The Incomplete

Scheme 48 Reference Manual

for release 0.55

Richard Kelsey and Jonathan Rees
with a chapter by Mike Sperber

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.) Support for numbers is weak: bignums are slow and floating point is almost nonexistent (see description of floatnums, below).

Scheme 48 is under continual development. Please report bugs, especially in the VM, especially core dumps, to scheme-48-bugs@zurich.ai.mit.edu. 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 mail to scheme-48-request@zurich.ai.mit.edu 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 strings to the procedure specified in the `,build` command as for example:

```
> (define (f a) (for-each display a) (newline) 0)
> ,build f foo.image
> ,exit
% scheme48vm -i foo.image -a mumble "foo x"
mumblefoo x
%
```

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 6.5 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/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 5.7 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 3.7) 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 occured. 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 `auto-levels` switch described below can be used to disable the automatic pushing of new levels.

The command processor's evaluation package and the value of the `inspect-focus-value` switch 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 switches 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 `auto-levels` switch 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. Also, it works in any package, unlike `(load "`*filename*`")`, which will work only work 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` switch can be used 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 when writing command programs, which are described in section 3.8 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*
`,expand-all` *form*
Show macro expansion of *form*, if any. `,expand` performs a single macro expansion while `,expand-all` fully expands all macros in *form*.

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

3.6 Switches

There are a number of binary switches that control the behavior of the command processor. They can be set using the `,set` and `,unset` commands.

`,set` *switch* `[on | off | ?]`
This sets the value of mode-switch *switch*. The second argument defaults to `on`. If the second argument is `?` the value of *switch* is is displayed and not changed. Doing `,set ?` will display a list of the switches and their current values.

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

The switches are as follows:

`batch`
: 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.

`auto-levels`
: 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 errors
- confusion among some newcomers

With `auto-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.

`inspect-focus-value`
: Enable or disable 'inspection' mode, which is used for inspecting data structures and continuations. Inspection mode is described in section 3.7.

`break-on-warnings`
: 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`
: 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`
: 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`
: 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.

3.7 Inspection mode

There is a data inspector available via the `,inspect` and `,debug` commands or by setting the `inspect-focus-value` switch. 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, by unsetting the `inspect-focus-value` switch, or by going to a command level where the `inspect-focus-value` is not set. 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)
> ,0
'(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.

3.8 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.

3.9 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*
: 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 strings. 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).

  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 3.10 for more information.

3.10 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.

3.11 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.

3.12 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 *form*s came. *Filename* is then used by the command processor to determine the package in which the *form*s 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 *accessing* 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 4.2.

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.

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/)
```

```
⟨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.

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

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.

$$
\begin{array}{rl}
\langle\text{definition}\rangle \longrightarrow & \texttt{(define-module (}\langle\text{name}\rangle\ \texttt{(}\langle\text{name}\rangle\ \langle\text{interface}\rangle\texttt{)}^{*}\texttt{)} \\
& \quad \langle\text{definition}\rangle^{*} \\
& \quad \langle\text{name}\rangle\texttt{)} \\
| & \texttt{(define}\ \langle\text{name}\rangle\ \texttt{(}\langle\text{name}\rangle\ \langle\text{name}\rangle^{*}\texttt{))}
\end{array}
$$

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` *struct-name**
: The `,open` command opens a new structure in the current package, as if the package's definition's `open` clause had listed *struct-name*.

`,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`
  The `,new-package` command creates a new package, in which only the standard Scheme bindings are visible, and moves the command processor to it.

`,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/more-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 defined must also have a way to create "reflective towers" (a misnomer; a better name would be "syntactic towers"). A new reflective 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-reflective-tower` creates an interface that exports a configuration package's tower. The following example uses `export-reflective-tower` and the `,structure` command to obtain a tower maker and create a new configuration environment.

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

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[9]) 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 ASCII character encoding

These are in the structure `ascii`.

- `(char->ascii` *char*`)` $\rightarrow$ *integer*
- `(ascii->char` *integer*`)` $\rightarrow$ *char*

These are identical to `char->integer` and `integer->char` except that they use the ASCII encodingASCII encoding (appendix A).

- `ascii-limit` integer
- `ascii-whitespaces` list of integers

`Ascii-limit` is one more than the largest value that `char->ascii` may return. `Ascii-whitespaces` is a list of the ASCII values of whitespace characters (space, horizontal tab, line feed (= newline), vertical tab, form feed, and carriage return).

5.2 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.

5.3 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 *i* ...) $\rightarrow$ *byte-vector*
- (byte-vector-length *byte-vector*) $\rightarrow$ *integer*
- (byte-vector-ref *byte-vector k*) $\rightarrow$ *integer*
- (byte-vector-set! *byte-vector k i*)

5.4 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*) $\rightarrow$ *boolean*
- (make-cell *value*) $\rightarrow$ *cell*
- (cell-ref *cell*) $\rightarrow$ *value*
- (cell-set! *cell value*)

5.5 Queues

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

- (make-queue) $\rightarrow$ *queue*
- (queue? *value*) $\rightarrow$ *boolean*
- (queue-empty? *queue*) $\rightarrow$ *boolean*
- (enqueue! *queue value*)
- (dequeue! *queue*) $\rightarrow$ *value*

`Make-queue` creates an empty queue, `queue?` is a predicate for identifying queues, `queue-empty?` tells you if a queue is empty, `enqueue!` and `dequeue!` add and remove values.

- (queue-length *queue*) $\rightarrow$ *integer*
- (queue->list *queue*) $\rightarrow$ *values*
- (list->queue *values*) $\rightarrow$ *queue*
- (delete-from-queue! *queue value*) $\rightarrow$ *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. `List->queue` returns a queue containing *values*, preserving their order. `Delete-from-queue` removes the first instance of *value* from `queue`, using `eq?` for comparisons. `Delete-from-queue` returns `#t` if *value* is found and `#f` if it is not.

5.6 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*$_0$ `...)` $\rightarrow$ *array*
- `(array` *dimensions* *element*$_0$ `...)` $\rightarrow$ *array*
- `(copy-array` *array*`)` $\rightarrow$ *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 non-negative 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*`)` $\rightarrow$ *boolean*

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

- `(array-ref` *array* *index*$_0$ `...)` $\rightarrow$ *value*
- `(array-set!` *array* *value* *index*$_0$ `...)`
- `(array->vector` *array*`)` $\rightarrow$ *vector*
- `(array-dimensions` *array*`)` $\rightarrow$ *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-dimensions` returns the dimensions of the array as a list.

- `(make-shared-array` *array* *linear-map* *dimension*$_0$ `...)` $\rightarrow$ *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 ((dimensions (array-dimensions array)))
    (make-shared-array array
                       (lambda (x y)
                         (list y x))
                       (cadr dimensions)
                       (car dimensions))))

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

5.7 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.

- (`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 6.9.3) can be used to control how records are stored in heap images.

5.7.1 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*`)` $\rightarrow$ *record*
- `(record` *value* ...`)` $\rightarrow$ *record-vector*
- `(record?` *value*`)` $\rightarrow$ *boolean*
- `(record-length` *record*`)` $\rightarrow$ *integer*
- `(record-type` *record*`)` $\rightarrow$ *value*
- `(record-ref` *record i*`)` $\rightarrow$ *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)`.

5.7.2 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 approriate 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-discloser` (re-exported by `external-calls` (section 6.9.3)).

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 (sematically) 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.

5.8 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
```

5.9 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?` 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`.

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

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

5.10 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*`)` $\rightarrow$ *output-port*
- `(current-row` *port*`)` $\rightarrow$ *integer or* `#f`
- `(current-column` *port*`)` $\rightarrow$ *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 (open-output-port "/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)
```

5.11 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*`)` $\rightarrow$ *fluid*
- `(fluid` *fluid*`)` $\rightarrow$ *value*
- `(let-fluid` *fluid value thunk*`)` $\rightarrow$ *value(s)*
- `(let-fluids` *fluid*$_0$ *value*$_0$ *fluid*$_1$ *value*$_1$ *... thunk*`)` $\rightarrow$ *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)
```

5.12 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))))
```

5.13 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`.

5.13.1 `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 defines):

```
(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).

5.13.2 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])
```

5.13.3 Sequence types

The predefined sequence types are:

- `(list*` *elt-var list*`)` 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

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 `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 `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))))
```

5.13.4 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
- `(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%`.

5.13.5 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)))))
```

5.13.6 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 `synchronized?` 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 (synchronized? state-vars step)
    ((list* synchronized? (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 ...))))
```

5.13.7 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.

5.14 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 `regexp`.

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.

5.14.1 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*.

5.14.2 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.

5.14.3 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 occurences 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`.

5.14.4 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"`.

5.14.5 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*`)` $\rightarrow$ *reg-exp*
- `(no-submatches` *reg-exp*`)` $\rightarrow$ *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*`)` $\rightarrow$ *boolean*
- `(exact-match?` *reg-exp string*`)` $\rightarrow$ *boolean*
- `(match` *reg-exp string*`)` $\rightarrow$ *match or* `#f`
- `(match-start` *match*`)` $\rightarrow$ *index*
- `(match-end` *match*`)` $\rightarrow$ *index*
- `(match-submatches` *match*`)` $\rightarrow$ *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}))
```

5.15 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 the SRFI web site at `http://srfi.schemers.org`.

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

- SRFI 1 – List Library
- SRFI 2 – `and-let*`
- 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

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

The 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

Mixing Scheme 48 and C

Mike Sperber and Richard Kelsey

This chapter describes an 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.

6.1 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.

6.1.1 Scheme structures

The structure `external-calls` has most of the Scheme functions described here. The others are in `dynamic-externals`, which has the functions for dynamic loading and name lookup from Section 6.5, and `shared-bindings`, which has the additional shared-binding functions described in Section 6.2.3.

6.1.2 C naming conventions

The names of all of Scheme 48's visible C bindings begin with '`s48_`' (for procedures and variables) or '`S48_`' (for macros). Whenever a C name is derived from a Scheme identifier, we replace '`-`' with '`_`' and convert letters to lowercase for procedures and uppercase for macros. A final '`?`' converted to '`_p`' ('`_P`' in C macro names). A final '`!`' is dropped. Thus the C macro for Scheme's `pair?` is `S48_PAIR_P` and the one for `set-car!` is `S48_SET_CAR`. 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`. The C type for Scheme values is defined there to be `s48_value`.

6.1.3 Garbage collection

Scheme 48 uses a copying garbage collector. 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. The garbage collector may run whenever an object is allocated in the heap. C variables whose values are Scheme 48 objects and which are live across heap allocation calls need to be registered with the garbage collectorgarbage collector. See section 6.9 for more information.

6.2 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.

6.2.1 Exporting Scheme values to C

- `(define-exported-binding` *name value*`)` $\rightarrow$ *shared-binding*

- `s48_value s48_get_imported_binding(char *name)`
- `s48_value S48_SHARED_BINDING_REF(s48_value shared_binding)`

`Define-exported-binding` makes *value* available to C code under as *name* which must be a *string*, creating a new shared binding if necessary. The C function `s48_get_imported_binding` returns the shared binding defined for `name`, again creating it if necessary. The C macro `S48_SHARED_BINDING_REF` dereferences a shared binding, returning its current value.

6.2.2 Exporting C values to Scheme

- `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 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 6.9.

- `(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"))`

6.2.3 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*`)` → *boolean*
- `(shared-binding-name` *shared-binding*`)` → *string*
- `(shared-binding-is-import?` *shared-binding*`)` → *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(x)`
- `int       S48_SHARED_BINDING_IS_IMPORT_P(s48_value s_b)`
- `s48_value S48_SHARED_BINDING_NAME(s48_value s_b)`
- `void      S48_SHARED_BINDING_SET(s48_value s_b, s48_value v)`

6.3 Calling C functions from Scheme

There are three different ways to call C functions from Scheme, depending on how the C function was obtained.

- `(call-imported-binding` *binding* arg_0 `...)` $\rightarrow$ *value*
- `(call-external` *external* arg_0 `...)` $\rightarrow$ *value*
- `(call-external-value` *value name* arg_0 `...)` $\rightarrow$ *value*

Each of these applies its first argument, a C function, to the rest of the arguments. For `call-imported-binding` the function argument must be an imported binding. For `call-external` the function argument must be an external bound in the current process (see Section 6.5). For `call-external-value` *value* must be a byte vector whose contents is a pointer to a C function and *name* should be a string naming the function. The *name* argument is used only for printing error messages.

For all of these, the C function is passed the arg_i values 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` *name* `(`*formal* `...))` syntax
- `(import-lambda-definition` *name* `(`*formal* `...)` *c-name*`)` syntax

These macros simplify importing functions from C. 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 ...)
    (external-apply temp formal ...)))
```

If *c-name* is not supplied, it is derived from *name* by converting all letters to lowercase and replacing '-' with '_'.

6.4 Adding external modules to the `Makefile`

Getting access to C bindings from Scheme requires that the C code be compiled and linked in with the Scheme 48 virtual machine and that the relevant shared bindings be created. The Scheme 48 makefile has rules for compiling and linking external code and for specifying initialization functions that should be called on startup. There are three `Makefile` variables that control which external modules are included in the executable for the virtual machine (`scheme48vm`). `EXTERNAL_OBJECTS` lists the object files to be included in `scheme48vm`, `EXTERNAL_FLAGS` is a list of `ld` flags to be used when creating `scheme48vm`, and `EXTERNAL_INITIALIZERS` is a list of C procedures to be called on startup. The procedures listed in `EXTERNAL_INITIALIZERS` should take no arguments and have a return type of `void`. After changing the definitions of any of these variables you should do `make scheme48vm` to rebuild the virtual machine.

6.5 Dynamic loading

External code can be loaded into a running Scheme 48 process and C object-file bindings can be dereferenced at runtime and their values called (although not all versions of Unix support all of this). The required Scheme functions are in the structure `dynamic-externals`.

- `(dynamic-load` *string*`)`

`Dynamic-load` loads the named file into the current process, raising an exception if the file cannot be found or if dynamic loading is not supported by the operating system. The file must have been compiled and linked appropriately. 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
```

- `(get-external` *string*`)` → *external*
- `(external?` *x*`)` → *boolean*
- `(external-name` *external*`)` → *string*
- `(external-value` *external*`)` → *byte-vector*

These functions give access to values bound in the current process, and are used for retrieving values from dynamically-loaded files. `Get-external` returns an *external* object that contains the value of *name*, raising an exception if there is no such value in the current process. `External?` is the predicate for externals, and `external-name` and `external-value` return the name and value of an external. The value is returned as byte vector of length four (on 32-bit architectures). The value is that which was extant when `get-external` was called. The following two functions can be used to update the values of externals.

- `(lookup-external` *external*`)` → *boolean*
- `(lookup-all-externals)` → *boolean*

`Lookup-external` updates the value of *external* by looking up its name in the current process, returning `#t` if the name is bound and `#f` if it is not. `Lookup-all-externals` calls `lookup-external` on all extant externals, returning `#f` any are unbound.

- `(call-external` *external* arg_0 `...)` → *value*

An external whose value is a C procedure can be called using `call-external`. See Section 6.3 for more information.

In some versions of Unix retrieving a value from the current process may require a non-trivial amount of computation. We recommend that a dynamically-loaded file contain a single initialization procedure that creates shared bindings for the values exported by the file.

6.6 Compatibility

Scheme 48's old `external-call` function is still available in the structure `externals`, which now also includes `external-name` and `external-value`. The old `scheme48.h` file has been renamed `old-scheme48.h`.

6.7 Accessing Scheme data from C

The C header file `scheme48.h` provides access to Scheme 48 data structures. The type `s48_value` is used for Scheme values. When the type of a value is known, such as the integer returned by `vector-length` or the boolean returned by `pair?`, the corresponding C procedure returns a C value of the appropriate type, and not a `s48_value`. Predicates return `1` for true and `0` for false.

6.7.1 Constants

The following macros denote Scheme constants:

- `S48_FALSE` is `#f`.
- `S48_TRUE` is `#t`.
- `S48_NULL` is the empty list.
- `S48_UNSPECIFIC` 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` 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`).

6.7.2 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(s48_value)`
- `unsigned char s48_extract_char(s48_value)`
- `char *        s48_extract_string(s48_value)`
- `char *        s48_extract_byte_vector(s48_value)`
- `long          s48_extract_integer(s48_value)`
- `double        s48_extract_double(s48_value)`
- `s48_value S48_ENTER_BOOLEAN(int)`
- `s48_value s48_enter_char(unsigned char)`
- `s48_value s48_enter_string(char *)` (may GC)
- `s48_value s48_enter_byte_vector(char *, long)` (may GC)
- `s48_value s48_enter_integer(long)` (may GC)
- `s48_value s48_enter_double(double)` (may GC)

`S48_EXTRACT_BOOLEAN` is false if its argument is `#f` and true otherwise. `S48_ENTER_BOOLEAN` is `#f` if its argument is zero and `#t` otherwise.

`s48_extract_string` and `s48_extract_byte_vector` return pointers to the actual storage used by the string or byte vector. These pointers are valid only until the next garbage collection; see Section 6.9.

The second argument to `s48_enter_byte_vector` is the length of byte vector.

`s48_enter_integer()` 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 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` is a macro that true if its argument is a fixnum and false otherwise.

- `int       S48_TRUE_P(s48_value)`
- `int       S48_FALSE_P(s48_value)`

`S48_TRUE_P` is true if its argument is `S48_TRUE` and `S48_FALSE_P` is true if its argument is `S48_FALSE`.

- `int       S48_FIXNUM_P(s48_value)`
- `long      s48_extract_fixnum(s48_value)`
- `s48_value s48_enter_fixnum(long)`
- `long      S48_MAX_FIXNUM_VALUE`
- `long      S48_MIN_FIXNUM_VALUE`

An error is signalled if `s48_extract_fixnum`'s argument is not a fixnum or if the argument to `s48_enter_fixnum` is less than `S48_MIN_FIXNUM_VALUE` or greater than `S48_MAX_FIXNUM_VALUE` (-2^{29} and $2^{29}-1$ in the current system).

6.7.3 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(s48_value, s48_VALUE)`
- `int       S48_CHAR_P(s48_value)`

- `int       S48_PAIR_P(s48_value)`
- `s48_value S48_CAR(s48_value)`
- `s48_value S48_CDR(s48_value)`
- `void      S48_SET_CAR(s48_value, s48_value)`
- `void      S48_SET_CDR(s48_value, s48_value)`
- `s48_value s48_cons(s48_value, s48_value)` (may GC)
- `long      s48_length(s48_value)`

- `int       S48_VECTOR_P(s48_value)`
- `long      S48_VECTOR_LENGTH(s48_value)`
- `s48_value S48_VECTOR_REF(s48_value, long)`
- `void      S48_VECTOR_SET(s48_value, long, s48_value)`
- `s48_value s48_make_vector(long, s48_value)` (may GC)

- `int       S48_STRING_P(s48_value)`

- `long      S48_STRING_LENGTH(s48_value)`
- `char      S48_STRING_REF(s48_value, long)`
- `void      S48_STRING_SET(s48_value, long, char)`
- `s48_value s48_make_string(long, char)` (may GC)

- `int       S48_SYMBOL_P(s48_value)`
- `s48_value s48_SYMBOL_TO_STRING(s48_value)`

- `int       S48_BYTE_VECTOR_P(s48_value)`
- `long      S48_BYTE_VECTOR_LENGTH(s48_value)`
- `char      S48_BYTE_VECTOR_REF(s48_value, long)`
- `void      S48_BYTE_VECTOR_SET(s48_value, long, int)`
- `s48_value s48_make_byte_vector(long, int)` (may GC)

6.8 Calling Scheme functions from C

External code that has been called from Scheme can call back to Scheme procedures using the following function.

- `scheme_value s48_call_scheme(s48_value p, long nargs, ...)`

This calls the Scheme procedure `p` on `nargs` arguments, which are passed as additional arguments to `s48_call_scheme`. There may be at most twelve arguments. The value returned by the Scheme procedure is returned 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.

6.9 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. Variables bound to values in the Scheme 48 heap need to be registered with the garbage collector so that the value will be retained and so that the variables will be updated if the garbage collector moves the object. The garbage collector has no facility for updating pointers to the interiors of objects, so such pointers, for example the ones returned by `EXTRACT_STRING`, will likely become invalid when a garbage collection occurs.

6.9.1 Registering objects with the GC

A set of macros are used to manage the registration of local variables with the garbage collector.

- `S48_DECLARE_GC_PROTECT(`n`)`
- `void S48_GC_PROTECT_`n`(s48_value`$_1$`, ..., s48_value`$_n$`)`
- `void S48_GC_UNPROTECT()`

`S48_DECLARE_GC_PROTECT(`n`)`, where $1 \leq n \leq 9$, allocates storage for registering n variables. At most one use of `S48_DECLARE_GC_PROTECT` may occur in a block. `S48_GC_PROTECT_`n`(`v_1`, ..., `v_n`)` registers the n variables (l-values) with the garbage collector. It must be within scope of a `S48_DECLARE_GC_PROTECT(`n`)` and be before any code which can cause a GC. `S48_GC_UNPROTECT` removes the block's protected variables from the garbage collector's list. It must be called at the end of the block after any code which may cause a garbage collection. Omitting any of the three may cause serious and hard-to-debug problems. Notably, the garbage collector may relocate an object and invalidate `s48_value` variables which are not protected.

A `gc-protection-mismatch` exception is raised if, when a C procedure returns to Scheme, the calls to `S48_GC_PROTECT()` have not been matched by an equal number of calls to `S48_GC_UNPROTECT()`.

Global variables may also be registered with the garbage collector.

- `void S48_GC_PROTECT_GLOBAL(`*value*`)`

`S48_GC_PROTECT_GLOBAL` permanently registers the variable *value* (an l-value) with the garbage collector. There is no way to unregister the variable.

6.9.2 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_value S48_MAKE_VALUE(type)` (may GC)
- `type      S48_EXTRACT_VALUE(s48_value, type)`
- `type *    S48_EXTRACT_VALUE_POINTER(s48_value, type)`
- `void      S48_SET_VALUE(s48_value, type, value)`

`S48_MAKE_VALUE` makes a byte vector large enough to hold an object whose type is *type*. `S48_EXTRACT_VALUE` returns the contents of a byte vector cast to *type*, and `S48_EXTRACT_VALUE_POINTER` returns a pointer to the contents of the byte vector.

The value returned by S48_EXTRACT_VALUE_POINTER is valid only until the next garbage collection.

S48_SET_VALUE stores value into the byte vector.

6.9.3 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.

6.10 Using Scheme records in C code

External modules can create records and access their slots positionally.

- s48_value S48_MAKE_RECORD(s48_value) (may GC)
- int S48_RECORD_P(s48_value)
- s48_value S48_RECORD_TYPE(s48_value)
- s48_value S48_RECORD_REF(s48_value, long)
- void S48_RECORD_SET(s48_value, long, s48_value)

The argument to S48_MAKE_RECORD 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 scheme_value
  thing_record_type_binding = SCHFALSE;

void initialize_things(void)
{
  S48_GC_PROTECT_GLOBAL(thing_record_type_binding);
  thing_record_type_binding =
     s48_get_imported_binding("thing-record-type");
}

scheme_value make_thing(scheme_value a, scheme_value b)
{
  s48_value thing;
  s48_DECLARE_GC_PROTECT(2);

  S48_GC_PROTECT_2(a, b);

  thing = s48_make_record(thing_record_type_binding);
  S48_RECORD_SET(thing, 0, a);
  S48_RECORD_SET(thing, 1, b);

  S48_GC_UNPROTECT();

  return thing;
}
```

Note that the variables `a` and `b` must be protected against the possibility of a garbage collection occuring during the call to `s48_make_record()`.

6.11 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.

- `s48_raise_scheme_exception(int type, int nargs, ...)`

`s48_raise_scheme_exception` is the base procedure for raising exceptions. `type` is the type of exception, and should be one of the `S48_EXCEPTION_...` constants defined in `scheme48arch.h`. `nargs` is the number of additional values to be included in the exception; these follow the `nargs` argument and should all have type `s48_value`. `s48_raise_scheme_exception` never returns.

The following procedures are available for raising particular types of exceptions. Like `s48_raise_scheme_exception` these never return.

- `s48_raise_argument_type_error(scheme_value)`
- `s48_raise_argument_number_error(int nargs, int min, int max)`
- `s48_raise_range_error(long value, long min, long max)`
- `s48_raise_closed_channel_error()`
- `s48_raise_os_error(int errno)`
- `s48_raise_out_of_memory_error()`

An argument type error indicates that the given value is of the wrong type. An argument number error is raised when the number of arguments, `nargs`, should be, but isn't, between `min` and `max`, inclusive. Similarly, a range error indicates that `value` is not between between `min` and `max`, inclusive.

The following macros raise argument type errors if their argument does not have the required type. `S48_CHECK_BOOLEAN` raises an error if its argument is neither `#t` or `#f`.

- `void S48_CHECK_BOOLEAN(s48_value)`
- `void S48_CHECK_SYMBOL(s48_value)`
- `void S48_CHECK_PAIR(s48_value)`
- `void S48_CHECK_STRING(s48_value)`
- `void S48_CHECK_INTEGER(s48_value)`
- `void S48_CHECK_CHANNEL(s48_value)`
- `void S48_CHECK_BYTE_VECTOR(s48_value)`
- `void S48_CHECK_RECORD(s48_value)`
- `void S48_CHECK_SHARED_BINDING(s48_value)`

6.12 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.

- `char      S48_UNSAFE_EXTRACT_CHAR(s48_value)`
- `char *    S48_UNSAFE_EXTRACT_STRING(s48_value)`
- `long      S48_UNSAFE_EXTRACT_INTEGER(s48_value)`
- `long      S48_UNSAFE_EXTRACT_DOUBLE(s48_value)`

- `long      S48_UNSAFE_EXTRACT_FIXNUM(s48_value)`
- `s48_value S48_UNSAFE_ENTER_FIXNUM(long)`

- `s48_value S48_UNSAFE_CAR(s48_value)`
- `s48_value S48_UNSAFE_CDR(s48_value)`
- `void      S48_UNSAFE_SET_CAR(s48_value, s48_value)`
- `void      S48_UNSAFE_SET_CDR(s48_value, s48_value)`

- `long      S48_UNSAFE_VECTOR_LENGTH(s48_value)`
- `s48_value S48_UNSAFE_VECTOR_REF(s48_value, long)`
- `void      S48_UNSAFE_VECTOR_SET(s48_value, long, s48_value)`

- `long      S48_UNSAFE_STRING_LENGTH(s48_value)`

- char   S48_UNSAFE_STRING_REF(s48_value, long)
- void   S48_UNSAFE_STRING_SET(s48_value, long, char)

- s48_value S48_UNSAFE_SYMBOL_TO_STRING(s48_value)

- long   S48_UNSAFE_BYTE_VECTOR_LENGTH(s48_value)
- char   S48_UNSAFE_BYTE_VECTOR_REF(s48_value, long)
- void   S48_UNSAFE_BYTE_VECTOR_SET(s48_value, long, int)

- s48_value S48_UNSAFE_SHARED_BINDING_REF(s48_value s_b)
- int   S48_UNSAFE_SHARED_BINDING_P(x)
- int   S48_UNSAFE_SHARED_BINDING_IS_IMPORT_P(s48_value s_b)
- s48_value S48_UNSAFE_SHARED_BINDING_NAME(s48_value s_b)
- void   S48_UNSAFE_SHARED_BINDING_SET(s48_value s_b, s48_value value)

- s48_value S48_UNSAFE_RECORD_TYPE(s48_value)
- s48_value S48_UNSAFE_RECORD_REF(s48_value, long)
- void   S48_UNSAFE_RECORD_SET(s48_value, long, s48_value)

- type   S48_UNSAFE_EXTRACT_VALUE(s48_value, type)
- type *   S48_UNSAFE_EXTRACT_VALUE_POINTER(s48_value, type)
- void   S48_UNSAFE_SET_VALUE(s48_value, type, value)

Chapter 7

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 7.10 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`	all of the above

Scheme 48's POSIX interface differs from Scsh's [10, 11] 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.

7.1 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`.

7.1.1 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.

7.1.2 `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 strings. *Env* should be a list of strings 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 strings 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 strings; unlike with the other four procedures, *name* is not added to this list (hence `-with-alias`).

7.2 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.

- `(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*

`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.

7.2.1 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

7.2.2 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

7.2.3 Sending signals

- `(signal-process` *process-id signal*`)`

Send *signal* to the process corresponding to *process-id*.

7.2.4 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*`)` → *signal-queue*
- `(signal-queue?` *x*`)` → *boolean*
- `(signal-queue-monitored-signals` *signal-queue*`)` → *list of signals*
- `(dequeue-signal!` *signal-queue*`)` → *signal*
- `(maybe-dequeue-signal!` *queue-queue*`)` → *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.

7.3 Process environment

These are in structures `posix-process-data` and `posix`.

7.3.1 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 7.1.1 for operations on process ids.

- `(get-user-id)` → *user-id*
- `(get-effective-user-id)` → *user-id*
- `(set-user-id!` *user-id*`)`

- `(get-group-id)` → *group-id*
- `(get-effective-group-id)` → *group-id*
- `(set-group-id!` *group-id*`)`

Every process has both the original and effective user id and group id. The effective values may be set, but not the original ones.

- `(get-groups)` → *group-ids*
- `(get-login-name)` → *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.

7.3.2 Environment variables

- `(lookup-environment-variable` *string*`)` → *string or* `#f`
- `(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. `Environment-alist` returns the entire environment as a list of `(`*name-string* `.` *value-string*`)` pairs.

7.4 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` *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*`)` → *string*
- `(user-info-id` *user-info*`)` → *user-id*
- `(user-info-group` *user-info*`)` → *group-id*
- `(user-info-home-directory` *user-info*`)` → *string*
- `(user-info-shell` *user-info*`)` → *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` *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*`)` → *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.

7.5 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*

7.6 Files and directories

These procedures are in structures `posix-files` and `posix`.

7.6.1 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 strings*

This is the obvious utility; it returns a list of the names in directory *name*.

7.6.2 Working directory

- `(working-directory)` → *string*
- `(set-working-directory!` *string*`)`

These return and set the working directory.

7.6.3 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 string *path*. 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*

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. 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.

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*)

`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` *name file-mode*)
- (`make-fifo` *file-mode*)

These two procedures make new directories and fifo files.

- (`unlink` *path*)
- (`remove-directory` *path*)
- (`rename` *old-path new-path*)

`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* 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`.

7.6.4 File information

- (`get-file-info` *name*) → *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 named file. `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*) → *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*`)` → *integer*
- `(file-info-inode` *file-info*`)` → *integer*

The device and inode numbers uniquely determine a file.

- `(file-info-link-count` *file-info*`)` → *integer*
- `(file-info-size` *file-info*`)` → *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*`)` → *user-id*
- `(file-info-group` *file-info*`)` → *group-id*
- `(file-info-mode` *file-info*`)` → *file-mode*

These return the owner, group, and access mode of a file.

- `(file-info-last-access` *file-info*`)` → *time*
- `(file-info-last-modification` *file-info*`)` → *time*
- `(file-info-last-info-change` *file-info*`)` → *time*

These return the time the file was last read, modified, or had its status modified

7.6.5 File modes

A file mode is a boxed integer representing a file protection mask.

- `(file-mode permission-name ...)` → *file-mode* syntax
- `(file-mode?` *x*`)` → *boolean*
- `(file-mode+` *file-mode* ...`)` → *file-mode*
- `(file-mode-` *file-mode0 file-mode1*`)` → *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*`)` → *boolean*
- `(file-mode<=?` *file-mode0 file-mode1*`)` → *boolean*
- `(file-mode>=?` *file-mode0 file-mode1*`)` → *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*`)` → *integer*
- `(integer->file-mode` *integer*`)` → *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

7.7 Time

These procedures are in structures `posix-time` and `posix`.

- `(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 `times`.

- `(time->string` *time*`)` $\rightarrow$ *string*

`Time->string` returns a string representation of *time* in the following form.

```
"Wed Jun 30 21:49:08 1993
"
```

7.8 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*`)` $\rightarrow$ *boolean*
- `(port->fd` *port*`)` $\rightarrow$ *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*`)` $\rightarrow$ *fd-port*
- `(dup-switching-mode` *fd-port*`)` $\rightarrow$ *fd-port*
- `(dup2` *fd-port file-descriptor*`)` $\rightarrow$ *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.

7.9 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. 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 5.14.

• (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 submatches? starts-line? ends-line?*`)`
  $\rightarrow$ *boolean or list of matches*
- `(match?` *x*`)` $\rightarrow$ *boolean*
- `(match-start` *match*`)` $\rightarrow$ *integer*
- `(match-end` *match*`)` $\rightarrow$ *integer*

`Regexp-match` matches the regular expression against the characters in *string*. 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.

7.10 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?`
`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`$\vert$`v][e`$\vert$`p`$\vert\epsilon$`]`	`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`

C procedure	Scheme procedure(s)
`getlogin`	`get-login-name`
`getpid`	`get-process-id`
`getppid`	`get-parent-process-id`
`getuid`	`get-user-id`
`isatty`	`port-is-a-terminal?`
`link`	`link`
`lstat`	`get-file/link-info`
`mkdir`	`make-directory`
`mkfifo`	`make-fifo`
`open`	`open-file`
`opendir`	`open-directory-stream`
`pipe`	`open-pipe`
`read`	`read-char, read-block`
`readdir`	`read-directory-stream`
`rename`	`rename`
`rmdir`	`remove-directory`
`setgid`	`set-group-id!`
`setuid`	`set-user-id!`
`stat`	`get-file-info`
`time`	`current-time`
`ttyname`	`port-terminal-name`
`umask`	`set-file-creation-mask!`
`uname`	`os-name, os-node-name, os-release-name, os-version-name, machine-name`
`unlink`	`unlink`
`waitpid`	`wait-for-child-process`
`write`	`write-char, write-block`

Appendix A

ASCII character encoding

"ASCII" stands for "American Standard Code for Information Interchange". The ASCII standard is a seven-bit code published by the United States of America Standards Institute (USASI) in 1968. The ASCII encoding forms the first half of ISO-8859-1 (Latin1) which in turn forms the first page of ISO 10646 (Unicode).

The Scheme 48 procedures `char->ascii` and `ascii->char` (section 5.1) give access to the ASCII encoding.

n_{10}	n_{16}		Unicode name	n_{10}	n_{16}		Unicode name
0	0	NUL	null	64	40	@	commercial at
1	1	SOH	start of heading	65	41	A	latin capital letter a
2	2	STX	start of text	66	42	B	latin capital letter b
3	3	ETX	end of text	67	43	C	latin capital letter c
4	4	EOT	end of transmission	68	44	D	latin capital letter d
5	5	ENQ	enquiry	69	45	E	latin capital letter e
6	6	ACK	acknowledge	70	46	F	latin capital letter f
7	7	BEL	bell	71	47	G	latin capital letter g
8	8	BS	backspace	72	48	H	latin capital letter h
9	9	HT	horizontal tabulation	73	49	I	latin capital letter i
10	A	LF	line feed	74	4A	J	latin capital letter j
11	B	VT	vertical tabulation	75	4B	K	latin capital letter k
12	C	FF	form feed	76	4C	L	latin capital letter l
13	D	CR	carriage return	77	4D	M	latin capital letter m
14	E	SO	shift out	78	4E	N	latin capital letter n
15	F	SI	shift in	79	4F	O	latin capital letter o
16	10	DLE	data link escape	80	50	P	latin capital letter p
17	11	DC1	device control one	81	51	Q	latin capital letter q
18	12	DC2	device control two	82	52	R	latin capital letter r
19	13	DC3	device control three	83	53	S	latin capital letter s
20	14	DC4	device control four	84	54	T	latin capital letter t
21	15	NAK	negative acknowledge	85	55	U	latin capital letter u
22	16	SYN	synchronous idle	86	56	V	latin capital letter v
23	17	ETB	end of transmission block	87	57	W	latin capital letter w
24	18	CAN	cancel	88	58	X	latin capital letter x
25	19	EM	end of medium	89	59	Y	latin capital letter y
26	1A	SUB	substitute	90	5A	Z	latin capital letter z
27	1B	ESC	escape	91	5B	[	left square bracket
28	1C	FS	file separator	92	5C	\	reverse solidus
29	1D	GS	group separator	93	5D	]	right square bracket

n_{10}	n_{16}		Unicode name	n_{10}	n_{16}		Unicode name
30	1E	RS	record separator	94	5E	^	circumflex accent
31	1F	US	unit separator	95	5F	_	low line
32	20	SPACE	space	96	60	`	grave accent
33	21	!	exclamation mark	97	61	a	latin small letter a
34	22	"	quotation mark	98	62	b	latin small letter b
35	23	#	number sign	99	63	c	latin small letter c
36	24	$	dollar sign	100	64	d	latin small letter d
37	25	%	percent sign	101	65	e	latin small letter e
38	26	&	ampersand	102	66	f	latin small letter f
39	27	'	apostrophe	103	67	g	latin small letter g
40	28	(	left parenthesis	104	68	h	latin small letter h
41	29	)	right parenthesis	105	69	i	latin small letter i
42	2A	*	asterisk	106	6A	j	latin small letter j
43	2B	+	plus sign	107	6B	k	latin small letter k
44	2C	,	comma	108	6C	l	latin small letter l
45	2D	-	hyphen-minus	109	6D	m	latin small letter m
46	2E	.	full stop	110	6E	n	latin small letter n
47	2F	/	solidus	111	6F	o	latin small letter o
48	30	0	digit zero	112	70	p	latin small letter p
49	31	1	digit one	113	71	q	latin small letter q
50	32	2	digit two	114	72	r	latin small letter r
51	33	3	digit three	115	73	s	latin small letter s
52	34	4	digit four	116	74	t	latin small letter t
53	35	5	digit five	117	75	u	latin small letter u
54	36	6	digit six	118	76	v	latin small letter v
55	37	7	digit seven	119	77	w	latin small letter w
56	38	8	digit eight	120	78	x	latin small letter x
57	39	9	digit nine	121	79	y	latin small letter y
58	3A	:	colon	122	7A	z	latin small letter z
59	3B	;	semicolon	123	7B	{	left curly bracket
60	3C	<	less-than sign	124	7C	|	vertical line
61	3D	=	equals sign	125	7D	}	right curly bracket
62	3E	>	greater-than sign	126	7E	~	tilde
63	3F	?	question mark	127	7F	DEL	delete

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] 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.

[10] Olin Shivers and Brian D. Calrstrom. Scsh Reference Manual, scsh release 0.4. MIT Laboratory for Computer Science Also available at URL http://swissnet.ai.mit.edu/scsh/.

[11] 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

The principal entry for each term, procedure, or keyword is listed first, separated from the other entries by a semicolon.