge sort
`vector-merge-sort`	Vector merge sort
`vector-heap-sort`	Vector heap sort
`vector-quick-sort`	Vector quick sort
`vector-quick-sort3`	Vector quick sort with 3-way comparisons
`vector-insert-sort`	Vector insertion sort
`delete-neighbor-duplicates`	List and vector delete neighbor duplicates
`binary-searches`	Vector binary search

Note that there is no "list insert sort" package, as you might as well always use list merge sort. The reference implementation's destructive list merge sort will do fewer `set-cdr!`s than a destructive insert sort.

Procedure naming and functionality Almost all of the procedures described below are variants of two basic operations: sorting and merging. These procedures are consistently named by composing a set of basic lexemes to indicate what they do.

Lexeme	Meaning
`sort`	The procedure sorts its input data set by some $<$ comparison procedure.
`merge`	The procedure merges two ordered data sets into a single ordered result.
`stable`	This lexeme indicates that the sort is a stable one.
`vector`	The procedure operates upon vectors.
`list`	The procedure operates upon lists.
`!`	Procedures that end in `!` are allowed, and sometimes required, to reuse their input storage to construct their answer.

Types of parameters and return values In the procedures specified below,

- A `<` or `=` parameter is a procedure accepting two arguments taken from the specified procedure's data set(s), and returning a boolean;
- `Start` and `end` parameters are exact, non-negative integers that serve as vector indices selecting a subrange of some associated vector. When specified, they must satisfy the relation

  $$0 \le start \le end \le \texttt{(vector-length } vector\texttt{)}$$

  where *vector* is the associated vector.

Passing values to procedures with these parameters that do not satisfy these types is an error.

If a procedure is said to return "unspecified," this means that nothing at all is said about what the procedure returns, not even the number of return values. Such a procedure is not even required to be consistent from call to call in the nature or number of its return values. It is simply required to return a value (or values) that may be passed to a command continuation, e.g. as the value of an expression appearing as a non-terminal subform of a `begin` expression. Note that in R^{5}RS, this restricts such a procedure to returning a single value; non-R^{5}RS systems may not even provide this restriction.

`sorting`—general sorting package

This library provides basic sorting and merging functionality suitable for general programming. The procedures are named by their semantic properties, i.e., what they do to the data (sort, stable sort, merge, and so forth).

- `(list-sorted?` $<$ *list*`)` $\rightarrow$ *boolean*
- `(list-merge` $<$ *list*$_1$ *list*$_2$`)` $\rightarrow$ *list*
- `(list-merge!` $<$ *list*$_1$ *list*$_2$`)` $\rightarrow$ *list*
- `(list-sort` $<$ *lis*`)` $\rightarrow$ *list*
- `(list-sort!` $<$ *lis*`)` $\rightarrow$ *list*
- `(list-stable-sort` $<$ *list*`)` $\rightarrow$ *list*
- `(list-stable-sort!` $<$ *list*`)` $\rightarrow$ *list*

- `(list-delete-neighbor-dups = ` *list*`)` $\rightarrow$ *list*
- `(vector-sorted? < ` *v [start [end]]*`)` $\rightarrow$ *boolean*
- `(vector-merge < ` v_1 v_2 *[start*1 *[end*1 *[start*2 *[end*2*]]]]*`)` $\rightarrow$ *vector*
- `(vector-merge! < ` *v* v_1 v_2 *[start [start*1 *[end*1 *[start*2 *[end*2*]]]]]*`)`
- `(vector-sort < ` *v [start [end]]*`)` $\rightarrow$ *vector*
- `(vector-sort! < ` *v [start [end]]*`)`
- `(vector-stable-sort < ` *v [start [end]]*`)` $\rightarrow$ *vector*
- `(vector-stable-sort! < ` *v [start [end]]*`)`
- `(vector-delete-neighbor-dups = ` *v [start [end]]*`)` $\rightarrow$ *vector*

Procedure	Suggested algorithm
`list-sort`	vector heap or quick
`list-sort!`	list merge sort
`list-stable-sort`	vector merge sort
`list-stable-sort!`	list merge sort
`vector-sort`	heap or quick sort
`vector-sort!`	heap or quick sort
`vector-stable-sort`	vector merge sort
`vector-stable-sort!` merge sort	

`List-Sorted?` and `vector-sorted?` return true if their input list or vector is in sorted order, as determined by their < comparison parameter.

All four merge operations are stable: an element of the initial list $list_1$ or vector $vector_1$ will come before an equal-comparing element in the second list $list_2$ or vector $vector_2$ in the result.

The procedures

- `list-merge`
- `list-sort`
- `list-stable-sort`
- `list-delete-neighbor-dups`

do not alter their inputs and are allowed to return a value that shares a common tail with a list argument.

The procedure

- `list-sort!`
- `list-stable-sort!`

are "linear update" operators—they are allowed, but not required, to alter the cons cells of their arguments to produce their results.

On the other hand, the `list-merge!` procedure make only a single, iterative, linear-time pass over its argument list, using `set-cdr!`s to rearrange the cells of the list into the final result —it works "in place." Hence, any cons cell appearing in the result must have originally appeared in an input. The intent of this iterative-algorithm commitment is to allow the programmer to be sure that if, for example, `list-merge!` is asked to merge two ten-million-element lists, the operation will complete without performing some extremely (possibly twenty-million) deep recursion.

The vector procedures

- `vector-sort`
- `vector-stable-sort`
- `vector-delete-neighbor-dups`

do not alter their inputs, but allocate a fresh vector for their result, of length $end - start$.

The vector procedures

- `vector-sort!`
- `vector-stable-sort!`

sort their data in-place. (But note that `vector-stable-sort!` may allocate temporary storage proportional to the size of the input .)

`Vector-merge` returns a vector of length $(end_1 - start_1 + (end_2 - start_2)$.

`Vector-merge!` writes its result into vector v, beginning at index *start*, for indices less than $end = start + (end_1 - start_1) + (end_2 - start_2)$. The target subvector $v[start, end)$ may not overlap either source subvector $vector_1[start_1, end_1)$ $vector_2[start_2, end_2)$.

The `...-delete-neighbor-dups-...` procedures: These procedures delete adjacent duplicate elements from a list or a vector, using a given element-equality procedure. The first/leftmost element of a run of equal elements is the one that survives. The list or vector is not otherwise disordered.

These procedures are linear time—much faster than the $O(n^2)$ general duplicate-element deletors that do not assume any "bunching" of elements (such as the ones provided by SRFI 1). If you want to delete duplicate elements from a large list or vector, you can sort the elements to bring equal items together, then use one of these procedures, for a total time of $O(n \log(n))$.

The comparison procedure = passed to these procedures is always applied $(= \ x \ y)$ where x comes before y in the containing list or vector.

- `List-delete-neighbor-dups` does not alter its input list; its answer may share storage with the input list.
- `Vector-delete-neighbor-dups` does not alter its input vector, but rather allocates a fresh vector to hold the result.

Examples:

```
(list-delete-neighbor-dups = '(1 1 2 7 7 7 0 -2 -2))
  ⟹ (1 2 7 0 -2)
```

```
(vector-delete-neighbor-dups = '#(1 1 2 7 7 7 0 -2 -2))
  ⟹ #(1 2 7 0 -2)

(vector-delete-neighbor-dups = '#(1 1 2 7 7 7 0 -2 -2) 3 7)
  ⟹ #(7 0 -2)
```

Algorithm-specific sorting packages

These packages provide more specific sorting functionality, that is, specific commitment to particular algorithms that have particular pragmatic consequences (such as memory locality, asymptotic running time) beyond their semantic behaviour (sorting, stable sorting, merging, etc.). Programmers that need a particular algorithm can use one of these packages.

`sorted`—**sorted predicates**

- `(list-sorted?` $<$ *list*`)` $\rightarrow$ *boolean*
- `(vector-sorted?` $<$ *vector*`)` $\rightarrow$ *boolean*
- `(vector-sorted?` $<$ *vector start*`)` $\rightarrow$ *boolean*
- `(vector-sorted?` $<$ *vector start end*`)` $\rightarrow$ *boolean*

Return `#f` iff there is an adjacent pair $\ldots x, y \ldots$ in the input list or vector such that $y < x$. The optional *start*/*end* range arguments restrict `vector-sorted?` to the indicated subvector.

`list-merge-sort`—**list merge sort**

- `(list-merge-sort` $<$ *list*`)` $\rightarrow$ *list*
- `(list-merge-sort!` $<$ *list*`)` $\rightarrow$ *list*
- `(list-merge` $list_1$ $<$ $list_2$`)` $\rightarrow$ *list*
- `(list-merge!` $list_1$ $<$ $list_2$`)` $\rightarrow$ *list*

The sort procedures sort their data using a list merge sort, which is stable. (The reference implementation is, additionally, a "natural" sort. See below for the properties of this algorithm.)

The `!` procedures are destructive—they use `set-cdr!`s to rearrange the cells of the lists into the proper order. As such, they do not allocate any extra cons cells—they are "in place" sorts.

The merge operations are stable: an element of $list_1$ will come before an equal-comparing element in $list_2$ in the result list.

`vector-merge-sort`—**vector merge sort**

- `(vector-merge-sort` $<$ *vector [start [end [temp]]]*`)` $\rightarrow$ *vector*
- `(vector-merge-sort!` $<$ *vector [start [end [temp]]]*`)`
- `(vector-merge` $<$ $vector_1$ $vector_2$ [$start_1$ [end_1 [$start_2$ [end_2]]]]`)` $\rightarrow$ *vector*

- `(vector-merge! <` *vector* *vector*$_1$ *vector*$_2$ *[start [start*$_1$ *[end*$_1$ *[start*$_2$ *[end*$_2$*]]]]]*`)`

The sort procedures sort their data using vector merge sort, which is stable. (The reference implementation is, additionally, a "natural" sort. See below for the properties of this algorithm.)

The optional *start/end* arguments provide for sorting of subranges, and default to 0 and the length of the corresponding vector.

Merge-sorting a vector requires the allocation of a temporary "scratch" work vector for the duration of the sort. This scratch vector can be passed in by the client as the optional *temp* argument; if so, the supplied vector must be of size $\leq$ *end*, and will not be altered outside the range [start,end). If not supplied, the sort routines allocate one themselves.

The merge operations are stable: an element of *vector*$_1$ will come before an equal-comparing element in *vector*$_2$ in the result vector.

- `Vector-merge-sort!` leaves its result in $vector[start, end)$.
- `Vector-merge-sort` returns a vector of length $end - start$.
- `Vector-merge` returns a vector of length $(end_1 - start_1) + (end_2 - start_2)$.
- `Vector-merge!` writes its result into *vector*, beginning at index *start*, for indices less than $end = start + (end_1 - start_1) + (end_2 - start_2)$. The target subvector

  $$vector[start, end)$$

  may not overlap either source subvector

  $$vector_1[start_1, end_1), \text{ or } vector_2[start_2, end_2).$$

`vector-heap-sort`—**vector heap sort**

- `(vector-heap-sort <` *vector [start [end]]*`)` $\rightarrow$ *vector*
- `(vector-heap-sort! <` *vector [start [end]]*`)`

These procedures sort their data using heap sort, which is not a stable sorting algorithm.

`Vector-heap-sort` returns a vector of length $end - start$. `Vector-heap-sort!` is in-place, leaving its result in $vector[start, end)$.

`vector-quick-sort`—**vector quick sort**

- `(vector-quick-sort <` *vector [start [end]]*`)` $\rightarrow$ *vector*
- `(vector-quick-sort! <` *vector [start [end]]*`)`

These procedures sort their data using quick sort, which is not a stable sorting algorithm.

`Vector-quick-sort` returns a vector of length $end - start$. `Vector-quick-sort!` is in-place, leaving its result in $vector[start, end)$.

`vector-quick-sort3`—**vector quick sort with 3-way comparisons**

- `(vector-quick-sort3` *comp vector [start [end]]*`)` $\rightarrow$ *vector*
- `(vector-quick-sort3!` *comp vector [start [end]]*`)`

These procedures sort their data using quick sort, which is not a stable sorting algorithm.

`Vector-quick-sort3` returns a vector of length $end - start$. `Vector-quick-sort3!` is in-place, leaving its result in $vector[start, end)$.

These procedures implement a variant of quick-sort that takes a three-way comparison procedure C. C compares a pair of elements and returns an exact integer whose sign indicates their relationship:

$$\begin{aligned} (Cxy) < 0 &\quad\Rightarrow\quad x < y \\ (Cxy) = 0 &\quad\Rightarrow\quad x = y \\ (Cxy) > 0 &\quad\Rightarrow\quad x > y \end{aligned}$$

To help remember the relationship between the sign of the result and the relation, use the procedure $-$ as the model for C: $(-xy) < 0$ means that $x < y$; $(-xy) > 0$ means that $x > y$.

`vector-insert-sort`—**vector insertion sort**

- `(vector-insert-sort` $<$ *vector [start [end]]*`)` $\rightarrow$ *vector*
- `(vector-insert-sort!` $<$ *vector [start [end]]*`)`

These procedures stably sort their data using insertion sort.

- `Vector-insert-sort` returns a vector of length $end - start$.
- `Vector-insert-sort!` is in-place, leaving its result in $vector[start, end)$.

`delete-neighbor-duplicates`—**list and vector delete neighbor duplicates**

- `(list-delete-neighbor-dups` $=$ *list*`)` $\rightarrow$ *list*
- `(list-delete-neighbor-dups!` $=$ *list*`)` $\rightarrow$ *list*
- `(vector-delete-neighbor-dups` $=$ *vector [start [end]]*`)` $\rightarrow$ *vector*
- `(vector-delete-neighbor-dups!` $=$ *vector [start [end]]*`)` $\rightarrow$ end'

These procedures delete adjacent duplicate elements from a list or a vector, using a given element-equality procedure =. The first/leftmost element of a run of equal elements is the one that survives. The list or vector is not otherwise disordered.

These procedures are linear time—much faster than the $O(n^2)$ general duplicate-element deletors that do not assume any "bunching" of elements (such as the ones provided by SRFI 1). If you want to delete duplicate elements from a large list or vector, you can sort the elements to bring equal items together, then use one of these procedures, for a total time of $O(n \log(n))$.

The comparison procedure = passed to these procedures is always applied

(= *x* *y*)

where *x* comes before *y* in the containing list or vector.

- `List-delete-neighbor-dups` does not alter its input list; its answer may share storage with the input list.
- `Vector-delete-neighbor-dups` does not alter its input vector, but rather allocates a fresh vector to hold the result.
- `List-delete-neighbor-dups!` is permitted, but not required, to mutate its input list in order to construct its answer.
- `Vector-delete-neighbor-dups!` reuses its input vector to hold the answer, packing its answer into the index range [*start*, *end'*), where *end'* is the non-negative exact integer returned as its value. It returns *end'* as its result. The vector is not altered outside the range [*start*, *end'*).

Examples:

```
(list-delete-neighbor-dups = '(1 1 2 7 7 7 0 -2 -2))
  ⟹ (1 2 7 0 -2)

(vector-delete-neighbor-dups = '#(1 1 2 7 7 7 0 -2 -2))
  ⟹ #(1 2 7 0 -2)

(vector-delete-neighbor-dups = '#(1 1 2 7 7 7 0 -2 -2) 3 7)
  ⟹ #(7 0 -2)

;; Result left in v[3,9):
(let ((v (vector 0 0 0 1 1 2 2 3 3 4 4 5 5 6 6)))
  (cons (vector-delete-neighbor-dups! = v 3)
        v))
   ⟹ (9 . #(0 0 0 1 2 3 4 5 6 4 4 5 5 6 6))
```

`binary-searches`—**vector binary search**

- (`vector-binary-search` < *elt->key key vector [start [end]]*) → *integer or* `#f`
- (`vector-binary-search3` *compare-proc vector [start [end]]*) → *integer or* `#f`

`vector-binary-search` searches *vector* in range [*start*, *end*) (which default to 0 and the length of *vector*, respectively) for an element whose associated key is equal to *key*. The procedure *elt->key* is used to map an element to its associated key. The elements of the vector are assumed to be ordered by the < relation on these keys. That is,

```
(vector-sorted? (lambda (x y) (< (elt->key x) (elt->key y)))
                vector start end) ⟹ true
```

An element *e* of *vector* is a match for *key* if it's neither less nor greater than the key:

```
(and (not (< (elt->key e) key))
     (not (< key (elt->key e))))
```

If there is such an element, the procedure returns its index in the vector as an exact integer. If there is no such element in the searched range, the procedure returns false.

```
(vector-binary-search < car 4 '#((1 . one) (3 . three)
                                 (4 . four) (25 . twenty-five)))
⟹ 2

(vector-binary-search < car 7 '#((1 . one) (3 . three)
                                 (4 . four) (25 . twenty-five)))
⟹ #f
```

`Vector-binary-search3` is a variant that uses a three-way comparison procedure *compare-proc*. *Compare-proc* compares its parameter to the search key, and returns an exact integer whose sign indicates its relationship to the search key.

$$\begin{array}{lccccl} (\textit{compare-proc}\ x) & < & 0 & \Rightarrow & x & < & \textit{search-key} \\ (\textit{compare-proc}\ x) & = & 0 & \Rightarrow & x & = & \textit{search-key} \\ (\textit{compare-proc}\ x) & > & 0 & \Rightarrow & x & > & \textit{search-key} \end{array}$$

```
(vector-binary-search3 (lambda (elt) (- (car elt) 4))
                       '#((1 . one) (3 . three)
                          (4 . four) (25 . twenty-five)))
⟹ 2
```

5.20.3 Algorithmic properties

Different sort and merge algorithms have different properties. Choose the algorithm that matches your needs:

Vector insert sort Stable, but only suitable for small vectors—$O(n^2)$.

Vector quick sort Not stable. Is fast on average—$O(n \log(n))$—but has bad worst-case behaviour. Has good memory locality for big vectors (unlike heap sort). A clever pivot-picking trick (median of three samples) helps avoid worst-case behaviour, but pathological cases can still blow up.

Vector heap sort Not stable. Guaranteed fast—$O(n \log(n))$ *worst* case. Poor locality on large vectors. A very reliable workhorse.

Vector merge sort Stable. Not in-place—requires a temporary buffer of equal size. Fast—$O(n \log(n))$—and has good memory locality for large vectors.

The implementation of vector merge sort provided by this implementation is, additionally, a "natural" sort, meaning that it exploits existing order in the input data, providing $O(n)$ best case.

Destructive list merge sort Stable, fast and in-place (i.e., allocates no new cons cells). "Fast" means $O(n \log(n))$ worst-case, and substantially better if the data is already mostly ordered, all the way down to linear time for a completely-ordered input list (i.e., it is a "natural" sort).

Note that sorting lists involves chasing pointers through memory, which can be a loser on modern machine architectures because of poor cache and page locality. Sorting vectors has inherently better locality.

This implementation's destructive list merge and merge sort implementations are opportunistic—they avoid redundant `set-cdr!`s, and try to take long already-ordered runs of list structure as-is when doing the merges.

Pure list merge sort Stable and fast—$O(n \log(n))$ worst-case, and possibly $O(n)$, depending upon the input list (see discussion above).

Algorithm	Stable?	Worst case	Average case	In-place
Vector insert	Yes	$O(n^2)$	$O(n^2)$	Yes
Vector quick	No	$O(n^2)$	$O(n \log(n))$	Yes
Vector heap	No	$O(n \log(n))$	$O(n \log(n))$	Yes
Vector merge	Yes	$O(n \log(n))$	$O(n \log(n))$	No
List merge	Yes	$O(n \log(n))$	$O(n \log(n))$	Either

5.21 Regular expressions

This section describes a functional interface for building regular expressions and matching them against strings. The matching is done using the POSIX regular expression package. Regular expressions are in the structure `regexps`.

A regular expression is either a character set, which matches any character in the set, or a composite expression containing one or more subexpressions. A regular expression can be matched against a string to determine success or failure, and to determine the substrings matched by particular subexpressions.

• `(regexp?` *value*`)` → *boolean*

Returns `#t` if *value* is a regular expression created using the functional interface for regular expressions, and `#f` otherwise.

5.21.1 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` *character-or-string ...*`)` → *char-set*
- `(range` *low-char high-char*`)` → *char-set*
- `(ranges` *low-char high-char ...*`)` → *char-set*
- `(ascii-range` *low-char high-char*`)` → *char-set*
- `(ascii-ranges` *low-char high-char ...*`)` → *char-set*

`Set` returns a set that contains the character arguments and the characters in any string arguments. `Range` returns a character set that contain all characters between *low-char* and *high-char*, inclusive. `Ranges` returns a set that contains all characters in the given ranges. `Range` and `ranges` use the ordering induced by `char->integer`. `Ascii-range` and `ascii-ranges` use the ASCII ordering. It is an error for a *high-char* to be less than the preceding *low-char* in the appropriate ordering.

- `(negate` *char-set*`)` → *char-set*
- `(intersection` *char-set char-set*`)` → *char-set*
- `(union` *char-set char-set*`)` → *char-set*
- `(subtract` *char-set char-set*`)` → *char-set*

These perform the indicated operations on character sets.

The following character sets are predefined:

```
lower-case    (set "abcdefghijklmnopqrstuvwxyz")
upper-case    (set "ABCDEFGHIJKLMNOPQRSTUVWXYZ")
alphabetic    (union lower-case upper-case)
numeric       (set "0123456789")
alphanumeric  (union alphabetic numeric)
punctuation   (set "!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~")
graphic       (union alphanumeric punctuation)
printing      (union graphic (set #\space))
control       (negate printing)
blank         (set #\space (ascii->char 9)) ; 9 is tab
whitespace    (union (set #\space) (ascii-range 9 13))
hexdigit      (set "0123456789abcdefABCDEF")
```

The above are taken from the default locale in POSIX. The characters in `whitespace` are *space*, *tab*, *newline* (= *line feed*), *vertical tab*, *form feed*, and *carriage return*.

5.21.2 Anchoring

- `(string-start)` → *reg-exp*
- `(string-end)` → *reg-exp*

`String-start` returns a regular expression that matches the beginning of the string being matched against; string-end returns one that matches the end.

5.21.3 Composite expressions

- `(sequence` *reg-exp ...*`)` → *reg-exp*
- `(one-of` *reg-exp ...*`)` → *reg-exp*

`Sequence` matches the concatenation of its arguments, `one-of` matches any one of its arguments.

- `(text` *string*`)` → *reg-exp*

`Text` returns a regular expression that matches the characters in *string*, in order.

- `(repeat` *reg-exp*`)` → *reg-exp*
- `(repeat` *count reg-exp*`)` → *reg-exp*
- `(repeat` *min max reg-exp*`)` → *reg-exp*

`Repeat` returns a regular expression that matches zero or more occurrences of its *reg-exp* argument. With no count the result will match any number of times (*reg-exp**). With a single count the returned expression will match *reg-exp* 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* and *min* should be exact, non-negative integers; *max* should either be an exact non-negative integer or `#f`.

5.21.4 Case sensitivity

Regular expressions are normally case-sensitive.

- `(ignore-case` *reg-exp*`)` → *reg-exp*
- `(use-case` *reg-exp*`)` → *reg-exp*

The value returned by `ignore-case` is identical its argument except that 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 later uses of the these procedures. By way of example, the following matches `"ab"` but not `"aB"`, `"Ab"`, or `"AB"`.

```
(text "ab")
```

while

```
(ignore-case (test "ab"))
```

matches `"ab"`, `"aB"`, `"Ab"`, and `"AB"` and

```
(ignore-case (sequence (text "a")
                       (use-case (text "b"))))
```

matches `"ab"` and `"Ab"` but not `"aB"` or `"AB"`.

5.21.5 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 be each successful submatch.

- `(submatch` *key reg-exp*`)` → *reg-exp*
- `(no-submatches` *reg-exp*`)` → *reg-exp*

`Submatch` returns a regular expression that matches its argument and causes the result of matching its argument to be reported by the `match` procedure. *Key* is used to indicate the result of this particular submatch in the alist of successful submatches returned by `match`. Any value may be used as a *key*. `No-submatches` returns an expression identical to its argument, except that all submatches have been elided.

- `(any-match?` *reg-exp string*`)` → *boolean*
- `(exact-match?` *reg-exp string*`)` → *boolean*
- `(match` *reg-exp string*`)` → *match or* `#f`
- `(match-start` *match*`)` → *index*
- `(match-end` *match*`)` → *index*
- `(match-submatches` *match*`)` → *alist*

`Any-match?` returns `#t` if *string* matches *reg-exp* or contains a substring that does, and `#f` otherwise. `Exact-match?` returns `#t` if *string* matches *reg-exp* and `#f` otherwise.

`Match` returns `#f` if *reg-exp* does not match *string* and a match record if it does match. A match record contains three values: the beginning and end of the substring that matched the pattern and an a-list of submatch keys and corresponding match records for any submatches that also matched. `Match-start` returns the index of the first character in the matching substring and `match-end` gives index of the first character after the matching substring. `Match-submatches` returns an alist of submatch keys and match records. Only the top match record returned by `match` has a submatch alist.

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 *reg-exp*. Subsequent calls with the same *reg-exp* will be more efficient.

The C interface to the POSIX regular expression code uses ASCII `nul` as an end-of-string marker. The matching procedures will ignore any characters following an embedded ASCII `nul`s in *string*.

```
(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

(match pattern "abc")              → #{match 0 3}
(match pattern "abx")              → #f
(match pattern "xxabcxx")          → #{match 2 5}

(let ((x (match (sequence (text "ab")
                          (submatch 'foo (text "cd"))
                          (text "ef"))
                "xxxabcdefxx")))
  (list x (match-submatches x)))
  → (#{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}))
```

5.22 SRFIs

'SRFI' stands for 'Scheme Request For Implementation'. An SRFI is a description of an extension to standard Scheme. Draft and final SRFI documents, a FAQ, and other information about SRFIs can be found at `http://srfi.schemers.org`.

Scheme 48 includes implementations of the following (final) SRFIs:

- SRFI 1 – List Library
- SRFI 2 – `and-let*`
- SRFI 4 – Homogeneous numeric vector datatypes (see note below)
- SRFI 5 – `let` with signatures and rest arguments
- SRFI 6 – Basic string ports
- SRFI 7 – Program configuration
- SRFI 8 – `receive`
- SRFI 9 – Defining record types
- SRFI 11 – Syntax for receiving multiple values

- SRFI 13 – String Library
- SRFI 14 – Character-Set Library (see note below)
- SRFI 16 – Syntax for procedures of variable arity
- SRFI 17 – Generalized `set!`
- SRFI 19 – Time Data Types and Procedures
- SRFI 22 – Running Scheme Scripts on Unix
- SRFI 23 – Error reporting mechanism
- SRFI 25 – Multi-dimensional Array Primitives
- SRFI 26 – Notation for Specializing Parameters without Currying
- SRFI 27 – Sources of Random Bits
- SRFI 28 – Basic Format Strings
- SRFI 31 – A special form `rec` for recursive evaluation
- SRFI 34 – Exception Handling for Programs
- SRFI 37 – args-fold: a program argument processor
- SRFI 40 – A Library of Streams
- SRFI 42 – Eager Comprehensions
- SRFI 43 – Vector library
- SRFI 45 – Primitives for Expressing Iterative Lazy Algorithms
- SRFI 60 – Integers as Bits
- SRFI 61 – A more general cond clause
- SRFI 62 – S-expression comments
- SRFI 63 – Homogeneous and Heterogeneous Arrays
- SRFI 66 – Octet Vectors
- SRFI 67 – Compare Procedures
- SRFI 74 – Octet-Addressed Binary Blocks
- SRFI 78 – Lightweight testing

Documentation on these can be found at the web site mentioned above.

SRFI 4 specifies an external representation for homogeneous numeric vectors that is incompatible with R^{5}RS. The Scheme 48 version of SRFI 4 does not support this external representation.

SRFI 14 includes the procedure `->char-set` which is not a standard Scheme identifier (in R^{5}RS the only required identifier starting with `-` is `-` itself). In the Scheme 48 version of SRFI 14 we have renamed `->char-set` as `x->char-set`.

SRFI bindings can be accessed either by opening the appropriate structure (the structure `srfi-`*n* contains SRFI *n*) or by loading structure `srfi-7` and then using the `,load-srfi-7-program` command to load an SRFI 7-style program. The syntax for the command is

`,load-srfi-7-program` *name filename*

This creates a new structure and associated package, binds the structure to *name* in the configuration package, and then loads the program found in *filename* into the package.

As an example, if the file `test.scm` contains

```
(program (code (define x 10)))
```

this program can be loaded as follows:

```
> ,load-package srfi-7
> ,load-srfi-7-program test test.scm
[test]
> ,in test
test> x
10
test>
```

Chapter 6

Unicode

Scheme 48 fully supports ISO 10646 (Unicode): Scheme characters represent Unicode scalar values, and Scheme strings are arrays of scalar values. More information on Unicode can be found at `http://www.unicode.org/`.

6.1 Characters and their codes

Scheme 48 internally represents characters as Unicode scalar values. The `unicode` structure contains procedures for converting between characters and scalar values:

- `(char->scalar-value` *char*`)` $\rightarrow$ *integer*
- `(scalar-value->char` *integer*`)` $\rightarrow$ *char*
- `(scalar-value?` *integer*`)` $\rightarrow$ *boolean*

`Char->scalar-value` returns the scalar value of a character, and `scalar-value->char` converts in the other direction. `Scalar-value->char` signals an error if passed an integer that is not a scalar value.

Note that the Unicode scalar value range is

$$[0, \#xD7FF] \cup [\#xE000, \#x10FFFF]$$

In particular, this excludes the surrogates, which UTF-16 uses to encode scalar values with two 16-bit words. Note that this representation differs from that of Java, which uses UTF-16 code units as the character representation—Scheme 48 effectively uses UTF-32, and is thus in line with other Scheme implementations and the current Unicode proposal for R^{6}RS, as set forth in SRFI 75.

The R^{5}RS procedures `char->integer` and `integer->char` are synonyms for `char->scalar-value` and `scalar-value->char`, respectively.

6.2 Character and string literals

The syntax specified here is in line with the current Unicode proposal for R^{6}RS, as set forth in SRFI 75, except for case-sensitivity. (Scheme 48 is case-insensitive.)

6.2.1 Character literals

The following character names are available in addition to what R^{5}RS provides:

- `#\nul` (ASCII 0)
- `#\alarm` (ASCII 7)
- `#\backspace` (ASCII 8)
- `#\tab` (ASCII 9)
- `#\vtab` (ASCII 11)
- `#\page` (ASCII 12)
- `#\return` (ASCII 13)
- `#\esc` (ASCII 27)
- `#\rubout` (ASCII 127)
- `#\x`⟨x⟩⟨x⟩... hex, explicitly or implicitly delimited, where ⟨x⟩⟨x⟩... denotes the scalar value of the character

6.2.2 String literals

The following escape characters in string literals are available in addition to what R^{5}RS provides:

- `\a`: alarm (ASCII 7)
- `\b`: backspace (ASCII 8)
- `\t`: tab (ASCII 9)
- `\n`: linefeed (ASCII 10)
- `\v`: vertical tab (ASCII 11)
- `\f`: formfeed (ASCII 12)
- `\r`: return (ASCII 13)
- `\e`: escape (ASCII 27)
- `\'`: quote (ASCII 39, same as unquoted)

- \⟨newline⟩⟨intraline whitespace⟩: elided (allows a single-line string to span source lines)
- `\x`⟨x⟩⟨x⟩. . . `;` hex, where ⟨x⟩⟨x⟩. . . denotes the scalar value of the character

6.2.3 Identifiers and symbol literals

Where R^{5}RS allows a ⟨letter⟩, Scheme 48 allows in addition any character whose scalar value is greater than 127 and whose Unicode general category is Lu, Ll, Lt, Lm, Lo, Mn, Mc, Me, Nd, Nl, No, Pd, Pc, Po, Sc, Sm, Sk, So, or Co.

Moreover, when a backslash appears in a symbol, it must start a `\x`⟨x⟩⟨x⟩. . . `;` escape, which identifies an arbitrary character to include in the symbol. Note that a backslash itself can be specified as `\x5C;`.

6.3 Character classification and case mappings

The R^{5}RS character predicates—`char-whitespace?`, `char-lower-case?`, `char-upper-case?`, `char-numeric?`, and `char-alphabetic?`—all treat the full Unicode range.

`Char-upcase` and `char-downcase` as well as `char-ci=?`, `char-ci<?`, `char-ci<=?`, `char-ci>?`, `char-ci>=?`, `string-ci=?`, `string-ci<?`, `string-ci>?`, `string-ci<=?`, `string-ci>=?` all use the standard simple locale-insensitive Unicode case folding.

In addition, Scheme 48 provides the `unicode-char-maps` structure for more complete access to the Unicode character classification with the following procedures and macros:

- `(general-category` *general-category-name*`)` → *general-category* syntax
- `(general-category?` *x*`)` → *boolean*
- `(general-category-id` *general-category*`)` → *string*
- `(char-general-category` *char*`)` → *general-category*

The syntax `general-category` returns a Unicode general category object associated with *general-category-name*. (See Figure 6.1 below.) `General-category?` is the predicate for general-category objects. `General-category-id` returns the Unicode category id as a string (also listed in Figure 6.1). `Char-general-category` returns the general category of a character.

- `(general-category-primary-category` *general-category*`)` → *primary-category*
- `(primary-category` *primary-category-name*`)` → *primary-category* syntax
- `(primary-category?` *x*`)` → *boolean*

`General-category-primary-category` maps the general category to its associated primary category—also listed in Figure 6.1. The `primary-category` syntax returns the primary-category object associated with *primary-category-name*. `Primary-category?` is the predicate for primary-category objects.

The `unicode-char-maps` procedure also provides the following additional case-mapping procedures for characters:

- `(char-titlecase?` *char*`)` → *boolean*
- `(char-titlecase` *char*`)` → *char*
- `(char-foldcase` *char*`)` → *char*

`Char-titlecase?` tests if a character is in titlecase. `Char-titlecase` returns the titlecase counterpart of a character. `Char-foldcase` folds the case of a character, i.e. maps it to uppercase first, then to lowercase. The following case-mapping procedures on strings are available:

- `(string-upcase` *string*`)` → *string*
- `(string-downcase` *string*`)` → *string*
- `(string-titlecase` *string*`)` → *string*
- `(string-foldcase` *string*`)` → *string*

These implement the simple case mappings defined by the Unicode standard—note that the length of the output string may be different from that of the input string.

6.4 SRFI 14

The SRFI 14 ("Character Sets") implementation in the `srfi-14` structure is fully Unicode-compliant.

6.5 R6RS

The `r6rs-unicode` structure exports the procedures from the `(r6rs unicode)` library of 5.91 draft of R^{6}RS that are not already in the `scheme` structure:

```
string-normalize-nfd
string-normalize-nfkd
string-normalize-nfc
string-normalize-nfkc
char-titlecase
char-title-case?
char-foldcase
string-upcase
string-downcase
string-foldcase
string-titlecase
```

The `r6rs-unicode` structure also exports a `char-general-category` procedure compatible with the `(r6rs unicode)` library. Note that, as Scheme 48 treats source code case-insensitively, the symbols it returns are all-lowercase.

6.6 I/O

Ports must encode any text a program writes to an output port to a byte sequence, and conversely decode byte sequences when a program reads text from an input port. Therefore, each port has an associated *text codec* that describes how encode and decode text.

Note that the interface to the text codec functionality is experimental and very likely to change in the future.

6.6.1 Text codecs

The `i/o` structure defines the following procedures:

- `(port-text-codec` *port*`)` → *text-codec*
- `(set-port-text-codec!` *port text-codec*`)`

These two procedures retrieve and set the text codec associated with a port, respectively. A program can set text codec of a port at any time, even if it has already performed I/O on the port.

The `text-codecs` structure defines the following procedures and macros:

- `(text-codec?` *x*`)` → *boolean*
- `null-text-codec` text-codec
- `us-ascii-codec` text-codec
- `latin-1-codec` text-codec
- `utf-8-codec` text-codec
- `utf-16le-codec` text-codec
- `utf-16be-codec` text-codec
- `utf-32le-codec` text-codec
- `utf-32be-codec` text-codec
- `(find-text-codec` *string*`)` → *text-codec or* `#f`

`Text-codec?` is the predicate for text codecs. `Null-text-codec` is primarily meant for null ports that never yield input and swallow all output. The following text codecs implement the US-ASCII, Latin-1, Unicode UTF-8, Unicode UTF-16 (little-endian), Unicode UTF-16 (big-endian), Unicode UTF-32 (little-endian), Unicode UTF-32 (big-endian) encodings, respectively.

`Find-text-codec` finds the codec associated with an encoding name. The names of the above encodings are `"null"`, `"US-ASCII"`, `"ISO8859-1"`, `"UTF-8"`, `"UTF-16LE"`, `"UTF-16BE"`, `"UTF-32LE"`, and `"UTF-32BE"`, respectively.

6.6.2 Text-codec utilities

The `text-codec-utils` structure exports a few utilities for dealing with text codecs:

- `(guess-port-text-codec-according-to-bom` *port*`)` → *text-codec or* `#f`
- `(set-port-text-codec-according-to-bom!` *port*`)` → *boolean*

These procedures look at the byte-order-mark (also called the "BOM", `U+FEFF`) at the beginning of a port and guess the appropriate text codec. This works only for UTF-16 (little-endian and big-endian) and UTF-8. `Guess-port-text-codec-according-to-bom` returns the text codec, or `#f` if it found no UTF-16 or UTF-8 BOM. Note that this actually reads from the port. If the guess does not succeed, it is probably a good idea to re-open the port. `Set-port-text-codec-according-to-bom!` calls `guess-port-text-codec-according-to-bom`, sets the port text codec to the result if successful and returns `#t`. If it is not successful, it returns `#f`. As with `guess-port-text-codec-according-to-bom`, this reads from the port, whether successful or not.

6.6.3 Creating text codecs

- `(make-text-codec` *strings encode-proc decode-proc*`)` → *text-codec*
- `(text-codec-names` *text-codec*`)` → *list of strings*
- `(text-codec-encode-char-proc` *text-codec*`)` → *encode-proc*
- `(text-codec-decode-char-proc` *text-codec*`)` → *decode-proc*
- `(define-text-codec` *id name encode-proc decode-proc*`)` syntax
- `(define-text-codec` *id* `(`*name* `...)` *encode-proc decode-proc*`)` syntax

`Make-text-codec` constructs a text codec from a list of names, and an encode and a decode procedure. (See below on how to construct encode and decode procedures.) `Text-codec-names`, `text-codec-encode-char-proc`, and `text-codec-decode-char-proc` are the accessors for text codec. The `define-text-codec` is a shorthand for binding a global identifier to a text codec. Its first form is for codecs with only one name, the second for codecs with several names.

Encoding and decoding procedures work as follows:

- `(`*encode-proc char buffer start count*`)` → *boolean maybe-count*
- `(`*decode-proc buffer start count*`)` → *maybe-char count*

An *encode-proc* consumes a character *char* to encode, a byte vector *buffer* to receive the encoding, an index *start* into the buffer, and a block size *count*. It is supposed to encode the bytes into the block at $[start, start + count)$. If the encoding is successful, the procedure must return `#t` and the number of bytes needed by the encoding. If the character cannot be encoded at all, the procedure must return `#f` and `#f`. If the encoding is possible but the space is not sufficient, the procedure must return `#f` and a total number of bytes needed for the encoding.

A *decode-proc* consumes a byte vector *buffer*, an index *start* into the buffer, and a block size *count*. It is supposed to decode the bytes at indices $[start, start + count)$. If the decoding is successful, it must return the decoded character at the beginning of the block, and the number of bytes consumed. If the block cannot begin with or be a prefix of a valid encoding, the procedure must return `#f` and `#f`. If the block contains a true prefix of a valid encoding, the procedure must return `#f` and a total count of bytes (including the buffer) needed to complete the encoding. Note that this byte count is only a guess: the system will provide that many bytes, but the decoding procedures might still signal an incomplete encoding, causing the system to try to obtain more.

6.7 Default encodings

The default encoding for new ports is UTF-8. For the default `current-input-port`, `current-output-port`, and `current-error-port`, Scheme 48 consults the OS for encoding information.

For Unix, it consults `nl_langinfo(3)`, which in turn consults the `LC_` environment variables. If the encoding is not defined that way, Scheme 48 reverts to US-ASCII.

Under Windows, Scheme 48 uses Unicode I/O (using UTF-16) for the default ports connected to the console, and Latin-1 for default ports that are not.

general-category-name	*primary-category-name*	Unicode category id
`uppercase-letter`	`letter`	`"Lu"`
`lowercase-letter`	`letter`	`"Ll"`
`titlecase-letter`	`letter`	`"Lt"`
`modified-letter`	`letter`	`"Lm"`
`other-letter`	`letter`	`"Lo"`
`non-spacing-mark`	`mark`	`"Mn"`
`combining-spacing-mark`	`mark`	`"Mc"`
`enclosing-mark`	`mark`	`"Me"`
`decimal-digit-number`	`number`	`"Nd"`
`letter-number`	`number`	`"Nl"`
`other-number`	`number`	`"No"`
`opening-punctuation`	`punctuation`	`"Ps"`
`closing-punctuation`	`punctuation`	`"Pe"`
`initial-quote-punctuation`	`punctuation`	`"Pi"`
`final-quote-punctuation`	`punctuation`	`"Pf"`
`dash-punctuation`	`punctuation`	`"Pd"`
`connector-punctuation`	`punctuation`	`"Pc"`
`other-punctuation`	`punctuation`	`"Po"`
`currency-symbol`	`symbol`	`"Sc"`
`mathematical-symbol`	`symbol`	`"Sm"`
`modifier-symbol`	`symbol`	`"Sk"`
`other-symbol`	`symbol`	`"So"`
`space-separator`	`separator`	`"Zs"`
`paragraph-separator`	`separator`	`"Zp"`
`line-separator`	`separator`	`"Zl"`
`control-character`	`miscellaneous`	`"Cc"`
`formatting-character`	`miscellaneous`	`"Cf"`
`surrogate`	`miscellaneous`	`"Cs"`
`private-use-character`	`miscellaneous`	`"Co"`
`unassigned`	`miscellaneous`	`"Cn"`

Figure 6.1: Unicode general categories and primary categories

Chapter 7

Threads

This chapter describes Scheme 48's thread system: Scheme 48 threads are fully preemptive; all threads (currently) run within a single operating system process. Scheme 48 allows writing customized, nested schedulers, and provides numerous facilities for the synchronization of shared-memory programs, most importantly *proposals* for optimistic concurrency.

7.1 Creating and controlling threads

The bindings described in this section are part of the `threads` structure.

- `(spawn` *thunk*`)` $\rightarrow$ *thread*
- `(spawn` *thunk name*`)` $\rightarrow$ *thread*

`Spawn` creates a new thread, passes that thread to the current scheduler, and instructs the scheduler to run *thunk* in that thread. The *name* argument (a symbol) associates a symbolic name with the thread; it is purely for debugging purposes.

- `(relinquish-timeslice)`
- `(sleep` *time-in-milliseconds*`)`
- `(terminate-current-thread)`

`Relinquish-timeslice` instructs the scheduler to run another thread, thus relinquishing the timeslice of the current thread. `Sleep` does the same and asks the scheduler to suspend the current thread for at least *time-in-milliseconds* milliseconds before resuming it. Finally, `terminate-current-thread` terminates the current thread.

Each thread is represented by a thread object. The following procedures operate on that object:

- `(current-thread)` $\rightarrow$ *thread*
- `(thread?` *thing*`)` $\rightarrow$ *boolean*

- `(thread-name` *thread*`)` → *name*
- `(thread-uid` *thread*`)` → *integer*

`Current-thread` returns the thread object associated with the currently running thread. `Thread?` is the predicate for thread objects. `Thread-name` extracts the name of the thread, if one was specified in the call to `spawn`, `#f` otherwise. `Thread-uid` returns the *uid* of the thread, a unique integer assigned by the thread system.

7.2 Advanced thread handling

The following bindings are part of the `threads-internal` structure:

- `(terminate-thread!` *thread*`)`
- `(kill-thread!` *thread*`)`

`Terminate-thread!` unwinds the thread associated with *thread*, running any pending `dynamic-wind` *after* thunks (in that thread), after which the thread terminates. `Kill-thread!` causes the thread associated with *thread* to terminate immediately without unwinding its continuation.

7.3 Debugging multithreaded programs

Debugging multithreaded programs can be difficult.

As described in section 3.12, when any thread signals an error, Scheme 48 stops running all of the threads at that command level.

The following procedure (exported by the structure `debug-messages`) is useful in debugging multi-threaded programs.

- `(debug-message` $element_0$ ...`)`

`Debug-message` prints the elements to '`stderr`', followed by a newline. The only types of values that `debug-message` prints in full are small integers (fixnums), strings, characters, symbols, booleans, and the empty list. Values of other types are abbreviated as follows:

pair	`(...)`
vector	`#(...)`
procedure	`#{procedure}`
record	`#{<name of record type>}`
all others	`???`

The great thing about `debug-message` is that it bypasses Scheme 48's I/O and thread handling. The message appears immediately, with no delays or errors.

7.4 Optimistic concurrency

Most of the bindings described in this section are part of the `proposals` structure—the low-level bindings described at the very end of the section are part of the `low-proposals` structure.

A *proposal* is a record of reads from and and writes to locations in memory. Each thread has an associated *current proposal* (which may be `#f`). The *logging* operations listed below record any values read or written in the current proposal. A reading operation, such as `provisional-vector-ref`, first checks to see if the current proposal contains a value for the relevant location. If so, that value is returned as the result of the read. If not, the current contents of the location are stored in the proposal and then returned as the result of the read. A logging write to a location stores the new value as the current contents of the location in the current proposal; the contents of the location itself remain unchanged.

Committing to a proposal verifies that any reads logged in the proposal are still valid and, if so, performs any writes that the proposal contains. A logged read is valid if, at the time of the commit, the location contains the same value it had at the time of the original read (note that this does not mean that no change occurred, simply that the value now is the same as the value then). If a proposal has an invalid read then the effort to commit fails; no change is made to the value of any location. The verifications and subsequent writes to memory are performed atomically with respect to other proposal commit attempts.

The `queues` structure (with source in `scheme/big/queue.scm`) is a thoroughly commented example of a moderately complex data structure made thread-safe using optimistic concurrency.

- `(call-ensuring-atomicity` *thunk*`)` → *value ...*
- `(call-ensuring-atomicity!` *thunk*`)`
- `(ensure-atomicity` *exp* `...)` → *value ...* syntax
- `(ensure-atomicity!` *exp* `...)` syntax

If there is a proposal in place `call-ensuring-atomicity` and `call-ensuring-atomicity!` simply make a (tail-recursive) call to *thunk*. If the current proposal is `#f` they create a new proposal, install it, call *thunk*, and then try to commit to the proposal. This process repeats, with a new proposal on each iteration, until the commit succeeds. `Call-ensuring-atomicity` returns whatever values are returned by *thunk* on its final invocation, while `ensure-atomicity!` discards any such values and returns nothing.

`Ensure-atomicity` and `ensure-atomicity!` are macro versions of `call-ensuring-atomicity` and `call-ensuring-atomicity!`: `(ensure-atomicity` *exp* `...)` expands into `(call-ensuring-atomicity (lambda ()` *exp* `...))`; likewise for `ensure-atomicity!` and `call-ensuring-atomicity!`.

- `(provisional-car` *pair*`)` → *value*
- `(provisional-cdr` *pair*`)` → *value*
- `(provisional-set-car!` *pair value*`)`
- `(provisional-set-cdr!` *pair value*`)`
- `(provisional-cell-ref` *cell*`)` → *value*

- `(provisional-cell-set!` *cell value*`)`
- `(provisional-vector-ref` *vector i*`)` → *value*
- `(provisional-vector-set!` *vector i value*`)`
- `(provisional-string-ref` *vector i*`)` → *char*
- `(provisional-string-set!` *vector i char*`)`
- `(provisional-byte-vector-ref` *vector i*`)` → *k*
- `(provisional-byte-vector-set!` *vector i k*`)`

These are all logging versions of their Scheme counterparts. Reads are checked when the current proposal is committed and writes are delayed until the commit succeeds. If the current proposal is `#f` these perform exactly as their Scheme counterparts.

The following implementation of a simple counter may not function properly when used by multiple threads.

```
(define (make-counter)
  (let ((value 0))
    (lambda ()
      (set! value (+ value 1))
      value)))
```

Here is the same procedure using a proposal to ensure that each increment operation happens atomically. The value of the counter is kept in a cell (see section 5.6) to allow the use of logging operations.

```
(define (make-counter)
  (let ((value (make-cell 0)))
    (lambda ()
      (ensure-atomicity
        (lambda ()
          (let ((v (+ (provisional-cell-ref value)
                      1)))
            (provisional-cell-set! value v)
            v))))))
```

Because `ensure-atomicity` creates a new proposal only if there is no existing proposal in place, multiple atomic actions can be merged into a single atomic action. For example, the following procedure increments an arbitrary number of counters at the same time. This works even if the same counter appears multiple times; `(step-counters! c0 c0)` would add two to the value of counter `c0`.

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

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

This is the same as `define-record-type` except all field reads and writes are logged in the current proposal. If the optional list of field tags is present then only those fields will be logged.

- `(call-atomically` *thunk*`)` → *value(s)*
- `(call-atomically!` *thunk*`)`
- `(atomically` *exp* `...)` → *value(s)* syntax
- `(atomically!` *exp* `...)` syntax

`Call-atomically` and `call-atomically!` are identical to `call-ensuring-atomicity` and `call-ensuring-atomicity!` except that they always install a new proposal before calling `thunk`. The current proposal is saved and then restored after `thunk` returns. `Call-atomically` and `call-atomically!` are useful if `thunk` contains code that is not to be combined with any other operation.

`Atomically` and `atomically!` are macro versions of `call-atomically` and `call-atomically!`: `(atomically` *exp* `...)` expands into `(call-atomically (lambda ()` *exp* `...))`; likewise for `atomically!` and `call-atomically!`.

The following procedures and macro are intended primarily for use in implementing new synchronization primitives or complex thread-safe data structures.

- `(with-new-proposal (`*lose*`)` *exp* `...)` → *value ...* syntax
- `(maybe-commit)` → *boolean*
- `(proposal-active?)` → *boolean*
- `(remove-current-proposal!)`
- `(invalidate-current-proposal!)`

`With-new-proposal` saves the current proposal, installs a new one, executes the forms in the body, reinstalls the formerly current proposal, and returns whatever the last body form returned. It also binds *lose* to a thunk repeating the procedure of installing a new procedure and running the body. Typically, the body will call `maybe-commit` and, if that fails, tail-call *lose* to try again. If *lose* is called from a non-tail position of the body, the results are unspecified (and probably harmful).

`Maybe-commit` verifies that any reads logged in the current proposal are still valid and, if so, performs any writes that it contains. A logged read is valid if, at the time of the commit, the location read contains the same value it had at the time of the original read (note that this does not mean that no change occurred, simply that the value now is the same as the value then). `Maybe-commit` returns `#t` if the commit succeeds and `#f` if it fails.

`Proposal-active?` returns `#t` if a proposal is active, and `#f` otherwise. `Remove-current-proposal!` removes and discards the current proposal; this can be used to clean up before raising an error. `Invalidate-current-proposal!` ensures that any attempt to commit the current proposal will fail; this can be used if an operation on a thread-safe data structure detects that it has seen the data structure in an inconsistent state.

The following procedures give access to the low-level proposal mechanism. They are defined in the `low-proposals` structure.

- `(make-proposal)` → *proposal*
- `(current-proposal)` → *proposal*
- `(set-current-proposal!` *proposal*`)`

`Make-proposal` creates a new proposal. `Current-proposal` and `set-current-proposal` access and set the current thread's proposal. It is an error to pass to `set-current-proposal!` a proposal that is already in use.

7.5 Condition variables

Condition variables (defined in the `condvars` structure) allow threads perform condition synchronization: It allows threads to block, waiting for a specified condition—associated with a condition variable—to occur, and other threads to wake up the waiting threads when the condition is fulfilled.

Note that, in Scheme 48, condition variables work in conjunction with proposals, not with mutex locks or semaphores, as in most other implementations of this concept.

- `(make-condvar)` → *condvar*
- `(make-condvar` *id*`)` → *condvar*
- `(condvar?` *thing*`)` → *boolean*
- `(set-condvar-has-value?!` *condvar boolean*`)`
- `(condvar-has-value?` *condvar*`)` → *boolean*
- `(set-condvar-value!` *condvar value*`)`
- `(condvar-value` *condvar*`)` → *value*
- `(maybe-commit-and-wait-for-condvar` *condvar*`)` → *boolean*
- `(maybe-commit-and-set-condvar!` *condvar value*`)` → *boolean*

`Make-condvar` creates a condition variable. (The optional *id* argument is only for debugging purposes; the discloser for condition variables prints it out if present.) `Condvar?` is the predicate for condition variables.

Each condition variable has an associated value and a flag `has-value?` signalling if the condition has already occured. The accessor for flag is `condvar-has-value?`; `set-condvar-has-value?!` sets it. Both are provisional operations and go through the current proposal. `Set-condvar-value!` sets the value of the condition variable (provisionally), and `condvar-value` extracts it.

`Maybe-commit-and-wait-for-condvar` attempts to commit the current proposal. If the commit succeeds, it suspends the current thread and registers it with the *condvar* condition variable. Upon waking up again `maybe-commit-and-wait-for-condvar` returns `#t`, If the commit fails, `maybe-commit-and-set-condvar` returns `#f`.

`Maybe-commit-and-set-condvar!` sets the value of the *condvar* condition variable to *value*, (provisionally) sets the `has-value?` flag to `#t`, and then attempt to commit the current proposal. Upon success, it wakes up all suspended threads registered with *condvar* and returns `#t`, otherwise, it returns `#f`.

7.6 Mutual exclusion

Scheme 48 also has more traditional mutual-exclusion synchronization abstractions, specifically mutex locks and placeholders. Note that typically synchronization via optimistic concurrency is usually preferable: Mutual exclusion often puts the running program into an inconsistent state for the time of the inclusion, which has adverse effects on modularity and interruptibility.

7.6.1 Locks

The `locks` structure contains bindings that implement standard mutex locks:

- `(make-lock)` $\rightarrow$ *lock*
- `(lock?` *thing*`)` $\rightarrow$ *boolean*
- `(obtain-lock` *lock*`)`
- `(maybe-obtain-lock` *lock*`)` $\rightarrow$ *boolean*
- `(release-lock` *lock*`)`

`Make-lock` creates a lock in the “released” state. `Lock?` is the predicate for locks.

`Obtain-lock` atomically checks if *lock* is in the “released” state. If it is, the lock is put into the “obtained” state, and `obtain-lock` returns immediately. If the lock is in the “obtained” state, the current thread is suspended and registered with the lock. `Maybe-obtain-lock`, like `obtain-lock`, checks the state of *lock*: if it is “released,” the lock is put into the “obtained” state, if it is “obtained,” `maybe-obtain-lock` returns immediately. `Maybe-obtain-lock` returns `#t` if it was able to obtain the lock, and `#f` otherwise.

`Release-lock` does nothing if *lock* is in the “released” state. If it is in the “obtained” state, `release-lock` causes one of the threads suspended on an `obtain-lock` lock operation to continue execution. If that thread is the last thread registered with the lock, the lock is transferred to the “released” state. In any case, `release-lock` returns immediately.

7.6.2 Placeholders

The `placeholders` structure contains bindings for *placeholders*—thread-safe, write-once variables, akin to ID-90 I-structures or CML I-variables.

The typical scenario for placeholders is that, say, a thread A computes a value needed by another thread B at some unspecified time. Both threads share access to a placeholder; when A has computed the value, it places it into the placeholder. When B needs the value, it extracts it from placeholder, blocking if necessary.

- `(make-placeholder)` → *placeholder*
- `(make-placeholder` *id*`)` → *placeholder*
- `(placeholder?` *thing*`)` → *boolean*
- `(placeholder-set!` *placeholder value*`)`
- `(placeholder-value` *placeholder*`)` → *value*

`Make-placeholder` creates an empty placeholder. (The optional *id* argument is only for debugging purposes; the discloser for placeholders prints it out if present.) `Placeholder?` is the predicate for placeholders.

`Placeholder-set!` places a value into a placeholder. Doing this more than once signals an error. `Placeholder-value` extracts the value from the placeholder and returns it. If the placeholder is empty, it blocks the current thread until it becomes full.

7.7 Writing custom synchronization abstractions

The bindings explained in this section are part of the `threads-internal` structure. They are concerned with suspending threads and making them runnable again upon some later event.

Typically, a suspended thread needs to be recorded in a queue somewhere for later waking-up. To allow a thread to be recorded in multiple queues (say, when it waits for one of a number of events), such *thread queues* are ordinary queues containing cells that, in turn, contain the thread objects themselves. Each thread has at most one such cell associated with it which is shared among all queues (or other data structures) holding on to the suspended thread. The cell is cleared when the thread is woken up.

- `(thread-queue-empty?` *thread-queue*`)` → *boolean*
- `(maybe-dequeue-thread!` *thread-queue*`)` → *boolean*

`Thread-queue-empty?` atomically checks whether the *thread-queue* thread queue is empty, i.e., if it does not contain non-empty cells. `Maybe-dequeue-thread!` provisionally dequeues a thread from *thread-queue* if it contains one. It returns the dequeued thread or `#f` if the queue is empty.

- `(maybe-commit-and-block` *cell*`)` → *boolean*
- `(maybe-commit-and-block-on-queue` *thread-queue*`)` → *boolean*
- `(maybe-commit-and-make-ready` *thread-or-queue*`)` → *boolean*

`Maybe-commit-and-block` attempts to commit the current proposal. If this succeeds, the current thread is blocked, the thread's cell is set to *cell*, and `#t` is returned. Otherwise, `#f` is returned. `Maybe-commit-and-block-on-queue` is like `maybe-commit-and-block`, excepts that it creates a fresh cell for the thread and enqueues it in *thread-queue* if the commit succeeds.

`Maybe-commit-and-make-ready` accepts either a thread object or a thread queue as an argument. In either case, `maybe-commit-and-make-ready` tries to commit the current proposal. If that succeeds, `maybe-commit-and-make-ready` makes its argument runnable: if *thread-or-queue* is a thread, that thread is made runnable, if it is a thread queue, all threads on the queue are made runnable. (In the latter case, none of the threads actually runs until all have been made runnable.) `Maybe-commit-and-make-ready` returns `#t` if it succeeded, and `#f` otherwise.

7.8 Concurrent ML abstractions

The interface to the Concurrent ML abstractions in Scheme 48 is mostly analogous to the original implementation shipped with SML/NJ [9]. Note that both the interface and implementation are new and may change in future releases.

The main terminological difference is that CML events are called *rendezvous* in Scheme 48. For more information on programming with the CML abstractions, Reppy's book [9] is recommended.

7.8.1 Basic rendezvous combinators

The basic rendezvous combinators live in the `rendezvous` structure.

- `never-rv` rendezvous
- `(always-rv` *value*`)` → *rendezvous*

`Never-rv` is a rendezvous that is never enabled for synchronization. (It is the same as the `never` event in CML.) `Always-rv` returns a rendezvous that is always enabled for synchronization, and always yields the same value *value*. (This is the same as the `alwaysEvt` function in CML.)

- `(choose` *rendezvous* ...`)` → *rendezvous*

`Choose` creates a rendezvous representing the choice of its arguments: Synchronization on the resulting rendezvous will synchronize on one of the arguments to `choose`, depending on which becomes enabled first. (This is the same as the `choose` function in CML.)

- `(wrap` *rendezvous proc*`)` → *rendezvous*

`Wrap` wraps a post-synchronization procedure around *rendezvous*: When the resulting rendezvous is synchronized, *rendezvous* is synchronized, and the value it yields is passed to *proc*; the value returned by *proc* then is the result of the synchronization. (This is the same as the CML `wrap` function.)

- `(guard` *thunk*`)` → *rendezvous*

`Guard` delays the creation of a rendezvous until synchronization time: It returns a rendezvous that will, upon synchronization, turn into the rendezvous returned by *thunk*. `Guard` can be used to perform pre-synchronization actions such as resource allocation. (This is the same as the CML `guard` function.)

- `(with-nack` *proc*`)` → *rendezvous*

`With-nack`, like `guard`, creates a delayed rendezvous: Upon synchronization, the rendezvous actually used is the one returned by *proc*. In addition to the functionality offered by `guard`, *proc* receives, as an argument, another rendezvous which becomes enabled when *another* rendezvous involved in the synchronization (via `choose`) is picked instead of the one produced by *proc*. (This is the same as the CML `withNack` function.)

- `(sync` *rendezvous*`)` → *value*
- `(select` *rendezvous* ...`)` → *value*

`Sync` synchronizes the current thread on rendezvous *rendezvous*, returning the value it yields. `Select` synchronizes on the choice of its argument; `(select` r_1 ...r_n`)` is semantically equivalent to `(sync (choose select` r_1 ...r_n`))`, but may be implemented more efficiently. (These are the same as the CML functions `sync` and `select`.)

7.8.2 Synchronous channels

The `rendezvous-channels` structure contains abstractions for bidirectional, synchronous channels for communicating between two threads.

- `(make-channel)` → *channel*
- `(channel?` *x*`)` → *boolean*

`Make-channel` creates a new synchronous channel. (This is the same as the CML `channel` function.) `Channel?` is the predicate for synchronous channels.

- `(send-rv` *channel value*`)` → *rendezvous*
- `(send` *channel value*`)`

`Send-rv` creates a rendezvous that, upon synchronization, sends message *value* on the synchronous channel *channel*. The synchronization suceeds only when another thread attempts to receive a message from *channel*. (This is the same as the CML `sendEvt` function.) `Send` directly sends a message *value* on channel *channel*; `(send` *c v*`)` is equivalent to `(sync (send-rv` *c v*`))`. (`Send` is the same as the CML `send` function.)

- `(receive-rv` *channel*`)` → *rendezvous*
- `(receive` *channel*`)`

`Receive-rv` creates a rendezvous which, upon synchronization, receives a message on channel *channel*. (This is the same as the CML `recEvt` function.) `Receive` directly receives a message on channel *channel*; `(receive` *c v*`)` is equivalent to `(sync (receive-rv` *c v*`))`. (`Receive` is the same as the CML `recv` function.)

7.8.3 Synchronous variables

Two structures contain abstractions for synchronous variables: the `rendezvous-placeholders` structure for so-called *placeholders* (write-once variables), and the `rendezvous-jars` structure for *jars* (which allow multiple updates.)

Placeholders

Placeholders are write-once variables. The placeholders implemented by the `rendezvous-placeholders` structure offer equivalent functionality to the placeholders implemented by the `placeholders` structure (see Section 7.6.2), but additionally allow converting a placeholder into a rendezvous. Note, however, that placeholders from `placeholders` are different from and not interchangeable with placeholders from `rendezvous-placeholders`.

- `(make-placeholder)` → *placeholder*
- `(make-placeholder` *id*`)` → *placeholder*
- `(placeholder?` *x*`)` → *boolean*

`Make-placeholder` creates an empty placeholder. (The optional *id* argument is only for debugging purposes; the discloser for placeholders prints it out if present.) (This is the same as the CML `iVar` function.) `Placeholder?` is the predicate for placeholders.

- `(placeholder-set!` *placeholder value*`)`

`Placeholder-set!` places a value into a placeholder. Doing this more than once signals an error. (This is the same as the CML `iPut` function.)

- `(placeholder-value-rv` *placeholder*`)` → *rendezvous*
- `(placeholder-value` *placeholder*`)` → *value*

`Placeholder-value` extracts the value from the placeholder and returns it. If the placeholder is empty, it blocks the current thread until it becomes full. (This is the same as the CML `iGet` function.) `Placeholder-value-rv` creates a rendezvous that will, upon synchronization, extract the value from the placeholder and yield it as a result. (This is the same as the CML `iGetEvt` function.)

Jars

A jar is a synchronous variable which can have two states: full and empty. It becomes full when a value it put into it; putting a value into a full jar is an error. Conversely, it becomes empty when a value is taken out of it. Trying to take a value out of an empty jar blocks until it becomes full. (Jars are similar to ID-90 M-structures.) Jars live in the `rendezvous-jars` structure.

- `(make-jar)` → *jar*
- `(make-jar` *id*`)` → *jar*
- `(jar?` *x*`)` → *boolean*

`Make-jar` creates an empty jar. (The optional *id* argument is only for debugging purposes; the discloser for jars prints it out if present.) (This is the same as the CML `mVar` function.) `Jar?` is the predicate for jars.

- `(jar-put!` *jar value*`)`

`Jar-put!` places a value into a jar if it is empty. Applying `jar-put!` to a full jar is an error. (This is the same as the CML `mPut` function.)

- `(jar-take-rv` *placeholder*`)` → *rendezvous*
- `(jar-take` *placeholder*`)` → *value*

`Jar-take` takes a value from a full jar, emptying it in the process. If the jar is empty, `jar-take` blocks until it becomes full. (This is the same as the CML `mTake` function.) `Jar-take-rv` creates a rendezvous that, upon synchronization, will extract the value from a jar and empty it in the process. (This is the same as the CML `mTakeEvt` function.)

7.8.4 Timeouts

The `rendezvous-time` structure allows creating rendezvous for alarms and timeouts:

- `(after-time-rv` *milliseconds*`)` → *rendezvous*
- `(at-real-time-rv` *time*`)` → *rendezvous*

`After-time-rv` creates a rendezvous that becomes enabled at time interval *milliseconds* after synchronization. (Actually, *milliseconds* is a minimum waiting time; the actual delay may be longer.) (This is the same as the CML `timeOutEvt` function.) `At-real-time-rv` creates a rendezvous that becomes enabled at an absolute time specified by *time*; this absolute time is specified in the same way as the return value `real-time` from the `time` structure. (This is the same as the CML `atTimeEvt` function.)

7.8.5 CML to Scheme correspondence

The following table lists the Scheme names that correspond to particular CML names.

CML name	Scheme name
	`rendezvous`
`never`	`never-rv`
`alwaysEvt`	`always-rv`
`choose`	`choose`
`wrap`	`wrap`
`guard`	`guard`
`withNack`	`with-nack`
`sync`	`sync`
`select`	`select`
	`rendezvous-channels`
`channel`	`make-channel`
`sendEvt`	`send-rv`
`send`	`send`

recEvt	receive-rv
rec	receive
	rendezvous-placeholders
iVar	make-placeholder
iPut	placeholder-set!
iGet	placeholder-value
iGetEvt	placeholder-value-rv
	rendezvous-jars
mVar	make-jar
mTake	jar-take
mTakeEvt	jar-take-rv
mPut	jar-put!
	rendezvous-time
timeOutEvt	after-time-rv
atTimeEvt	at-real-time-rv

Chapter 8

Mixing Scheme 48 and C

This chapter describes the foreign-function interface for calling C functions from Scheme, calling Scheme functions from C, and allocating storage in the Scheme heap. Scheme 48 manages stub functions in C that negotiate between the calling conventions of Scheme and C and the memory allocation policies of both worlds. No stub generator is available yet, but writing stubs is a straightforward task.

The foreign-function interface is modeled after the Java Native Interface (JNI), more information can be found at http://java.sun.com/javase/6/docs-/technotes/guides/jni/index.html.

Currently, Scheme 48 supports two foreign-function interfaces: The old GCPROTECT-style and the new JNI-style interface (this chapter) live side by side. The old interface is deprecated and will go away in a future release. Section 8.12 gives a recipe how to convert external code from the old to the new interface.

8.1 Available facilities

The following facilities are available for interfacing between Scheme 48 and 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 Scheme 48 to signal errors.
- External code may call back into Scheme. Scheme 48 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.

8.1.1 Scheme structures

The structure `external-calls` has most of the Scheme functions described here. The others are in `load-dynamic-externals`, which has the functions for dynamic loading and name lookup from Section 8.4, and `shared-bindings`, which has the additional shared-binding functions described in section 8.2.3.

8.1.2 C naming conventions

The names of all of Scheme 48's visible C bindings begin with '`s48_`' (for procedures, variables, and macros). Note that the new foreign-function interface does not distinguish between procedures and macros. Whenever a C name is derived from a Scheme identifier, we replace '`-`' with '`_`' and convert letters to lowercase. A final '`?`' converted to '`_p`', a final '`!`' is dropped. As a naming convention, all functions and macros of the new foreign-function interface end in '`_2`' (for now) to make them distinguishable from the old interface's functions and macros. Thus the C macro for Scheme's `pair?` is `s48_pair_p_2` and the one for `set-car!` is `s48_set_car_2`. Procedures and macros that do not check 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`.

8.1.3 Garbage collection and reference objects

Scheme 48 uses a precise, copying garbage collector. The garbage collector may run whenever an object is allocated in the heap. The collector must be able to locate all references to objects allocated in the Scheme 48 heap in order to ensure that storage is not reclaimed prematurely and to update references to objects moved by the collector. This interface takes care of communicating to the garbage collector what objects it uses in most situations. It relieves the programmer from having to think about garbage collector interactions in the common case.

This interface does not give external code direct access to Scheme objects. It introduces one level of indirection as external code never accepts or returns Scheme values directly. Instead, external code accepts or returns *reference objects* of type `s48_ref_t` that refer to Scheme values (their C type is defined to be `s48_value`). This indirection is only needed as an interface to external code, interior pointers in Scheme objects are unaffected.

There are two types of reference objects:

local references A local reference is valid for the duration of a function call from Scheme to external code and is automatically freed after the external function returns to the virtual machine.

global references A global reference remains valid until external code explicitly frees it.

Scheme objects that are passed to external functions are passed as local references. External functions return Scheme objects as local references. External code has to manually manage Scheme objects that outlive a function call as global references. Scheme objects outlive a function call if they are assigned to a global variable of the external code or stored in long-living external objects, see section 8.7.1.

A local reference is valid only within the dynamic context of the native method that creates it. Therefore, a local reference behaves exactly like a local variable in the external code: It is live as long as external code can access it. To achieve this, every external function in the interface that accepts or returns reference objects takes a *call object* of type `s48_call_t` as its first argument. A call object corresponds to a particular call from Scheme to C. The call object holds all the references that belong to a call (like the call's arguments and return value) to external code from Scheme. External code may pass a local reference through multiple external functions. The foreign-function interface automatically frees all the local references a call object owns, along with the call object itself, when an external call returns to Scheme.

This means that in the common case of Scheme calling an external function that does some work on its arguments and returns without stashing any Scheme objects in global variables or global data structures, the external code does not need to do any bookkeeping, since all the reference objects the external code accumulates are local references. Once the call returns, the foreign-function interface frees all the local references.

For example, the functions to construct and access pairs are declared like this:

- `s48_ref_t s48_cons_2(s48_call_t call, s48_ref_t car, s48_ref_t cdr);`
- `s48_ref_t s48_car_2(s48_call_t call, s48_ref_t pair);`
- `s48_ref_t s48_cdr_2(s48_call_t call, s48_ref_t pair);`

This foreign-function interface takes a significant burden off the programmer as it handles most common cases automatically. If all the Scheme objects are live for the extent of the current external call, the programmer does not have to do anything at all. Since the lifetime of the Scheme objects is then identical with the lifetime of the according reference objects. In this case, the systems automatically manages both for the programmer. Using this foreign-function interface does not make the code more complex; the code stays compact and readable. The programmer has to get accustomed to passing the call argument around.

How to manage Scheme objects that outlive the current call is described in section 8.7.1.

Section 8.12 gives a recipe how to convert external code from the old GCPROTECT-style interface to the new JNI-style interface.

8.2 Shared bindings

Shared bindings are the means by which named values are shared between Scheme code and C code. There are two separate tables of shared bindings, one for values defined in Scheme and accessed from C and the other for values going the other way. Shared bindings actually bind names to cells, to allow a name to be looked up before it has been assigned. This is necessary because C initialization code may be run before or after the corresponding Scheme code, depending on whether the Scheme code is in the resumed image or is run in the current session.

8.2.1 Exporting Scheme values to C

- `(define-exported-binding` *name value*`)` → *shared-binding*

- `s48_ref_t s48_get_imported_binding_2(char *name)`
- `s48_ref_t s48_get_imported_binding_local_2(s48_call_t call, char *name)`
- `s48_ref_t s48_shared_binding_ref_2(s48_call_t call, s48_ref_t shared_binding)`

`Define-exported-binding` makes *value* available to C code under *name*, which must be a *string*, creating a new shared binding if necessary. The C function `s48_get_imported_binding_2` returns a global reference to the shared binding defined for `name`, again creating it if necessary, `s48_get_imported_binding_local_2` returns a local reference to the shared binding (see section 8.1.3 for details on reference objects). The C macro `s48_shared_binding_ref_2` dereferences a shared binding, returning its current value.

8.2.2 Exporting C values to Scheme

Since shared bindings are defined during initialization, i.e. outside an external call, there is no call object. Therefore, exporting shared bindings from C does not use the new foreign-function interfaces specifications.

- `void s48_define_exported_binding(char *name, s48_value v)`

- `(lookup-imported-binding` *string*`)` → *shared-binding*
- `(shared-binding-ref` *shared-binding*`)` → *value*

These are used to define shared bindings from C and to access them from Scheme. Again, if a name is looked up before it has been defined, a new binding is created for it.

The common case of exporting a C function to Scheme can be done using the macro `s48_export_function(`*name*`)`. This expands into

```
s48_define_exported_binding("name",
                            s48_enter_pointer(name))
```

which boxes the function pointer into a Scheme byte vector and then exports it. Note that `s48_enter_pointer` allocates space in the Scheme heap and might trigger a garbage collection; see Section 8.7.

- (import-definition *name*) syntax
- (import-definition *name c-name*) syntax

These macros simplify importing definitions from C to Scheme. They expand into

    (define *name* (lookup-imported-binding *c-name*))

where *c-name* is as supplied for the second form. For the first form *c-name* is derived from *name* by replacing '-' with '_' and converting letters to lowercase. For example, (import-definition my-foo) expands into

    (define my-foo (lookup-imported-binding "my_foo"))

8.2.3 Complete shared binding interface

There are a number of other Scheme functions related to shared bindings; these are in the structure shared-bindings.

- (shared-binding? *x*) $\rightarrow$ *boolean*
- (shared-binding-name *shared-binding*) $\rightarrow$ *string*
- (shared-binding-is-import? *shared-binding*) $\rightarrow$ *boolean*
- (shared-binding-set! *shared-binding value*)
- (define-imported-binding *string value*)
- (lookup-exported-binding *string*)
- (undefine-imported-binding *string*)
- (undefine-exported-binding *string*)

Shared-binding? is the predicate for shared-bindings. Shared-binding-name returns the name of a binding. Shared-binding-is-import? is true if the binding was defined from C. Shared-binding-set! changes the value of a binding. Define-imported-binding and lookup-exported-binding are Scheme versions of s48_define_exported_binding and s48_lookup_imported_binding. The two undefine- procedures remove bindings from the two tables. They do nothing if the name is not found in the table.

The following C macros correspond to the Scheme functions above.

```
• int       s48_shared_binding_p(s48_call_t call, x)
• int       s48_shared_binding_is_import_p(s48_call_t call, s48_ref_t s_b)
• s48_ref_t s48_shared_binding_name(s48_call_t call, s48_ref_t s_b)
• void      s48_shared_binding_set(s48_call_t call, s48_ref_t s_b, s48_ref_t v)
```

8.3 Calling C functions from Scheme

There are different ways to call C functions from Scheme, depending on how the C function was obtained.

- (call-imported-binding-2 *binding* arg_0 ...) $\rightarrow$ *value*

Each of these applies its first argument, a C function that accepts and/or returns objects of type s48_ref_t and has its first argument of type s48_call_t, to the rest of the arguments. For `call-imported-binding-2` the function argument must be an imported binding.

For all of these, the interface passes the current call object and the *arg*$_i$ values to the C function and the value returned is that returned by C procedure. 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 (this is clearly a problem; we are working on a solution).

- `(import-lambda-definition-2` *name* `(`*formal* `...))` syntax
- `(import-lambda-definition-2` *name* `(`*formal* `...)` *c-name*`)` syntax

These macros simplify importing functions from C that follow the return value and argument conventions of the foreign-function interface and use `s48_call_t` and `s48_ref_t` as their argument and return types. They define *name* to be a function with the given formals that applies those formals to the corresponding C binding. *C-name*, if supplied, should be a string. These expand into

```
(define temp (lookup-imported-binding c-name))
(define name
  (lambda (formal ...)
    (call-imported-binding-2 temp formal ...)))
```

If *c-name* is not supplied, it is derived from *name* by converting all letters to lowercase and replacing '`-`' with '`_`'.

8.4 Dynamic loading

External code can be loaded into a running Scheme 48—at least on most variants of Unix and on Windows. The required Scheme functions are in the structure `load-dynamic-externals`.

To be suitable for dynamic loading, the externals code must reside in a shared object. The shared object must define a function:

- `void s48_on_load(void)`

The `s48_on_load` is run upon loading the shared objects. It typically contains invocations of `S48_EXPORT_FUNCTION` to make the functionality defined by the shared object known to Scheme 48.

The shared object may also define either or both of the following functions:

- `void s48_on_unload(void)`
- `void s48_on_reload(void)`

Scheme 48 calls `s48_on_unload` just before it unloads the shared object. If `s48_on_reload` is present, Scheme 48 calls it when it loads the shared object for the second time, or some new version thereof. If it is not present, Scheme 48 calls `s48_on_load` instead. (More on that later.)

For Linux, 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
```

The following procedures provide the basic functionality for loading shared objects containing dynamic externals:

- `(load-dynamic-externals` *string plete? rrepeat? rresume?*`)` → *dynamic-externals*
- `(unload-dynamic-externals` *string*`)` → *dynamic-externals*
- `(reload-dynamic-externals` *dynamic-externals*`)`

`Load-dynamic-externals` loads the named shared objects. The *plete?* argument determines whether Scheme 48 appends the OS-specific suffix (typically `.so` for Unix, and `.dll` for Windows) to the name. The *rrepeat?* argument determines how `load-dynamic-externals` behaves if it is called again with the same argument: If this is true, it reloads the shared object (and calls its `s48_on_unload` on unloading if present, and, after reloading, `s48_on_reload` if present or `s48_on_load` if not), otherwise, it will not do anything. The *rresume?* argument determines if an image subsequently dumped will try to load the shared object again automatically. (The shared objects will be loaded before any record resumers run.) `Load-dynamic-externals` returns a handle identifying the shared object just loaded.

`Unload-dynamic-externals` unloads the shared object associated with the handle passed as its argument, previously calling its `s48_on_unload` function if present. Note that this invalidates all external bindings associated with the shared object; referring to any of them will probably crash the program.

`Reload-dynamic-externals` will reload the shared object named by its argument and call its `s48_on_unload` function before unloading, and, after reloading, `s48_on_reload` if present or `s48_on_load` if not.

- `(import-dynamic-externals` *string*`)` → *dynamic-externals*

This procedure represents the expected most usage for loading dynamic-externals. It is best explained by its definition:

```
(define (import-dynamic-externals name)
  (load-dynamic-externals name #t #f #t))
```

8.5 Accessing Scheme data from C

The C header file `scheme48.h` provides access to Scheme 48 data structures. The type `s48_ref_t` is used for reference objects that refer to Scheme values. When the type of a value is known, such as the integer returned by `vector-length` or the boolean returned by `pair?`, the corresponding C procedure returns a C value of the appropriate type, and not a `s48_ref_t`. Predicates return `1` for true and `0` for false.

8.5.1 Constants

The following macros denote Scheme constants:

- `s48_false_2(s48_call_t)` is `#f`.
- `s48_true_2(s48_call_t)` is `#t`.
- `s48_null_2(s48_call_t)` is the empty list.
- `s48_unspecific_2(s48_call_t)` is a value used for functions which have no meaningful return value (in Scheme 48 this value returned by the nullary procedure `unspecific` in the structure `util`).
- `s48_eof_2(s48_call_t)` is the end-of-file object (in Scheme 48 this value is returned by the nullary procedure `eof-object` in the structure `i/o-internal`).

8.5.2 Converting values

The following macros and functions convert values between Scheme and C representations. The 'extract' ones convert from Scheme to C and the 'enter's go the other way.

- `int       s48_extract_boolean_2(s48_call_t, s48_ref_t)`
- `long      s48_extract_char_2(s48_call_t, s48_ref_t)`
- `char *    s48_extract_byte_vector_2(s48_call_t, s48_ref_t)`
- `long      s48_extract_long_2(s48_call_t, s48_ref_t)`
- `long      s48_extract_unsigned_long_2(s48_call_t, s48_ref_t)`
- `double    s48_extract_double_2(s48_call_t, s48_ref_t)`
- `s48_ref_t s48_enter_boolean_2(s48_call_t, int)`
- `s48_ref_t s48_enter_char_2(s48_call_t, long)`
- `s48_ref_t s48_enter_byte_vector_2(s48_call_t, char *, long)` (may GC)
- `s48_ref_t s48_enter_long_2(s48_call_t, long)` (may GC)
- `s48_ref_t s48_enter_long_as_fixnum_2(s48_call_t, long)` (may GC)
- `s48_ref_t s48_enter_double_2(s48_call_t, double)` (may GC)

`s48_extract_boolean_2` is false if its argument is `#f` and true otherwise. `s48_enter_boolean_2` is `#f` if its argument is zero and `#t` otherwise.

The `s48_extract_char_2` function extracts the scalar value from a Scheme character as a C `long`. Conversely, `s48_enter_char_2` creates a Scheme character from a scalar value. (Note that ASCII values are also scalar values.)

The `s48_extract_byte_vector_2` function needs to deal with the garbage collector to avoid invalidating the returned pointer. For more details see section 8.7.3.

The second argument to `s48_enter_byte_vector_2` is the length of byte vector.

`s48_enter_long_2()` needs to allocate storage when its argument is too large to fit in a Scheme 48 fixnum. In cases where the number is known to fit within a fixnum (currently 30 bits on a 32-bits architecture and 62 bit on a 64-bits architecture including the sign), the following procedures can be used. These have the disadvantage of only having a limited range, but the advantage of never causing a garbage collection. `s48_fixnum_p_2(s48_call_t)` is a macro that true if its argument is a fixnum and false otherwise.

- `int       s48_fixnum_p_2(s48_call_t, s48_ref_t)`
- `s48_ref_t s48_enter_long_as_fixnum_2(s48_call_t, long)`
- `long      S48_MAX_FIXNUM_VALUE`
- `long      S48_MIN_FIXNUM_VALUE`

An error is signaled if the argument to `s48_enter_fixnum` is less than `S48_MIN_FIXNUM_VALUE` or greater than `S48_MAX_FIXNUM_VALUE` (-2^{29} and $2^{29}-1$ on a 32-bits architecture and -2^{61} and $2^{62}-1$ on a 64-bits architecture).

- `int       s48_true_p_2(s48_call_t, s48_ref_t)`
- `int       s48_false_p_2(s48_call_t, s48_ref_t)`

`s48_true_p` is true if its argument is `s48_true` and `s48_false_p` is true if its argument is `s48_false`.

- `s48_ref_t s48_enter_string_latin_1_2(s48_call_t, char*);` (may GC)
- `s48_ref_t s48_enter_string_latin_1_n_2(s48_call_t, char*, long);` (may GC)
- `long      s48_string_latin_1_length_2(s48_call_t, s48_ref_t);`
- `long      s48_string_latin_1_length_n_2(s48_call_t, s48_ref_t, long, long);`
- `void      s48_copy_latin_1_to_string_2(s48_call_t, char*, s48_ref_t);`
- `void      s48_copy_latin_1_to_string_n_2(s48_call_t, char*, long, s48_ref_t);`
- `void      s48_copy_string_to_latin_1_2(s48_call_t, s48_ref_t, char*);`
- `void      s48_copy_string_to_latin_1_n_2(s48_call_t, s48_ref_t, long, long, char*);`
- `s48_ref_t s48_enter_string_utf_8_2(s48_call_t, char*);` (may GC)
- `s48_ref_t s48_enter_string_utf_8_n_2(s48_call_t, char*, long);` (may GC)
- `long      s48_string_utf_8_length_2(s48_call_t, s48_ref_t);`
- `long      s48_string_utf_8_length_n_2(s48_call_t, s48_ref_t, long, long);`
- `long      s48_copy_string_to_utf_8_2(s48_call_t, s48_ref_t, char*);`
- `long      s48_copy_string_to_utf_8_n_2(s48_call_t, s48_ref_t, long, long, char*);`
- `s48_ref_t s48_enter_string_utf_16be_2(s48_call_t, char*);` (may GC)
- `s48_ref_t s48_enter_string_utf_16be_n_2(s48_call_t, char*, long);` (may GC)
- `long      s48_string_utf_16be_length_2(s48_call_t, s48_ref_t);`

- `long       s48_string_utf_16be_length_n_2(s48_call_t, s48_ref_t, long, long);`
- `long       s48_copy_string_to_utf_16be_2(s48_call_t, s48_ref_t, char*);`
- `long       s48_copy_string_to_utf_16be_n_2(s48_call_t, s48_ref_t, long, long, char*);`
- `s48_ref_t s48_enter_string_utf_16le_2(s48_call_t, char*);` (may GC)
- `s48_ref_t s48_enter_string_utf_16le_n_2(s48_call_t, char*, long);` (may GC)
- `long       s48_string_utf_16le_length_2(s48_call_t, s48_ref_t);`
- `long       s48_string_utf_16le_length_n_2(s48_call_t, s48_ref_t, long, long);`
- `long       s48_copy_string_to_utf_16le_2(s48_call_t, s48_ref_t, char*);`
- `long       s48_copy_string_to_utf_16le_n_2(s48_call_t, s48_ref_t, long, long, char*);`

The `s48_enter_string_latin_1_2` function creates a Scheme string, initializing its contents from its NUL-terminated, Latin-1-encoded argument. The `s48_enter_string_latin_1_n_2` function does the same, but allows specifying the length explicitly—no NUL terminator is necessary.

The `s48_string_latin_1_length_2` function computes the length that the Latin-1 encoding of its argument (a Scheme string) would occupy, not including NUL termination. The `s48_string_latin_1_length_2` function does the same, but allows specifying a starting index and a count into the input string.

The `s48_copy_latin_1_to_string_2` function copies Latin-1-encoded characters from its second NUL-terminated argument to the Scheme string that is its third argument. The `s48_copy_latin_1_to_string_n_2` does the same, but allows specifying the number of characters explicitly. The `s48_copy_string_to_latin_1_2` function converts the characters of the Scheme string specified as the second argument into Latin-1 and writes them into the string specified as the third argument. (Note that it does not NUL-terminate the result.) The `s48_copy_string_to_latin_1_n_2` function does the same, but allows specifying a starting index and a character count into the source string.

The `s48_extract_latin_1_from_string_2` function returns a buffer that contains the Latin-1 encoded characters including NUL termination of the Scheme string specified. The buffer that is returned is a local buffer managed by the foreign-function interface and is automatically freed on the return of the current call.

The `s48_enter_string_utf_8_2` function creates a Scheme string, initializing its contents from its NUL-terminated, UTF-8-encoded argument. The `s48_enter_string_utf_8_n_2` function does the same, but allows specifying the length explicitly—no NUL terminator is necessary.

The `s48_string_utf_8_length_2` function computes the length that the UTF-8 encoding of its argument (a Scheme string) would occupy, not including NUL termination. The `s48_string_utf_8_length_2` function does the same, but allows specifying a starting index and a count into the input string.

The `s48_copy_string_to_utf_8_2` function converts the characters of the Scheme string specified as the second argument into UTF-8 and writes them into the string specified as the third argument. (Note that it does not NUL-terminate the result.) The `s48_copy_string_to_utf_8_n_2` function does the same, but allows specifying a starting index and a character count into the source string. Both return the length of the written encodings in bytes.

The `s48_extract_utf_8_from_string_2` function returns a buffer that contains the UTF-8 encoded characters including NUL termination of the Scheme string specified. The buffer that is returned is a local buffer managed by the foreign-function interface and is automatically freed on the return of the current call.

The functions with `utf_16` in their names work analogously to their `utf_8` counterparts, but implement the UTF-16 encodings. The lengths returned be the `_length` and `copy_string_to` functions are in terms of UTF-16 code units. The `extract` function returns a local buffer that contains UTF-16 code units including NUL termination.

8.5.3 C versions of Scheme procedures

The following macros and procedures are C versions of Scheme procedures. The names were derived by replacing '`-`' with '`_`', '`?`' with '`_p`', and dropping '`!`.

- `int       s48_eq_p_2(s48_call_t, s48_ref_t, s48_ref_t)`
- `int       s48_char_p_2(s48_call_t, s48_ref_t)`
- `int       s48_null_p_2(s48_call_t, s48_ref_t)`

- `int       s48_pair_p_2(s48_call_t, s48_ref_t)`
- `s48_ref_t s48_car_2(s48_call_t, s48_ref_t)`
- `s48_ref_t s48_cdr_2(s48_call_t, s48_ref_t)`
- `void      s48_set_car_2(s48_call_t, s48_ref_t, s48_ref_t)`
- `void      s48_set_cdr_2(s48_call_t, s48_ref_t, s48_ref_t)`
- `s48_ref_t s48_cons_2(s48_call_t, s48_ref_t, s48_ref_t)` (may GC)
- `long      s48_length_2(s48_call_t, s48_ref_t)`

- `int       s48_vector_p_2(s48_call_t, s48_ref_t)`
- `long      s48_vector_length_2(s48_call_t, s48_ref_t)`
- `s48_ref_t s48_vector_ref_2(s48_call_t, s48_ref_t, long)`
- `void      s48_vector_set_2(s48_call_t, s48_ref_t, long, s48_ref_t)`
- `s48_ref_t s48_make_vector_2(s48_call_t, long, s48_ref_t)` (may GC)

- `int       s48_string_p_2(s48_call_t, s48_ref_t)`
- `long      s48_string_length_2(s48_call_t, s48_ref_t)`
- `long      s48_string_ref_2(s48_call_t, s48_ref_t, long)`
- `void      s48_string_set_2(s48_call_t, s48_ref_t, long, long)`
- `s48_ref_t s48_make_string_2(s48_call_t, long, char)` (may GC)

- `int       s48_symbol_p_2(s48_call_t, s48_ref_t)`
- `s48_ref_t s48_symbol_to_string_2(s48_call_t, s48_ref_t)`

- `int       s48_byte_vector_p_2(s48_call_t, s48_ref_t)`
- `long      s48_byte_vector_length_2(s48_call_t, s48_ref_t)`
- `char      s48_byte_vector_ref_2(s48_call_t, s48_ref_t, long)`
- `void      s48_byte_vector_set_2(s48_call_t, s48_ref_t, long, int)`
- `s48_ref_t s48_make_byte_vector_2(s48_call_t, long, int)` (may GC)

8.6 Calling Scheme functions from C

External code that has been called from Scheme can call back to Scheme procedures using the following function.

- `s48_ref_t s48_call_scheme_2(s48_call_t, s48_ref_t p, long nargs, ...)`

This calls the Scheme procedure `p` on `nargs` arguments, which are passed as additional arguments to `s48_call_scheme_2`. There may be at most twelve arguments. The value returned by the Scheme procedure is returned by the C procedure. Invoking any Scheme procedure may potentially cause a garbage collection.

There are some complications that occur when mixing calls from C to Scheme with continuations and threads. C only supports 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 procedure `c0`, which in turn calls Scheme procedure `s1`. Procedure `s1` can safely call the continuation `a`, because that is a downward use. When `a` is called Scheme 48 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 `c0` 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 Scheme 48 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 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 procedure which calls back to Scheme, at which point control switches to thread `t1`, which also calls C and then back to Scheme. At this point both `t0` and `t1` have active calls to C on the C stack, with `t1`'s C frame above `t0`'s. If thread `t0` attempts to return from Scheme to C it will block, as its frame is not accessible. Once `t1` has returned to C and from there to Scheme, `t0` will be able to resume. The return to Scheme is required because context switches can only occur while Scheme code is running. `T0` will also be able to resume if `t1` uses a continuation to throw past its call to C.

8.7 Interacting with the Scheme heap

Scheme 48 uses a copying, precise garbage collector. Any procedure that allocates objects within the Scheme 48 heap may trigger a garbage collection.

Local object references refer to values in the Scheme 48 heap and are automatically registered with the garbage collector by the interface for the duration of a function call from Scheme to C so that the values will be retained and the references will be updated if the garbage collector moves the object.

Global object references need to be created and freed explicitly for Scheme values that need to survive one function call, e.g. that are stored in global variables, global data structures or are passed to libraries. See section 8.7.1 for details.

Additionally, the interface provides *local buffers* that are malloc'd regions of memory valid for the duration of a function call and are freed automatically upon return from the external code. This relieves the programmer from explicitly freeing locally allocated memory. See section 8.7.2 for details.

The interface treats byte vectors in a special way, since the garbage collector has no facility for updating pointers to the interiors of objects, so that such pointers, for example the ones returned by `s48_unsafe_extract_byte_vector_2`, will likely become invalid when a garbage collection occurs. The interface provides a facility to prevent a garbage collection from invalidating pointers to byte vector's memory region, see section 8.7.3 for details.

8.7.1 Registering global references

When external code needs a reference object to survive the current call, the external code needs to do explicit bookkeeping. Local references must not be stored in global variables of the external code or passed to other threads, since all local references are freed once the call returns, which leads to a dangling pointer in the global variable or other thread respectively. Instead, promote a local reference to a global reference and store it in a global variable or pass to another thread as global references will survive the current call. Since the foreign-function interface never automatically frees global references, the programmer must free them at the right time.

- `s48_ref_t s48_make_global_ref(s48_value obj)`
- `void      s48_free_global_ref(s48_ref_t ref)`
- `s48_ref_t s48_local_to_global_ref(s48_ref_t ref)`

`s48_make_global_ref` permanently registers the Scheme value *obj* as a global reference with the garbage collector. Basic Scheme values are `_s48_value_true`, `_s48_value_false`, `_s48_value_null`, `_s48_value_unspecific`, `_s48_value_undefined`, and `_s48_value_eof`.

To free a global reference an to unregister it with the garbage collector, use `s48_free_global_ref`. The function `s48_local_to_global_ref` promotes a local reference object to a global reference object.

For example, to maintain a global list of values, declare a static global variable

```
s48_ref_t global_list = NULL;
```

and initialize it in the external code's initialization function

```
global_list = s48_make_global_ref(_s48_value_null);
```

Note that you need to use a Scheme value (not a reference object) as the argument for `s48_make_global_ref`, since there is not yet a call object at the time external code gets initialized. To add `new_value` to the list, you can use the following snippet:

```
  temp = global_list;
  global_list =  s48_local_to_global_ref(s48_cons_2(call, new_value, global_list))
  s48_free_global_ref(temp);
```

You have to remember to always promote reference objects to global references when assigning to a global variable (and later, to free them manually). Note that it is more efficient to free the previous head of the list when adding a new element to the list.

8.7.2 Local buffers

The foreign-function interface supports the programmer with allocating memory in external code: The programmer can request chunks of memory, called local buffers, that are automatically freed on return from the current call.

- `void *s48_make_local_buf (s48_call_t, size_t)`
- `void s48_free_local_buf (s48_call_t, void *)`

The function `s48_make_local_buf` returns a block of memory of the given size in bytes. This memory freed by the foreign-function interface when the current call returns. To free the buffer manually, use `s48_free_local_buf`.

8.7.3 Special treatment for byte vectors

The interface treats byte vectors in a special way, since the garbage collector has no facility for updating pointers to the interiors of objects, so that such pointers, for example the ones returned by `s48_unsafe_extract_byte_vector_2`, will likely become invalid when a garbage collection occurs. The interface provides a facility to prevent a garbage collection from invalidating pointers to byte vector's memory region. It does this by copying byte vectors that are used in external code from and to the Scheme heap.

These functions create byte vectors:

- `s48_ref_t s48_make_byte_vector_2(s48_call_t, long)` (may GC)
- `s48_ref_t s48_make_unmovable_byte_vector_2(s48_call_t, long)` (may GC)
- `s48_ref_t s48_enter_byte_vector_2(s48_call_t, const char *, long)` (may GC)
- `s48_ref_t s48_enter_unmovable_byte_vector_2(s48_call_t, const char *, long)` (may GC)

`s48_make_byte_vector_2` creates a byte vector of given size, `s48_make_unmovable_byte_vector_2` creates a byte vector in that is not moved by the garbage collector (only the Bibop garbage collector supports this). The functions `s48_enter_byte_vector_2` and `s48_enter_unmovable_byte_vector_2` create and initialize byte vectors.

The following functions copy byte vectors from and to the Scheme heap:

- `void s48_extract_byte_vector_region_2(s48_call_t, s48_ref_t, long, long, char*)`
- `void s48_enter_byte_vector_region_2(s48_call_t, s48_ref_t, long, long, char*)`

- `void s48_copy_from_byte_vector_2(s48_call_t, s48_ref_t, char *)`
- `void s48_copy_to_byte_vector_2(s48_call_t, s48_ref_t, char *)`

`s48_extract_byte_vector_region_2` copies a given section from the given byte vector to its last argument, `s48_enter_byte_vector_region_2` copies the contents of its last argument to its first argument to the given index. `s48_copy_from_byte_vector_2` copies the whole byte vector to its last argument, `s48_copy_to_byte_vector_2` copies the contents of its last argument to the byte vector.

- `char *s48_extract_byte_vector_unmanaged_2(s48_call_t, s48_ref_t)`
- `void  s48_release_byte_vector_2(s48_call_t, s48_ref_t, char*)`

`s48_extract_byte_vector_unmanaged_2` returns a local buffer that is valid during the current external call and copies the contents of the given byte vector to the returned buffer. The returned byte vector may be a copy of the Scheme byte vector, changes made to the returned byte vector will not necessarily be reflected in Scheme until `s48_release_byte_vector_2` is called.

The following functions to access byte vectors come with the most support from the foreign-function interface. Byte vectors that are accessed via these functions are automatically managed by the interface and are copied back to Scheme on return from the current call:

- `char *s48_extract_byte_vector_2(s48_call_t, s48_ref_t)`
- `char *s48_extract_byte_vector_readonly_2(s48_call_t, s48_ref_t)`

`s48_extract_byte_vector_2` extracts a byte vector from Scheme by making a copy of the byte vectors contents and returning a pointer to that copy. Changes to the byte vector are automatically copied back to the Scheme heap when the function returns, external code raises an exception, or external code calls a Scheme function. `s48_extract_byte_vector_readonly_2` should be used for byte vectors that are not modified by external code, since these byte vectors are not copied back to Scheme.

8.7.4 Memory overhead

Each reference object consumes a certain amount of memory itself, in addition to the memory taken by the referred Scheme object itself. Even though local references are eventually freed on return of an external call, there are some situations where it is desirable to free local references explicitly, since waiting until the call returns may be too long or never happen, which could keep unneeded objects live:

- External code may create a large number of local references in a single external call. An example is the traversal of a list: Each call from external code to the functions that correspond to `car` and `cdr` returns a fresh local reference. To avoid the consumption of storage for local references proportional to the length of the list, the traversal must free the no-longer-needed references as it goes.

  For example, this is a straightforward definition of an external function that calculates the length of a list:

```
s48_ref_t
s48_length_2(s48_call_t call, s48_ref_t list)
{
  long i = 0;
  while (!(s48_null_p_2(call, list)))
    {
      list = s48_cdr_2(call, list);
      ++i;
    }
  return s48_unsafe_enter_long_as_fixnum_2(call, i);
}
```

In this implementation, each iteration step creates a new local reference object via `s48_cdr_2` that is actually only needed for the next iteration step. As a result, this function creates new local references for every element of the list. The local references are live during the entire function call.

To avoid consuming storage proportional to the length of the list for all those local reference objects, the improved version cleans up the unneeded local reference on every iteration step:

```
s48_ref_t
s48_length_2(s48_call_t call, s48_ref_t list)
{
  s48_ref_t l = s48_copy_local_ref(call, list);
  long i = 0;
  while (!(s48_null_p_2(call, l)))
    {
      s48_ref_t temp = l;
      l = s48_cdr_2(call, l);
      s48_free_local_ref(call, temp);
      ++i;
    }
  return s48_unsafe_enter_long_as_fixnum_2(call, i);
}
```

Note that without the call to `s48_copy_local_ref` the reference to the head of the list would be freed along with all the temporary references. This would render the whole list unusable after the return from `s48_length_2`.

- The external call does not return at all. If the external function enters an infinite event dispatch loop, for example, it is crucial that the programmer releases local references manually that he created inside the loop so that they do not accumulate indefinitely and lead to a memory leak.

- External code may hold a local reference to a large Scheme object. After the external code is done working on this object, it performs some additional computation before returning to the caller. The local reference to the large object prevents the object from being garbage collected until the external function returns, even if the object is no longer in use for the remainder of the computation. It is more space-efficient if the programmer frees the local reference when the external function does not need it any longer and will not return for quite some time.
- There are common situations where local references are created solely to be passed to another function and afterwards never used again. In this case, the called function can free the local references of the arguments.
- To improve memory usage while making subcalls from external calls, the foreign-function interface provides functionality to create a new (sub-)call object and clean the local references that are created during that subcall:
  - `s48_call_t s48_make_subcall(s48_call_t call)`
  - `void s48_free_subcall(s48_call_t subcall)`
  - `s48_ref_t s48_finish_subcall(s48_call_t call, s48_call_t subcall, s48_ref_t ref)`

  `s48_make_subcall` returns a new call object that represents a subcall of the current call and can be passed as the call argument to any subcalls of the current call. Upon return of a subcall, `s48_free_subcall` frees the subcall and all the local references associated with it. The function s48_finish_subcall also frees the subcall and all the local references associated with it, but copies its third argument to the current call, so that it survives the subcall.

8.7.5 Keeping C data structures in the Scheme heap

C data structures can be kept in the Scheme heap by embedding them inside byte vectors. The following macros can be used to create and access embedded C objects.

```
• s48_ref_t s48_make_value_2(s48_call_t, type)             (may GC)
• s48_ref_t s48_make_sized_value_2(s48_call_t, size)       (may GC)
• type      s48_extract_value_2(s48_call_t, s48_ref_t, type)
• long      s48_value_size_2(s48_call_t, s48_ref_t)
• type *    s48_extract_value_pointer_2(s48_call_t, s48_ref_t, type)
• void      s48_set_value_2(s48_call_t, s48_ref_t, type, value)
```

`s48_make_value_2` makes a byte vector large enough to hold an object whose type is *type*. `s48_make_sized_value_2` makes a byte vector large enough to hold an object of *size* bytes. `s48_extract_value_2` returns the contents of a byte vector cast to *type*, `s48_value_size_2` returns its size, and `s48_extract_value_pointer_2` returns a pointer to the contents of the byte vector. The value returned by `s48_extract_value_pointer_2` is valid only until the next garbage collection. `s48_set_value_2` stores `value` into the byte vector.

Pointers to C data structures can be stored in the Scheme heap:

- `s48_ref_t s48_enter_pointer_2(s48_call_t, void *)` (may GC)
- `void *    s48_extract_pointer_2(s48_call_t, s48_ref_t)` (may GC)

The function `s48_enter_pointer_2` makes a byte vector large enough to hold the pointer value and stores the pointer value in the byte vector. The function `s48_extract_pointer_2` extracts the pointer value from the scheme heap.

8.7.6 C code and heap images

Scheme 48 uses dumped heap images to restore a previous system state. The Scheme 48 heap is written into a file in a machine-independent and operating-system-independent format. The procedures described above 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 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.

- `(define-record-resumer` *record-type procedure*`)`

`Define-record-resumer` defines *procedure*, which should accept one argument, to be the resumer for *record-type*. The order in which resumer procedures are called is not specified.

The *procedure* argument to `define-record-resumer` may be `#f`, in which case records of the given type are not written out in heap images. When writing a heap image any reference to such a record is replaced by the value of the record's first field, and an exception is raised after the image is written.

8.8 Using Scheme records in C code

External modules can create records and access their slots positionally.

- `s48_ref_t s48_make_record_2(s48_call_t, s48_ref_t)` (may GC)
- `int       s48_record_p_2(s48_call_t, s48_ref_t)`
- `s48_ref_t s48_record_type_2(s48_call_t, s48_ref_t)`
- `s48_ref_t s48_record_ref_2(s48_call_t, s48_ref_t, long)`
- `void      s48_record_set_2(s48_call_t, s48_ref_t, long, s48_ref_t)`

The argument to `s48_make_record_2` should be a shared binding whose value is a record type. In C the fields of Scheme records are only accessible via offsets, with the first field having offset zero, the second offset one, and so forth. If the order of the fields is changed in the Scheme definition of the record type the C code must be updated as well.

For example, given the following 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:

```
(define-exported-binding "thing-record-type" :thing)
```

`Thing` records can then be made in C:

```
static s48_ref_t
  thing_record_type_binding = NULL;

void initialize_things(void)
{
  thing_record_type_binding =
     s48_get_imported_binding_2("thing-record-type");
}

s48_ref_t make_thing(s48_call_t call, s48_ref_t a, s48_ref_t b)
{
  s48_ref_t thing;
  thing = s48_make_record_2(call, thing_record_type_binding);
  s48_record_set_2(call, thing, 0, a);
  s48_record_set_2(call, thing, 1, b);
  return thing;
}
```

Note that the interface takes care of protecting all local references against the possibility of a garbage collection occurring during the call to `s48_make_record_2()`; also note that the record type binding is a global reference that is live until explicitly freed.

8.9 Raising exceptions from external code

The following macros explicitly raise certain errors, immediately returning to Scheme 48. Raising an exception performs all necessary clean-up actions to properly return to Scheme 48, including adjusting the stack of protected variables.

The following procedures are available for raising particular types of exceptions. These never return.

- `s48_assertion_violation_2(s48_call_t, const char* who, const char* message, long count`
- `s48_error_2(s48_call_t, const char* who, const char* message, long count, ...)`
- `s48_os_error_2(s48_call_t, const char* who, const char* message, long count, ...)`
- `s48_out_of_memory_error_2(s48_call_t, )`

An assertion violation signaled via `s48_assertion_violation_2` typically means that an invalid argument (or invalid number of arguments) has been passed. An error signaled via `s48_error_2` means that an environmental error (like an I/O error) has occurred. In both cases, `who` indicates the location of the error, typically the name of the function it occurred in. It may be `NULL`, in which the system guesses a name. The `message` argument is an error message encoded in UTF-8. Additional arguments may be passed that become part of the condition object that will be raised on the Scheme side: `count` indicates their number, and the arguments (which must be of type `s48_ref_t`) follow.

The `s48_os_error_2` function is like `s48_error_2`, except that the error message is inferred from an OS error code (as in `strerror`). The `s48_out_of_memory_error_2` function signals that the system has run out of memory.

The following macros raise assertion violations if their argument does not have the required type. `s48_check_boolean_2` raises an error if its argument is neither `#t` or `#f`.

- `void s48_check_boolean_2(s48_call_t, s48_ref_t)`
- `void s48_check_symbol_2(s48_call_t, s48_ref_t)`
- `void s48_check_pair_2(s48_call_t, s48_ref_t)`
- `void s48_check_string_2(s48_call_t, s48_ref_t)`
- `void s48_check_integer_2(s48_call_t, s48_ref_t)`
- `void s48_check_channel_2(s48_call_t, s48_ref_t)`
- `void s48_check_byte_vector_2(s48_call_t, s48_ref_t)`
- `void s48_check_record_2(s48_call_t, s48_ref_t)`
- `void s48_check_shared_binding_2(s48_call_t, s48_ref_t)`

8.10 External events

External code can push the occurrence of external events into the main Scheme 48 event loop and Scheme code can wait and act on external events.

On the Scheme side, the external events functionality consists of the following functions from the structure `primitives`:

- `(new-external-event-uid` *shared-binding-or-#f*`)` → *uid*
- `(unregister-external-event-uid!` *uid*`)`

And the following functions from the structure `external-events`:

- `(register-condvar-for-external-event!` *uid condvar*`)`
- `(wait-for-external-event` *condvar*`)`
- `(new-external-event)` → *uid condvar*

The function `new-external-event-uid` returns a fresh event identifier on every call. When called with a shared binding instead of `#f`, `new-external-event-uid` returns a named event identifier for permanent use. The function `unregister-external-event-uid` unregisters the given event identifier.

External events use condition variables to synchronize the occurrence of events, see section 7.5 for more information on condition variables. The function `register-condvar-for-external-event` registers a condition variable with an event identifier. For convenience, the function `new-external-event` combines `new-external-event-uid` and `register-condvar-for-external-event` and returns a fresh event identifier and the corresponding condition variable.

The function `wait-for-external-event` blocks the caller (on the condition variable) until the Scheme main event loop receives an event notification (by `s48_note_external_event`) of the event identifier that is registered with the given condition variable (with `register-condvar-for-external-event`). There is no guarantee that the caller of `wait-for-external-event` is unblocked on every event notification, therefore the caller has to be prepared to handle multiple external events that have occurred and external code has to be prepared to store multiple external events.

The following prototype is the interface on the external side:

- `void s48_note_external_event(long)`

External code has to collect external events and can use `s48_note_external_event` to signal the occurrence of an external event to the main event loop. The argument to `s48_note_external_event` is an event identifier that was previously registered on the Scheme side. Thus, external code has to obtain the event identifier from the Scheme side, either by passing the event identifier as an argument to the external function that calls `s48_note_external_event` or by exporting the Scheme value to C (see section 8.2.1).

Since the main event loop does not guarantee that every call to `s48_note_external_event` causes the just occurred event to get handled immediately, external code has to make sure that it can collect multiple external events (i.e. keep them in an appropriate data structure). It is safe for external code to call `s48_note_external_event` on every collected external event, though, even if older events have not been handled yet.

8.10.1 Collecting external events in external code

External code has to be able to collect multiple events that have occurred. Therefore, external code has to create the needed data structures to store the information that is associated with the occurred event. Usually, external code collects the events in a thread. An separate thread does not have an call argument, though, so it cannot create Scheme data structures. It must use C data structures to collect the events, for example it can create a linked list of events.

Since the events are later handled on the Scheme side, the information associated with the event needs to be visible on the Scheme side, too. Therefore, external code exports a function to Scheme that returns all current events as Scheme objects (the function that returns the events knows about the current call and thus can create Scheme objects). Scheme and external code might need to share Scheme record types that represent the event information. Typically, the function that returns the events converts the C event list into a Scheme event list by preserving the original order in which the events arrived. Note that the external list data structure that holds all events needs to be mutex locked on each access to preserve thread-safe manipulation of the data structure (the Scheme thread that processes events and the external thread that collects events may access the data structures at the same time).

8.10.2 Handling external events in Scheme

If the sole occurrence of an event does not suffice for the program, the Scheme side has to pull the information that is associated with an event from the C side. Then, the Scheme side can handle the event data. For example, a typical event loop on the Scheme side that waits on external events of an permanent event type that an long-running external thread produces may look like this:

```
(define *external-event-uid*
  (new-external-event-uid (lookup-imported-binding "my-event")))

(spawn-external-thread *external-event-uid*)

(let loop ()
  (let ((condvar (make-condvar)))
    (register-condvar-for-external-event! *external-event-uid* condvar)
    (wait-for-external-event condvar)
    (process-external-events! (get-external-events))
    (loop)))
```

In the above example, the variable `*external-event-uid*` is defined as a permanent event identifier. On every pass through the loop, a fresh condition variable is registered with the event identifier, then `wait-for-external-event` blocks on the condition variable until external code signals the occurrence of a matching event. Note that `process-external-events!` and `get-external-events` need to be defined by the user. The user-written function `get-external-events` returns all the events that the external code has collected since the last time `get-external-events` was called; the user-written function `process-external-events!` handles the events on the Scheme side.

When the Scheme side only waits for one single event, there is no need for an event loop and an permanent event identifier. Then, `new-external-event` is more convenient to use:

```
(call-with-values
  (lambda () (new-external-event))
  (lambda (uid condvar)
    (spawn-external-thread uid)
    (wait-for-external-event condvar)
    (unregister-external-event-uid! uid)
    ...))
```

Here, `new-external-event` returns a fresh event identifier and a fresh condition variable. The event identifier is passed to `spawn-external-thread` and the condition variable is used to wait for the occurrence of the external event.

External code uses `s48_note_external_event` to push the fact that an external event occurred into the main event loop, then the Scheme code needs to pull the actual event data from external code (in this example with `get-external-events`). The user-written function `spawn-external-thread` runs the external code that informs the Scheme side about the occurrence of external events. The event identifier is passed as an argument. The external-event-related parts of the implementation of `spawn-external-thread` in external code could look like this:

```
s48_ref_t
spawn_external_thread(s48_call_t call, s48_ref_t sch_event_uid) {
  ...
  s48_note_external_event(s48_extract_long_2(call, sch_event_uid));
  ...
}
```

The event identifier is extracted from its Scheme representation and used to inform the Scheme side about an occurrence of this specific event type.

8.11 Unsafe functions and macros

All of the C procedures and macros described above check that their arguments have the appropriate types and that indexes are in range. The following procedures and macros are identical to those described above, except that they do not perform type and range checks. They are provided for the purpose of writing more efficient code; their general use is not recommended.

- `long      s48_unsafe_extract_char_2(s48_call_t, s48_ref_t)`
- `s48_ref_t s48_unsafe_enter_char_2(s48_call_t, long)`
- `long      s48_unsafe_extract_integer_2(s48_call_t, s48_ref_t)`
- `long      s48_unsafe_extract_double_2(s48_call_t, s48_ref_t)`

- `long      s48_unsafe_extract_fixnum_2(s48_call_t, s48_ref_t)`
- `s48_ref_t s48_unsafe_enter_fixnum_2(s48_call_t, long)`

- `s48_ref_t s48_unsafe_car_2(s48_call_t, s48_ref_t)`
- `s48_ref_t s48_unsafe_cdr_2(s48_call_t, s48_ref_t)`
- `void      s48_unsafe_set_car_2(s48_call_t, s48_ref_t, s48_ref_t)`
- `void      s48_unsafe_set_cdr_2(s48_call_t, s48_ref_t, s48_ref_t)`

- `long      s48_unsafe_vector_length_2(s48_call_t, s48_ref_t)`
- `s48_ref_t s48_unsafe_vector_ref_2(s48_call_t, s48_ref_t, long)`
- `void      s48_unsafe_vector_set_2(s48_call_t, s48_ref_t, long, s48_ref_t)`

- `long      s48_unsafe_string_length_2(s48_call_t, s48_ref_t)`
- `char      s48_unsafe_string_ref_2(s48_call_t, s48_ref_t, long)`
- `void      s48_unsafe_string_set_2(s48_call_t, s48_ref_t, long, char)`

- `s48_ref_t s48_unsafe_symbol_to_string_2(s48_call_t, s48_ref_t)`

- `char *    s48_unsafe_extract_byte_vector_2(s48_call_t, s48_ref_t)`

- `long      s48_unsafe_byte_vector_length_2(s48_call_t, s48_ref_t)`
- `char      s48_unsafe_byte_vector_ref_2(s48_call_t, s48_ref_t, long)`
- `void      s48_unsafe_byte_vector_set_2(s48_call_t, s48_ref_t, long, int)`

Additionally to not performing type checks, the pointer returned by `s48_unsafe_extract_byte_vector_2` will likely become invalid when a garbage collection occurs. See section8.7.3 on how the interface deals with byte vectors in a proper way.

- `s48_ref_t s48_unsafe_shared_binding_ref_2(s48_call_t, s48_ref_t s_b)`
- `int       s48_unsafe_shared_binding_p_2(s48_call_t, x)`
- `int       s48_unsafe_shared_binding_is_import_p_2(s48_call_t, s48_ref_t s_b)`
- `s48_ref_t s48_unsafe_shared_binding_name_2(s48_call_t, s48_ref_t s_b)`
- `void      s48_unsafe_shared_binding_set_2(s48_call_t, s48_ref_t s_b, s48_ref_t value)`

- `s48_ref_t s48_unsafe_record_type_2(s48_call_t, s48_ref_t)`
- `s48_ref_t s48_unsafe_record_ref_2(s48_call_t, s48_ref_t, long)`
- `void      s48_unsafe_record_set_2(s48_call_t, s48_ref_t, long, s48_ref_t)`

- `type      s48_unsafe_extract_value_2(s48_call_t, s48_ref_t, type)`
- `type *    s48_unsafe_extract_value_pointer_2(s48_call_t, s48_ref_t, type)`
- `void      s48_unsafe_set_value_2(s48_call_t, s48_ref_t, type, value)`

8.12 Converting external code to the new foreign-function interface

It is straightforward to convert external code from the old foreign-function interface to the new foreign-function interface:

- Converting functions:
  - Add `s48_call call` as a first argument to every function prototype that returns or accepts a `s48_value`.
  - Replace every `s48_value` type in the function prototype and the body with `s48_ref_t`.
  - Add `call` as the first argument to every function call that returns or accepts a Scheme object.
  - Remove all the `GCPROTECT`-related code (i.e. `GCPROTECT` and `UNPROTECT`).
- Converting global (static) variables:
  - Replace `s48_value` type of the global variable with `s48_ref_t`, initialize these variables with `NULL`.
  - Set a real Scheme object in the initialization function of your code with one of these alternatives:
    * Use `s48_make_global_ref` to convert a `s48_value` to a global reference. For details and an example see section 8.7.1.

    * Use `s48_local_to_global_ref` to convert a local reference object to a global one.
    * If your global variable is supposed to hold a shared binding (e.g. an record type binding), you can use `s48_get_imported_binding_2` that returns a global reference.
  - Replace `S48_GC_PROTECT_GLOBAL` with `s48_local_to_global_ref` to convert a local reference object to a global one.
  - Use `s48_free_global_ref` to cleanup global references when appropriate.

If you add `#define NO_OLD_FFI 1` just above `#include <scheme48.h>` in your source code file, it will hide all the macros and prototype definitions of the old foreign-function interface. That way you can make sure that you are only using the new interface and the C compiler will remind you if you don't.

Chapter 9

Access to POSIX

This chapter describes Scheme 48's interface to the POSIX C calls [1]. Scheme versions of most of the functions in POSIX are provided. Both the interface and implementation are new and are likely to change in future releases. Section 9.12 lists which Scheme functions call which C functions.

Scheme 48's POSIX interface will likely change significantly in the future. The implementation is new and may have significant bugs.

The POSIX bindings are available in several structures:

`posix-processes`	fork, exec, and friends
`posix-process-data`	information about processes
`posix-files`	files and directories
`posix-i/o`	operations on ports
`posix-time`	time functions
`posix-users`	users and groups
`posix-regexps`	regular expression matching
`posix-syslog`	POSIX logging facility
`posix-errnos`	POSIX error codes
`posix`	all of the above

Scheme 48's POSIX interface differs from Scsh's [11, 12] in several ways. The interface here lacks Scsh's high-level constructs and utilities, such as the process notation, `awk` procedure, and parsing utilities. Scheme 48 uses distinct types for some values that Scsh leaves as symbols or unboxed integers; these include file types, file modes, and user and group ids. Many of the names and other interface details are different, as well.

9.1 Process primitives

The procedures described in this section control the creation of processes and the execution of programs. They are in the structures `posix-process` and `posix`.

9.1.1 Process creation and termination

- `(fork)` → *process-id or* `#f`
- `(fork-and-forget` *thunk*`)`

`Fork` creates a new child process and returns the child's process-id in the parent and `#f` in the child. `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.

- `(process-id?` *x*`)` → *boolean*
- `(process-id=?` *process-id0 process-id1*`)` → *boolean*
- `(process-id->integer` *process-id*`)` → *integer*
- `(integer->process-id` *integer*`)` → *process-id*

`Process-id?` is a predicate for process-ids, `process-id=?` compares two to see if they are the same, and `process-id-uid` returns the actual Unix id. `Process-id->integer` and `integer->process-id` convert process ids to and from integers.

- `(process-id-exit-status` *process-id*`)` → *integer or* `#f`
- `(process-id-terminating-signal` *process-id*`)` → *signal or* `#f`
- `(wait-for-child-process` *process-id*`)`

If a process terminates normally `process-id-exit-status` will return its exit status. If the process is still running or was terminated by a signal then `process-id-exit-status` will return `#f`. Similarly, if a child process was terminated by a signal `process-id-terminating-signal` will return that signal and will return `#f` if the process is still running or terminated normally. `Wait-for-child-process` blocks until the child process terminates. Scheme 48 may reap child processes before the user requests their exit status, but it does not always do so.

- `(exit` *status*`)`

Terminates the current process with the integer *status* as its exit status.

9.1.2 Exec

- `(exec` *program-name arg0 ...*`)`
- `(exec-with-environment` *program-name env arg0 ...*`)`
- `(exec-file` *filename arg0 ...*`)`
- `(exec-file-with-environment` *filename env arg0 ...*`)`
- `(exec-with-alias` *name lookup? maybe-env arguments*`)`

All of these replace the current program with a new one. They differ in how the new program is found, what its environment is, and what arguments it is passed. `Exec` and `exec-with-environment` look up the new program in the search path, while `exec-file` and `exec-file-with-environment` execute a particular file. The environment is either inherited from the current process (`exec` and `exec-file`) or given as an argument (`...-with-environment`).

Program-name and *filename* and any arg_i should be *os-string-thing* arguments (see section 5.15. *Env* should be a list of *os-string-thing* arguments of the form `"`*name*`=`*value*`"`. The first four procedures add their first argument, *program-name* or *filename*, before the *arg0* ... arguments.

`Exec-with-alias` is an omnibus procedure that subsumes the other four. *Name* is looked up in the search path if *lookup?* is true and is used as a filename otherwise. *Maybe-env* is either a list of *os-string-thing*s for the environment of the new program or `#f` in which case the new program inherits its environment from the current one. *Arguments* should be a list of *os-string-thing*s; unlike with the other four procedures, *name* is not added to this list (hence `-with-alias`).

9.2 Signals

There are two varieties of signals available, *named* and *anonymous*. A named signal is one for which we have a symbolic name, such as `kill` or `pipe`. Anonymous signals, for which we only have the current operating system's signal number, have no meaning in other operating systems. Named signals preserve their meaning in image files. Not all named signals are available from all OS's and there may be multiple names for a single OS signal number.

- `(signal` *signal-name*`)` → *signal* syntax
- `(name->signal` *symbol*`)` → *signal or* `#f`
- `(integer->signal` *integer*`)` → *signal*
- `(signal?` *x*`)` → *boolean*
- `(signal-name` *signal*`)` → *symbol or* `#f`
- `(signal-os-number` *signal*`)` → *integer*
- `(signal=?` *signal0 signal1*`)` → *boolean*

The syntax `signal` returns a (named) signal associated with *signal-name*. `Name->signal` returns a (named) signal or `#f` if the the signal *name* is not supported by the operating system. The signal returned by `integer->signal` is a named signal if *integer* corresponds to a named signal in the current operating system; otherwise it returns an anonymous signal. `Signal-name` returns a symbol if *signal* is named and `#f` if it is anonymous. `Signal=?` returns `#t` if *signal0* and *signal1* have the same operating system number and `#f` if they do not.

9.2.1 POSIX signals

The following lists the names of the POSIX signals.

`abrt`	abort - abnormal termination (as by abort())
`alrm`	alarm - timeout signal (as by alarm())
`fpe`	floating point exception
`hup`	hangup - hangup on controlling terminal or death of controlling process
`ill`	illegal instruction
`int`	interrupt - interaction attention
`kill`	kill - termination signal, cannot be caught or ignored
`pipe`	pipe - write on a pipe with no readers
`quit`	quit - interaction termination
`segv`	segmentation violation - invalid memory reference
`term`	termination - termination signal
`usr1`	user1 - for use by applications
`usr2`	user2 - for use by applications
`chld`	child - child process stopped or terminated
`cont`	continue - continue if stopped
`stop`	stop - cannot be caught or ignored
`tstp`	interactive stop
`ttin`	read from control terminal attempted by background process
`ttou`	write to control terminal attempted by background process
`bus`	bus error - access to undefined portion of memory

9.2.2 Other signals

The following lists the names of the non-POSIX signals that the system is currently aware of.

`trap`	trace or breakpoint trap
`iot`	IOT trap - a 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`	A synonym for SIGIO (System V)
`cld`	A synonym for SIGCHLD
`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`	A synonym for SIGPWR
`lost`	File lock lost
`winch`	Window resize signal (4.3 BSD, Sun)
`unused`	Unused signal

9.2.3 Sending signals

- `(signal-process` *process-id signal*`)`

Send *signal* to the process corresponding to *process-id*.

9.2.4 Receiving signals

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 read from the signal-queue. When the Scheme process receives a signal that signal is added to the received-signal queues of all signal-queues which are currently monitoring that particular signal.

- `(make-signal-queue` *signals*`)` $\to$ *signal-queue*
- `(signal-queue?` *x*`)` $\to$ *boolean*
- `(signal-queue-monitored-signals` *signal-queue*`)` $\to$ *list of signals*
- `(dequeue-signal!` *signal-queue*`)` $\to$ *signal*
- `(maybe-dequeue-signal!` *signal-queue*`)` $\to$ *signal or* `#f`

`Make-signal-queue` returns a new signal-queue that will monitor the signals in the list *signals*. `Signal-queue?` is a predicate for signal queues. `Signal-queue-monitored-signals` returns a list of the signals currently monitored by *signal-queue*. `Dequeue-signal!` and `maybe-dequeue-signal` both return the next received-but-unread signal from *signal-queue*. If *signal-queue*'s queue of signals is empty `dequeue-signal!` blocks until an appropriate signal is received. `Maybe-dequeue-signal!` does not block; it returns `#f` instead.

There is a bug in the current system that causes an erroneous deadlock error if threads are blocked waiting for signals and no other threads are available to run. A work around 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*`)`
- `(remove-signal-queue-signal!` *signal-queue signal*`)`

These two procedures can be used to add or remove signals from a signal-queue's list of monitored signals. When a signal is removed from a signal-queue's list of monitored signals any occurances of the signal are removed from that signal-queue's pending signals. In other words, `dequeue-signal!` and `maybe-dequeue-signal!` will only return signals that are currently on the signal-queue's list of signals.

9.3 Process environment

These are in structures `posix-process-data` and `posix`.

9.3.1 Process identification

- `(get-process-id)` → *process-id*
- `(get-parent-process-id)` → *process-id*

These return the process ids of the current process and its parent. See section 9.1.1 for operations on process ids.

- `(get-user-id)` → *user-id*
- `(get-effective-user-id)` → *user-id*
- `(set-user-id!` *user-id*`)`
- `(set-effective-user-id!` *user-id*`)`

- `(get-group-id)` → *group-id*
- `(get-effective-group-id)` → *group-id*
- `(set-group-id!` *group-id*`)`
- `(set-effective-group-id!` *group-id*`)`

Every process has both the original and effective user id and group id. The effective values may be set, but the original ones can only be set if the process has appropriate privelages.

- `(get-groups)` → *group-ids*
- `(get-login-name)` → *os-string*

`Get-groups` returns a list of the supplementary groups of the current process. `Get-login-name` returns a user name for the current process.

9.3.2 Environment variables

- `(lookup-environment-variable` *os-string-thing*`)` → *os-string or* `#f`
- `(set-environment-variable!` *name value*`)`
- `(environment-alist)` → *alist*

`Lookup-environment-variable` looks up its argument in the environment list and returns the corresponding value or `#f` if there is none. `Set-environment-variable!` sets the value of `name` in the environment list to `value`. If `name` is not already an environment variable, it's created. If it already exists, its value is overwritten with `value`. Both arguments must be os-string-things. `Environment-alist` returns the entire environment as a list of `(`*name-os-string* `.` *value-os-string*`)` pairs.

9.4 Users and groups

*User-id*s and *group-id*s are boxed integers representing Unix users and groups. The procedures in this section are in structures `posix-users` and `posix`.

- `(user-id?` *x*`)` → *boolean*
- `(user-id=?` *user-id0 user-id1*`)` → *boolean*
- `(user-id->integer` *user-id*`)` → *integer*
- `(integer->user-id` *integer*`)` → *user-id*

- `(group-id?` *x*`)` → *boolean*
- `(group-id=?` *group-id0 group-id1*`)` → *boolean*
- `(group-id->integer` *group-id*`)` → *integer*
- `(integer->group-id` *integer*`)` → *group-id*

User-ids and group-ids have their own own predicates and comparison, boxing, and unboxing functions.

- `(user-id->user-info` *user-id*`)` → *user-info*
- `(name->user-info` *os-string*`)` → *user-info*

These return the user info for a user identified by user-id or name.

- `(user-info?` *x*`)` → *boolean*
- `(user-info-name` *user-info*`)` → *os-string*
- `(user-info-id` *user-info*`)` → *user-id*
- `(user-info-group` *user-info*`)` → *group-id*
- `(user-info-home-directory` *user-info*`)` → *os-string*
- `(user-info-shell` *user-info*`)` → *os-string*

A `user-info` contains information about a user. Available are the user's name, id, group, home directory, and shell.

- `(group-id->group-info` *group-id*`)` → *group-info*
- `(name->group-info` *os-string*`)` → *group-info*

These return the group info for a group identified by group-id or name.

- `(group-info?` *x*`)` → *boolean*
- `(group-info-name` *group-info*`)` → *os-string*
- `(group-info-id` *group-info*`)` → *group-id*
- `(group-info-members` *group-info*`)` → *user-ids*

A `group-info` contains information about a group. Available are the group's name, id, and a list of members.

9.5 OS and machine identification

These procedures return strings that are supposed to identify the current OS and machine. The POSIX standard does not indicate the format of the strings. The procedures are in structures `posix-platform-names` and `posix`.

- `(os-name)` → *string*
- `(os-node-name)` → *string*
- `(os-release-name)` → *string*
- `(os-version-name)` → *string*
- `(machine-name)` → *string*

9.6 Files and directories

These procedures are in structures `posix-files` and `posix`.

9.6.1 Directory streams

Directory streams are like input ports, with each read operation returning the next name in the directory.

- `(open-directory-stream` *name*`)` → *directory*
- `(directory-stream?` *x*`)` → *boolean*
- `(read-directory-stream` *directory*`)` → *name or* `#f`
- `(close-directory-stream` *directory*`)`

`Open-directory-stream` opens a new directory stream. `Directory-stream?` is a predicate that recognizes directory streams. `Read-directory-stream` returns the next name in the directory or `#f` if all names have been read. `Close-directory-stream` closes a directory stream.

- `(list-directory` *name*`)` → *list of os-strings*

This is the obvious utility; it returns a list of the names in directory *name*.

9.6.2 Working directory

- `(working-directory)` → *os-string*
- `(set-working-directory!` *os-string-thing*`)`

These return and set the working directory.

9.6.3 File creation and removal

- `(open-file` *path file-options*`)` → *port*
- `(open-file` *path file-options file-mode*`)` → *port*

`Open-file` opens a port to the file named by *path*, which must be a *os-string-thing* argument. The *file-options* argument determines various aspects of the returned port. The optional *file-mode* argument is used only if the file to be opened does not already exist. The returned port is an input port if *file-options* includes `read-only`; otherwise it returns an output port. `Dup-switching-mode` can be used to open an input port for output ports opened with the `read/write` option.

- `(file-options` *file-option-name* `...)` → *file-options* syntax
- `(file-options-on?` *file-options file-options*`)` → *boolean*
- `(file-options-union` *file-options file-options*`)` → *file-options*

The syntax `file-options` returns a file-option with the indicated options set. `File-options-on?` returns true if its first argument includes all of the options listed in the second argument. `File-options-union` returns a file-options argument containing exactly all of the options listed in either argument. The following file options may be used with `open-file`.

`create`	create file if it does not already exist; a file-mode argument is required with this option
`exclusive`	an error will be raised if this option and `create` are both set and the file already exists
`no-controlling-tty`	if *path* is a terminal device this option causes the terminal to not become the controlling terminal of the process
`truncate`	file is truncated
`append`	writes are appended to existing contents
`nonblocking`	read and write operations do not block
`read-only`	port may not be written
`read-write`	file descriptor may be read or written
`write-only`	port may not be read

Only one of the last three options may be used. If `read-write` is specified, an output port is returned.

For example

```
(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 by anyone and written only by the owner. Once the file exists,

```
(open-file "some-file.txt"
           (file-options append write-only))
```

will open an output port that appends to the file.

The `append` and `nonblocking` options and the read/write nature of the port can be read using `i/o-flags`. The `append` and `nonblocking` options can be set using `set-i/o-flags!`.

To keep port operations from blocking the Scheme 48 process, output ports are set to be nonblocking at the time of creation (input ports are managed using `select()`). You can use `set-i/o-flags!` to make an output port blocking, for example just before a fork, but care should be exercised. The Scheme 48 runtime code may get confused if an I/O operation blocks.

• `(set-file-creation-mask!` *file-mode*`)`

Sets the file creation mask to be *file-mode*. Bits set in *file-mode* are cleared in the modes of any files or directories created by the current process.

• `(link` *existing new*`)`

Both *existing* and *new* must be *os-string-thing* arguments. `Link` makes path *new* be a new link to the file pointed to by path *existing*. The two paths must be in the same file system.

• `(make-directory` *path file-mode*`)`

- `(make-fifo` *path file-mode*`)`

These two procedures make new directories and fifo files. In both cases, *path* must be a *os-string-thing* argument.

- `(unlink` *path*`)`
- `(remove-directory` *path*`)`
- `(rename` *old-path new-path*`)`

Path, *old-path* and *new-path* must all be *os-string-thing* arguments. `Unlink` removes the link indicated by *path*. `Remove-directory` removes the indicated (empty) directory. `Rename` moves the file pointed to by *old-path* to the location pointed to by *new-path* (the two paths must be in the same file system). Any other links to the file remain unchanged.

- `(accessible?` *path access-mode* `.` *more-modes*`)` → *boolean*
- `(access-mode` *mode-name*`)` → *access-mode* syntax

`Accessible?` returns true if *path* (which must be a *os-string-thing* argument) is a file that can be accessed in the listed mode. If more than one mode is specified `accessible?` returns true if all of the specified modes are permitted. The *mode-name*s are: `read`, `write`, `execute`, `exists`.

9.6.4 File information

- `(get-file-info` *path*`)` → *file-info*
- `(get-file/link-info` *name*`)` → *file-info*
- `(get-port-info` *fd-port*`)` → *file-info*

`Get-file-info` and `get-file/link-info` both return a file info record for the file named by *path*, which must be a *os-string-thing* argument. `Get-file-info` follows symbolic links while `get-file/link-info` does not. `Get-port-info` returns a file info record for the file which *port* reads from or writes to. An error is raised if *fd-port* does not read from or write to a file descriptor.

- `(file-info?` *x*`)` → *boolean*
- `(file-info-name` *file-info*`)` → *os-string*

`File-info?` is a predicate for file-info records. `File-info-name` is the name which was used to get `file-info`, either as passed to `get-file-info` or `get-file/link-info`, or used to open the port passed to `get-port-info`.

- `(file-info-type` *file-info*`)` → *file-type*
- `(file-type?` *x*`)` → *boolean*
- `(file-type-name` *file-type*`)` → *symbol*
- `(file-type` *type*`)` → *file-type* syntax

`File-info-type` returns the type of the file, as a file-type object File types may be compared using `eq?`. The valid file types are:

```
regular
directory
character-device
block-device
fifo
symbolic-link
socket
other
```

`Symbolic-link` and `socket` are not required by POSIX.

- `(file-info-device` *file-info*`)` $\rightarrow$ *integer*
- `(file-info-inode` *file-info*`)` $\rightarrow$ *integer*

The device and inode numbers uniquely determine a file.

- `(file-info-link-count` *file-info*`)` $\rightarrow$ *integer*
- `(file-info-size` *file-info*`)` $\rightarrow$ *integer*

These return the number of links to a file and the file size in bytes. The size is only meaningful for regular files.

- `(file-info-owner` *file-info*`)` $\rightarrow$ *user-id*
- `(file-info-group` *file-info*`)` $\rightarrow$ *group-id*
- `(file-info-mode` *file-info*`)` $\rightarrow$ *file-mode*

These return the owner, group, and access mode of a file.

- `(file-info-last-access` *file-info*`)` $\rightarrow$ *time*
- `(file-info-last-modification` *file-info*`)` $\rightarrow$ *time*
- `(file-info-last-status-change` *file-info*`)` $\rightarrow$ *time*

These return the time the file was last read, modified, or had its status modified.

9.6.5 File modes

A file mode is a boxed integer representing a file protection mask.

- `(file-mode permission-name ...)` $\rightarrow$ *file-mode* syntax
- `(file-mode?` *x*`)` $\rightarrow$ *boolean*
- `(file-mode+` *file-mode …*`)` $\rightarrow$ *file-mode*
- `(file-mode-` *file-mode0 file-mode1*`)` $\rightarrow$ *file-mode*

`File-mode` is syntax for creating file modes. The mode-names are listed below. `File-mode?` is a predicate for file modes. `File-mode+` returns a mode that contains all of permissions of its arguments. `File-mode-` returns a mode that has all of the permissions of *file-mode0* that are not in *file-mode1*.

- `(file-mode=?` *file-mode0 file-mode1*`)` $\rightarrow$ *boolean*
- `(file-mode<=?` *file-mode0 file-mode1*`)` $\rightarrow$ *boolean*
- `(file-mode>=?` *file-mode0 file-mode1*`)` $\rightarrow$ *boolean*

`File-mode=?` returns true if the two modes are exactly the same. `File-mode<=?` returns true if *file-mode0* has a subset of the permissions of *file-mode1*. `File-mode>=?` is `file-mode<=?` with the arguments reversed.

- `(file-mode->integer` *file-mode*`)` $\rightarrow$ *integer*
- `(integer->file-mode` *integer*`)` $\rightarrow$ *file-mode*

`Integer->file-mode` and `file-mode->integer` translate file modes to and from the classic Unix file mode masks. These may not be the masks used by the underlying OS.

Permission name	Bit mask	
`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

Names for sets of permissions		
`owner`	`#o0700`	read, write, and execute by owner
`group`	`#o0070`	read, write, and execute by group
`other`	`#o0007`	read, write, and execute by others
`read`	`#o0444`	read by anyone
`write`	`#o0222`	write by anyone
`exec`	`#o0111`	execute by anyone
`all`	`#o0777`	anything by anyone

9.6.6 Symbolic links

- `(create-symbolic-link` *path1 path2*`)`

This creates a symbolic link at *path2* that contains *path1*. *Path1* and *path2* must be *os-string-thing* arguments.

- `(read-symbolic-link` *path*`)` $\rightarrow$ *os-string*

This returns contents of the symbolic link at *path*. *Path* must be an *os-string-thing* argument.

9.7 Time and Date

These procedures are in structures `posix-time` and `posix`.

9.7.1 Time

- `(make-time` *integer*`)` → *time*
- `(current-time)` → *time*
- `(time?` *x*`)` → *boolean*
- `(time-seconds` *time*`)` → *integer*

A `time` record contains an integer that represents time as the number of second since the Unix epoch (00:00:00 GMT, January 1, 1970). `Make-time` and `current-time` return `time`s, with `make-time`'s using its argument while `current-time`'s has the current time. `Time?` is a predicate that recognizes `time`s and `time-seconds` returns the number of seconds *time* represents.

- `(time=?` *time time*`)` → *boolean*
- `(time<?` *time time*`)` → *boolean*
- `(time<=?` *time time*`)` → *boolean*
- `(time>?` *time time*`)` → *boolean*
- `(time>=?` *time time*`)` → *boolean*

These perform various comparison operations on the `time`s.

- `(time->string` *time*`)` → *string*

`Time->string` returns a string representation of *time* in the locale's version of the following form.

```
"Wed Jun 30 21:49:08 1993
"
```

9.7.2 Date

A date is a time specification relative to a specific but implicit time zone, broken out into the familar year-month-day-hour-minute-second format.

- `(make-date` *second minute hour month-day month year week-day year-day dst*`)` → *date*
- `(date?` *x*`)` → *boolean*
- `(date-second` *date*`)` → *second*
- `(date-minute` *date*`)` → *minute*
- `(date-hour` *date*`)` → *hour*
- `(date-month-day` *date*`)` → *month-day*
- `(date-month` *date*`)` → *month*
- `(date-year` *date*`)` → *year*
- `(date-week-day` *date*`)` → *week-day*
- `(date-year-day` *date*`)` → *year-day*
- `(date-date-dst` *date*`)` → *dst*

These are the constructor, predicate and corresponding accessors for date objects. The meaning of the various field types are as follows:

seconds	seconds	(0–60)
minutes	minutes	(0–59)
hour	hours	(0–23)
month-day	day of month	(1–31)
mon	month of year	(0–11)
year	year	since 1900
week-day	day of week	(Sunday = 0)
year-day	day of year	(0–365)
dst	is summer time in effect?	`#t`, `#f` or unspecific

• `(date->string` *date*`)` → *string*

returns a string representation of *date* in the locale's version of the following form:

```
"Wed Jun 30 21:49:08 1993
"
```

• `(time->utc-date` *time*`)` → *date*
• `(time->local-date` *time*`)` → *date*

These convert a time object into a date object; the first does this relative to the UTC time zone, the second relative to the current timezone setting.

• `(date->time` *date*`)` → *time*

This converts a date object into a time object relative to the current timezone setting.

• `(format-date` *string date*`)` → *string*

This formats a date into a string, according to the format specification in the first argument. The format specification is according to the specification of the C `strftime` function:

`%a`	is replaced by the locale's abbreviated weekday name.
`%A`	is replaced by the locale's full weekday name.
`%b`	is replaced by the locale's abbreviated month name.
`%B`	is replaced by the locale's full month name.
`%c`	is replaced by the locale's appropriate date and time representation.
`%C`	is replaced by the year divided by 100 and truncated to an integer, as a decimal number (00–99).
`%d`	is replaced by the day of the month as a decimal number (01–31).
`%D`	is equivalent to "`%m/%d/%y`".
`%e`	is replaced by the day of the month as a decimal number (1–31); a single digit is preceded by a space.
`%F`	is equivalent to "`%Y-%m-%d`" (the ISO 8601 date format).
`%g`	is replaced by the last 2 digits of the week-based year (see below) as a decimal number (00–99).
`%G`	is replaced by the week-based year (see below) as a decimal number (e.g., 1997).

`%h`	is equivalent to “`%b`”.
`%H`	is replaced by the hour (24-hour clock) as a decimal number (00–23).
`%I`	is replaced by the hour (12-hour clock) as a decimal number (01–12).
`%j`	is replaced by the day of the year as a decimal number (001–366).
`%m`	is replaced by the month as a decimal number (01–12).
`%M`	is replaced by the minute as a decimal number (00–59).
`%n`	is replaced by a new-line character.
`%p`	is replaced by the locale’s equivalent of the AM/PM designations associated with a 12-hour clock.
`%r`	is replaced by the locale’s 12-hour clock time.
`%R`	is equivalent to “`%H:%M`”.
`%S`	is replaced by the second as a decimal number (00–60).
`%t`	is replaced by a horizontal-tab character.
`%T`	is equivalent to “`%H:%M:%S`” (the ISO 8601 time format).
`%u`	is replaced by the ISO 8601 weekday as a decimal number (1–7), where Monday is 1.
`%U`	is replaced by the week number of the year (the first Sunday as the first day of week 1) as a decimal number (00–53).
`%V`	is replaced by the ISO 8601 week number (see below) as a decimal number (01–53).
`%w`	is replaced by the weekday as a decimal number (0–6), where Sunday is 0.
`%W`	is replaced by the week number of the year (the first Monday as the first day of week 1) as a decimal number (00–53).
`%x`	is replaced by the locale’s appropriate date representation.
`%X`	is replaced by the locale’s appropriate time representation.
`%y`	is replaced by the last 2 digits of the year as a decimal number (00–99).
`%Y`	is replaced by the year as a decimal number (e.g., 1997).
`%z`	is replaced by the offset from UTC in the ISO 8601 format “`-0430`” (meaning 4 hours 30 minutes behind UTC, west of Greenwich), or by no characters if no time zone is determinable.
`%Z`	is replaced by the locale’s time zone name or abbreviation, or by no characters if no time zone is determinable.
`%%`	is replaced by `%`.

9.8 I/O

These procedures are in structures `posix-i/o` and `posix`.

• `(open-pipe)` $\rightarrow$ *input-port + output-port*

`Open-pipe` creates a new pipe and returns the two ends as an input port and an output port.

A *file descriptor* port (or *fd-port*) is a port that reads to or writes from an OS file descriptor. Fd-ports are returned by `open-input-file`, `open-output-file`, `open-file`, `open-pipe`, and other procedures.

- `(fd-port? `*port*`)` → *boolean*
- `(port->fd `*port*`)` → *integer or* `#f`

`Fd-port?` returns true if its argument is an fd-port. `Port->fd` returns the file descriptor associated with or `#f` if *port* is not an fd-port.

- `(remap-file-descriptors `*fd-spec ...*`)`

`Remap-file-descriptors` reassigns file descriptors to ports. The *fd-specs* indicate which port is to be mapped to each file descriptor: the first gets file descriptor `0`, the second gets `1`, and so forth. A *fd-spec* is either a port that reads from or writes to a file descriptor, or `#f`, with `#f` indicating that 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` and the current input port to file descriptor `2`.

- `(dup `*fd-port*`)` → *fd-port*
- `(dup-switching-mode `*fd-port*`)` → *fd-port*
- `(dup2 `*fd-port file-descriptor*`)` → *fd-port*

These change *fd-port*'s file descriptor and return a new port that uses *ports*'s old file descriptor. `Dup` uses the lowest unused file descriptor and `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-but `*port ...*`)`

`Close-all-but` closes all file descriptors whose associated ports are not passed to it as arguments.

- `(close-on-exec? `*port*`)` → *boolean*
- `(set-close-on-exec?! `*port boolean*`)`

`Close-on-exec?` returns true if `port` will be closed when a new program is exec'ed. `Set-close-on-exec?!` sets `port`'s close-on-exec flag.

- `(i/o-flags `*port*`)` → *file-options*
- `(set-i/o-flags! `*port file-options*`)`

These two procedures read and write various options for `port`. The options that can be read are `append`, `nonblocking`, `read-only`, `write-only`, and `read/write`. Only the `append` and `nonblocking` can be written.

- `(port-is-a-terminal? `*port*`)` → *boolean*
- `(port-terminal-name `*port*`)` → *string*

`Port-is-a-terminal?` returns true if *port* has an underlying file descriptor that is associated with a terminal. For such ports `port-terminal-name` returns the name of the terminal, for all others it returns `#f`.

9.9 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. Due to limitations in the underlying facility, only Latin-1 strings are guaranteed to work here—on some platforms, only ASCII may function correctly. Moreover, because the C interface uses zero bytes for marking the ends of strings, patterns and strings that contain zero bytes will not work correctly.

These procedures are in structures `posix-regexps` and `posix`.

An abstract data type for creating POSIX regular expressions is described in section 5.21.

- `(make-regexp` *string . regexp-options*`)` → *regexp*
- `(regexp-option` *option-name*`)` → *regexp-option* syntax

`Make-regexp` makes a new regular expression, using *string* as the pattern. The possible option names are:

`extended`	use the extended patterns
`ignore-case`	ignore case when matching
`submatches`	report submatches
`newline`	treat newlines specially

The regular expression is not compiled until it matched against a string, so any errors in the pattern string will not be reported until that point.

- `(regexp?` *x*`)` → *boolean*

This is a predicate for regular expressions.

- `(regexp-match` *regexp string start submatches? starts-line? ends-line?*`)`
  → *boolean or list of matches*
- `(match?` *x*`)` → *boolean*
- `(match-start` *match*`)` → *integer*
- `(match-end` *match*`)` → *integer*

`Regexp-match` matches the regular expression 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` is returned if it is not. Each match record contains the index of the character at the beginning of the match and one more than the index of the character at the end. The first match record gives the location of the substring that matched *regexp*. If the pattern in *regexp* contained submatches, then the results of these are returned in order, with a match records reporting submatches that succeeded and `#f` in place of those that did not.

Starts-line? should be true if *string* starts at the beginning of a line and *ends-line?* should be true if it ends one.

9.10 Syslog facility

The procedures in this section provide access to the POSIX syslog facility. The functionality is in a structure called `posix-syslog`. The Scheme 48 interface to the syslog facility differs significantly from that of the Unix library functionality in order to support multiple simultaneous connections to the syslog facility.

Log messages carry a variety of parameters beside the text of the message itself, namely a set of options controlling the output format and destination, the facility identifying the class of programs the message is coming from, an identifier specifying the concrete program, and the level identifying the importance of the message. Moreover, a log mask can prevent messages at certain levels to be actually sent to the syslog daemon.

Log options

A log option specifies details of the I/O behavior of the syslog facility. A syslog option is an element of a finite type (see Section 5.10) constructed by the `syslog-option` macro. The syslog facility works with sets of options which are represented as enum sets (see Section 5.11).

- `(syslog-option` *option-name*`)` → *option* syntax
- `(syslog-option?` *x*`)` → *boolean*
- `(make-syslog-options` *list*`)` → *options*
- `(syslog-options` *option-name* `...)` → *options* syntax
- `(syslog-options?` *x*`)` → *boolean*

`Syslog-option` constructs a log option from the name of an option. (The possible names are listed below.) `Syslog-option?` is a predicate for log options. Options are comparable using `eq?`. `Make-syslog-options` constructs a set of options from a list of options. `Syslog-options` is a macro which expands into an expression returning a set of options from names. `Syslog-options?` is a predicate for sets of options.

Here is a list of possible names of syslog options:

`console` If syslog cannot pass the message to syslogd it will attempt to write the message to the console.

`delay` Delay opening the connection to syslogd immediately until the first message is logged.

`no-delay` Open the connection to syslogd immediately. Normally the open is delayed until the first message is logged. Useful for programs that need to manage the order in which file descriptors are allocated.

NOTA BENE: The `delay` and `no-delay` options are included for completeness, but do not have the expected effect in the present Scheme interface: Because the Scheme interface has to multiplex multiple simultaneous connections to the syslog facility over a single one, open and close operations on that facility happen at unpredictable times.

`log-pid` Log the process id with each message: useful for identifying instantiations of daemons.

`no-wait` Do not wait for child processes.

Log facilities

A log facility identifies the originator of a log message from a finite set known to the system. Each originator is identified by a name:

- `(syslog-facility` *facility-name*`)` → *facility* syntax
- `(syslog-facility?` *x*`)` → *boolean*

`Syslog-facility` is macro that expands into an expression returning a facility for a given name. `Syslog-facility?` is a predicate for facilities. Facilities are comparable via `eq?`.

Here is a list of possible names of syslog facilities:

`authorization` The authorization system: login, su, getty, etc.

`cron` The cron daemon.

`daemon` System daemons, such as routed, that are not provided for explicitly by other facilities.

`kernel` Messages generated by the kernel.

`lpr` The line printer spooling system: lpr, lpc, lpd, etc.

`mail` The mail system.

`news` The network news system.

`user` Messages generated by random user processes.

`uucp` The uucp system.

`local0` `local1` `local2` `local3` `local4` `local5` `local6` `local7` Reserved for local use.

Log levels

A log level identifies the importance of a message from a fixed set of possible levels.

- `(syslog-level` *level-name*`)` → *level* syntax
- `(syslog-level?` *x*`)` → *boolean*

`Syslog-level` is macro that expands into an expression returning a facility for a given name. `Syslog-level?` is a predicate for facilities. Levels are comparable via `eq?`.

Here is a list of possible names of syslog levels:

`emergency` A panic condition. This is normally broadcast to all users.

`alert` A condition that should be corrected immediately, such as a corrupted system database.

`critical` Critical conditions, e.g., hard device errors.

`error` Errors.

`warning` Warning messages.

`notice` Conditions that are not error conditions, but should possibly be handled specially.

`info` Informational messages.

`debug` Messages that contain information normally of use only when debugging a program.

Log masks

A log masks can mask out log messages at a set of levels. A log mask is an enum set of log levels.

- `(make-syslog-mask` *list*`)` → *mask*
- `(syslog-mask` *level-name* `...)` → *mask* syntax
- `syslog-mask-all` mask
- `(syslog-mask-upto` *level*`)` → *mask*
- `(syslog-mask?` *x*`)` → *boolean*

`Make-syslog-mask` constructs a mask from a list of levels. `Syslog-mask` is a macro which constructs a mask from names of levels. `Syslog-mask-all` is a predefined log mask containing all levels. `Syslog-mask-upto` returns a mask consisting of all levels up to and including a certain level, starting with `emergency`.

Logging

Scheme 48 dynamically maintains implicit connections to the syslog facility specifying a current identifier, current options, a current facility and a current log mask. Every thread maintains it own implicit connection to syslog. Note that the connection is not implicitly preserved across a `spawn`.

- `(with-syslog-destination` *string options facility mask thunk*`)` → *value*

`With-syslog-destination` dynamically binds parameters of the implicit connection to the syslog facility and runs *thunk* within those parameter bindings, returning what *thunk* returns. Each of the parameters may be `#f` in which case the previous values will be used.

- `(syslog` *level message*`)`
- `(syslog` *level message string*`)`

- (`syslog` *level message string options*)
- (`syslog` *level message string options syslog-facility*)
- (`syslog` *level message channel*)

`Syslog` actually logs a message. Each of the parameters of the implicit connection (except for the log mask) can be explicitly specified as well for the current call to `syslog`, overriding the parameters of the channel. The parameters revert to their original values after the call.

The final form specifies the destination of the log message as a channel; see the next section.

Syslog channels

These procedures allow direct manipulation of syslog channels, the objects that represent connections to the syslog facility. Note that it is not necessary to explicitly open a syslog channel to do logging.

- (`open-syslog-channel`*string options facility mask*) $\rightarrow$ *channel*
- (`close-syslog-channel`*channel*)
- (`with-syslog-destination` *channel thunk*) $\rightarrow$ *value*

`Open-syslog-channel` and `close-syslog-channel` create and destroy a connection to the syslog facility, respectively. The specified form of calling `syslog` logs to the specified channel.

`With-syslog-channel` dynamically binds parameters of the implicit connection to the syslog facility to those specified in *channel* and runs *thunk* within those parameter bindings, returning what *thunk* returns.

9.11 Error codes

POSIX functions report the nature of an error via *system error numbers*—OS-specific integers that encode a variety of different error situations. At the Scheme level, error numbers are represented as *errnos*, objects that specify a name for the error sitation. Errnos are definted in the structures `posix-errnos` and `posix`.

Currently, the system reports such error situations by raising exceptions with condition type `&os-error`. The `&os-error` condition type has a field `code` that contains the system error number.

There are two varieties of errnos available, *named* and *anonymous*. A named errno is one for which we have a symbolic name, such as `fault` or `intr`. Anonymous errnos, for which we only have the current operating system's errno number, have no meaning in other operating systems. Named errnos preserve their meaning in image files. Not all named errnos are available from all OS's and there may be multiple names for a single OS errno number.

- (`errno` *errno-name*) $\rightarrow$ *errno* syntax
- (`name->errno` *symbol*) $\rightarrow$ *errno or* `#f`

- `(integer->errno` *integer*`)` → *errno*
- `(errno?` *x*`)` → *boolean*
- `(errno-name` *errno*`)` → *symbol or* `#f`
- `(errno-os-number` *errno*`)` → *integer*
- `(errno=?` *errno0 errno1*`)` → *boolean*

The syntax `errno` returns a (named) errno associated with *errno-name*. `Name->errno` returns a (named) errno or `#f` if the the errno *name* is not supported by the operating system. The errno returned by `integer->errno` is a named errno if *integer* corresponds to a named errno in the current operating system; otherwise it returns an anonymous errno. `Errno-name` returns a symbol if *errno* is named and `#f` if it is anonymous. `Errno=?` returns `#t` if *errno0* and *errno1* have the same operating system number and `#f` if they do not.

9.11.1 POSIX errnos

The following lists the names of the POSIX errnos.

`toobig`	Argument list too long.
`acces`	Permission denied.
`addrinuse`	Address in use.
`addrnotavail`	Address not available.
`afnosupport`	Address family not supported.
`again`	Resource unavailable, try again.
`already`	Connection already in progress.
`badf`	Bad file descriptor.
`badmsg`	Bad message.
`busy`	Device or resource busy.
`canceled`	Operation canceled.
`child`	No child processes.
`connaborted`	Connection aborted.
`connrefused`	Connection refused.
`connreset`	Connection reset.
`deadlk`	Resource deadlock would occur.
`destaddrreq`	Destination address required.
`dom`	Mathematics argument out of domain of function.
`dquot`	Reserved.
`exist`	File exists.
`fault`	Bad address.
`fbig`	File too large.
`hostunreach`	Host is unreachable.
`idrm`	Identifier removed.
`ilseq`	Illegal byte sequence.
`inprogress`	Operation in progress.
`intr`	Interrupted function.

`inval`	Invalid argument.
`io`	I/O error.
`isconn`	Socket is connected.
`isdir`	Is a directory.
`loop`	Too many levels of symbolic links.
`mfile`	Too many open files.
`mlink`	Too many links.
`msgsize`	Message too large.
`multihop`	Reserved.
`nametoolong`	Filename too long.
`netdown`	Network is down.
`netreset`	Connection aborted by network.
`netunreach`	Network unreachable.
`nfile`	Too many files open in system.
`nobufs`	No buffer space available.
`nodata`	No message is available on the STREAM head read queue.
`nodev`	No such device.
`noent`	No such file or directory.
`noexec`	Executable file format error.
`nolck`	No locks available.
`nolink`	Reserved.
`nomem`	Not enough space.
`nomsg`	No message of the desired type.
`noprotoopt`	Protocol not available.
`nospc`	No space left on device.
`nosr`	No STREAM resources.
`nostr`	Not a STREAM.
`nosys`	Function not supported.
`notconn`	The socket is not connected.
`notdir`	Not a directory.
`notempty`	Directory not empty.
`notsock`	Not a socket.
`notsup`	Not supported.
`notty`	Inappropriate I/O control operation.
`nxio`	No such device or address.
`opnotsupp`	Operation not supported on socket.
`overflow`	Value too large to be stored in data type.
`perm`	Operation not permitted.
`pipe`	Broken pipe.
`proto`	Protocol error.
`protonosupport`	Protocol not supported.
`prototype`	Protocol wrong type for socket.
`range`	Result too large.
`rofs`	Read-only file system.
`spipe`	Invalid seek.

`srch`	No such process.
`stale`	Reserved.
`time`	Stream ioctl() timeout.
`timedout`	Connection timed out.
`txtbsy`	Text file busy.
`wouldblock`	Operation would block.
`xdev`	Cross-device link.

9.12 C to Scheme correspondence

The following table lists the Scheme procedures that correspond to particular C procedures. Not all of the Scheme procedures listed are part of the POSIX interface.

C procedure	Scheme procedure(s)
`access`	`accessible?`
`asctime`	`date->string`
`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|`ϵ`]`	`exec, exec-with-environment, exec-file, exec-file-with-environment, exec-with-alias`
`_exit`	`exit`
`fcntl`	`io-flags, set-io-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-groups`
`getlogin`	`get-login-name`
`getpid`	`get-process-id`
`getppid`	`get-parent-process-id`

C procedure	Scheme procedure(s)
`getuid`	`get-user-id`
`gmtime`	`time->local-date`
`isatty`	`port-is-a-terminal?`
`link`	`link`
`localtime`	`time->utc-date`
`lstat`	`get-file/link-info`
`mkdir`	`make-directory`
`mkfifo`	`make-fifo`
`mktime`	`date->time`
`open`	`open-file`
`opendir`	`open-directory-stream`
`pipe`	`open-pipe`
`read`	`read-char, read-block`
`readdir`	`read-directory-stream`
`readlink`	`read-symbolic-link`
`rename`	`rename`
`rmdir`	`remove-directory`
`setgid`	`set-group-id!`
`setegid`	`set-effective-group-id!`
`setuid`	`set-user-id!`
`seteuid`	`set-effective-user-id!`
`stat`	`get-file-info`
`strftime`	`format-date`
`symlink`	`create-symbolic-link`
`syslog`	`syslog`
`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`

Bibliography

[1] Information technology – Portable Operating System Interface (POSIX). ISO/IEC 9945-1 ANSI/IEEE Std 1003.1. 2nd Ed., 1996.

[2] William Clinger and Jonathan Rees. Macros that work. *Principles of Programming Languages*, January 1991.

[3] William Clinger and Jonathan Rees (editors). Revised^4 report on the algorithmic language Scheme. *LISP Pointers* IV(3):1–55, July-September 1991.

[4] Pavel Curtis and James Rauen. A module system for Scheme. *ACM Conference on Lisp and Functional Programming,* pages 13–19, 1990.

[5] Richard Kelsey and Jonathan Rees. A Tractable Scheme Implementation. *Lisp and Symbolic Computation* 7:315–335 1994.

[6] Richard Kelsey, Will Clinger, Jonathan Rees (editors). Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation,* Vol. 11, No. 1, September, 1998. and *ACM SIGPLAN Notices*, Vol. 33, No. 9, October, 1998.

[7] David MacQueen. Modules for Standard ML. *ACM Conference on Lisp and Functional Programming,* 1984.

[8] Jonathan Rees and Bruce Donald. Program mobile robots in Scheme. *International Conference on Robotics and Automation,* IEEE, 1992.

[9] John H. Reppy. Concurrent Programming in ML. Cambridge University Press, 1999.

[10] Mark A. Sheldon and David K. Gifford. Static dependent types for first-class modules. *ACM Conference on Lisp and Functional Programming,* pages 20–29, 1990.

[11] Olin Shivers, Brian D. Carlstrom, Martin Gasbichler and Mike Sperber. Scsh Reference Manual, scsh release 0.6.6. Available at URL `http://www.scsh.net/`.

[12] Olin Shivers. A universal scripting framework, or Lambda: the ultimate "little language". *Concurrency and Parallelism, Programming, Networking, and Security,* pages 254–265, Springer 1996. Joxan Jaffar and Roland H. C. Yap, editors.

Index