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

1 Introduction 1

2 User’s guide 2
  2.1 Command line arguments .......................... 2
  2.2 Command processor ................................ 3
  2.3 Editing .............................................. 3
  2.4 Performance ........................................ 3
  2.5 Disassembler ....................................... 4
  2.6 Module system ..................................... 4
  2.7 Library ............................................. 5

3 Command processor 7
  3.1 Current focus value and # # ......................... 7
  3.2 Command levels .................................... 8
  3.3 Logistical commands ............................... 8
  3.4 Module commands .................................. 9
  3.5 Debugging commands .............................. 9
  3.6 Switches .......................................... 10
  3.7 Inspection mode ................................... 12
  3.8 Command programs ................................ 13
  3.9 Building images ................................... 14
  3.10 Resource query and control ....................... 14
  3.11 Threads .......................................... 15
  3.12 Quite obscure .................................... 16

4 Module system 17
  4.1 Introduction ........................................ 17
  4.2 The configuration language ....................... 18
  4.3 Interfaces .......................................... 20
  4.4 Macros ............................................. 21
  4.5 Higher-order modules ............................. 21
  4.6 Compiling and linking ............................. 21
  4.7 Semantics of configuration mutation .............. 22
  4.8 Command processor support ...................... 23
  4.9 Configuration packages ......................... 25
  4.10 Discussion ...................................... 27
5 Libraries

5.1 General utilities ........................................ 29
5.2 Pretty-printing .......................................... 30
5.3 ASCII character encoding ................................. 31
5.4 Bitwise integer operations ............................... 31
5.5 Byte vectors ............................................ 31
5.6 Sparse vectors ........................................... 32
5.7 Cells ....................................................... 32
5.8 Queues ..................................................... 32
5.9 Arrays ....................................................... 33
5.10 Records .................................................... 34
   5.10.1 Low-level access to records ...................... 35
   5.10.2 Record types ...................................... 36
5.11 Finite record types .................................... 37
5.12 Sets over finite types ................................ 38
5.13 Hash tables ............................................. 39
5.14 Port extensions ......................................... 40
5.15 Fluid bindings .......................................... 41
5.16 Shell commands ........................................ 42
5.17 Sockets ................................................... 43
5.18 Macros for writing loops ............................... 44
   5.18.1 Iterate ............................................ 44
   5.18.2 Reduce ............................................. 45
   5.18.3 Sequence types .................................... 45
   5.18.4 Synchronous sequences ......................... 46
   5.18.5 Examples ........................................... 47
   5.18.6 Defining sequence types ......................... 48
   5.18.7 Expanded code .................................... 48
5.19 Sorting lists and vectors ............................... 49
   5.19.1 Design rules ...................................... 49
   5.19.2 Procedure specification ......................... 52
   5.19.3 Algorithmic properties ......................... 59
5.20 Regular expressions .................................... 60
   5.20.1 Character sets .................................... 60
   5.20.2 Anchoring ......................................... 61
   5.20.3 Composite expressions ......................... 61
   5.20.4 Case sensitivity .................................. 61
   5.20.5 Submatches and matching ....................... 62
5.21 SRFIs ..................................................... 63

6 Threads ...................................................... 66

6.1 Creating and controlling threads ....................... 66
6.2 Advanced thread handling .............................. 67
6.3 Debugging multithreaded programs .................... 67
6.4 Optimistic concurrency ................................. 67
6.5 Condition variables .................................... 70
6.6 Mutual exclusion ....................................... 71
6.6.1 Locks ................................................. 71
6.6.2 Placeholders ........................................... 72
6.7 Writing custom synchronization abstractions ................. 72

7 Mixing Scheme 48 and C ........................................... 74
7.1 Available facilities ........................................... 74
  7.1.1 Scheme structures ........................................ 74
  7.1.2 C naming conventions ...................................... 74
  7.1.3 Garbage collection .......................................... 75
7.2 Shared bindings .................................................. 75
  7.2.1 Exporting Scheme values to C ............................... 75
  7.2.2 Exporting C values to Scheme ................................ 75
  7.2.3 Complete shared binding interface ............................ 76
7.3 Calling C functions from Scheme ................................ 77
7.4 Adding external modules to the Makefile ....................... 77
7.5 Dynamic loading ............................................... 78
7.6 Compatibility ................................................. 79
7.7 Accessing Scheme data from C .................................. 79
  7.7.1 Constants ................................................ 79
  7.7.2 Converting values ......................................... 79
  7.7.3 C versions of Scheme procedures ............................ 80
7.8 Calling Scheme functions from C ................................ 81
7.9 Interacting with the Scheme heap ............................... 82
  7.9.1 Registering objects with the GC ............................. 82
  7.9.2 Keeping C data structures in the Scheme heap ............... 82
  7.9.3 C code and heap images .................................... 83
7.10 Using Scheme records in C code ............................... 83
7.11 Raising exceptions from external code ....................... 84
7.12 Unsafe functions and macros ................................... 85

8 Access to POSIX .................................................. 87
8.1 Process primitives ............................................. 87
  8.1.1 Process creation and termination ......................... 87
  8.1.2 Exec ................................................. 88
8.2 Signals ..................................................... 89
  8.2.1 POSIX signals ........................................... 89
  8.2.2 Other signals ........................................... 90
  8.2.3 Sending signals .......................................... 91
  8.2.4 Receiving signals ........................................ 91
8.3 Process environment ........................................... 91
  8.3.1 Process identification ..................................... 92
  8.3.2 Environment variables ..................................... 92
8.4 Users and groups .............................................. 92
8.5 OS and machine identification ................................ 93
8.6 Files and directories ........................................... 93
  8.6.1 Directory streams ......................................... 93
  8.6.2 Working directory ........................................ 94
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.6.3 File creation and removal</td>
<td>94</td>
</tr>
<tr>
<td>8.6.4 File information</td>
<td>96</td>
</tr>
<tr>
<td>8.6.5 File modes</td>
<td>97</td>
</tr>
<tr>
<td>8.7 Time</td>
<td>98</td>
</tr>
<tr>
<td>8.8 I/O</td>
<td>98</td>
</tr>
<tr>
<td>8.9 Regular expressions</td>
<td>99</td>
</tr>
<tr>
<td>8.10 C to Scheme correspondence</td>
<td>100</td>
</tr>
<tr>
<td>A ASCII character encoding</td>
<td>102</td>
</tr>
<tr>
<td>B Bibliography</td>
<td>104</td>
</tr>
<tr>
<td>C Index</td>
<td>105</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Scheme 48 is an implementation of the Scheme programming language as described in the Revised\textsuperscript{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\textsuperscript{5} Scheme Report, providing neither more nor less in the initial user environment. (This is not to say that more isn’t available in other environments; see below.)

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

Send mail to scheme-48-request@s48.org 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 7.5 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):

```lisp
(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:

```lisp
(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!)

(define-structure cells (export make-cell
          cell-ref
          cell-set!))
Cells could also have been implemented using the record facility described in section 5.10 and available in structure define-record-type.

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:

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

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 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 \texttt{push} command is used, the command processor invokes a recursive copy of itself, preserving the dynamic state of the program when the breakpoint occurred. The recursive invocation creates a new \textit{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: \texttt{>} or \texttt{:} for the base level and \texttt{n>} or \texttt{n}: for all other levels, where \texttt{n} is the command-level nesting depth. The \texttt{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 \texttt{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.

\texttt{(eof)}

Discards the current command level and resumes running the level down. \texttt{(eof)} is usually control-D at a Unix shell or control-C using the Emacs \texttt{cmuscheme} library.

\texttt{,pop}

The same as \texttt{(eof)}.

\texttt{,proceed \texttt{[exp ...]}}

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

\texttt{,push}

Pushes a new command level on above the current one. This is useful if the \texttt{auto-levels} switch has been used to disable the automatic pushing of new levels for errors and interrupts.

\texttt{,reset \texttt{[number]}}

Pops down to a given level and restarts that level. \texttt{Number} defaults to zero, \texttt{,reset} restarts the command processor, discarding all existing levels.

Whenever moving to an existing level, either by sending an \texttt{(eof)} or by using \texttt{,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

\texttt{,load \textit{filename} …}

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

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

3.4 Module commands

There are many commands related to modules. Only the most commonly used mod-
ule commands are described here; documentation for the rest can be found in sec-
tion 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 structures visible in the current package. The packages
associated with the structures 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 configu-
ration 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 continu-
   ation or thread’s stack is displayed.
2. Otherwise, if the current command level was initiated because of a break-
   point 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.

\texttt{\textbackslash run \textit{exp}}

Evaluate \textit{exp}, printing the result(s) and making them (or a list of them, if \textit{exp} returns multiple results) the new focus object. The \texttt{\textbackslash run} command is useful when writing command programs, which are described in section 3.8 below.

\texttt{\textbackslash trace \textit{name} ...}

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

\texttt{\textbackslash untrace \textit{name} ...}

Stop tracing calls to the named procedure or procedures. With no argument, stop tracing all calls to all procedures.

\texttt{\textbackslash condition}

The \texttt{\textbackslash 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 \texttt{\textbackslash condition} followed by \texttt{\textbackslash inspect} to inspect the procedure and its arguments.

\texttt{\textbackslash bound? \textit{name}}

Display the binding of \textit{name}, if there is one, and otherwise prints 'Not bound'.

\texttt{\textbackslash expand \textit{form}}

\texttt{\textbackslash expand-all \textit{form}}

Show macro expansion of \textit{form}, if any. \texttt{\textbackslash expand} performs a single macro expansion while \texttt{\textbackslash expand-all} fully expands all macros in \textit{form}.

\texttt{\textbackslash where \textit{procedure}}

Display name of file containing \textit{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 \texttt{\textbackslash set} and \texttt{\textbackslash unset} commands.

\texttt{\textbackslash set \textit{switch} \{on | off | ?\}}

This sets the value of mode-switch \textit{switch}. The second argument defaults to on. If the second argument is ? the value of \textit{switch} is is displayed and not changed. Doing \texttt{\textbackslash 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.
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:

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

> ,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:

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

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

### 3.8 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 \texttt{config} and \texttt{exec} itself) should be lists of the form \((\texttt{command-name} \ \text{argument} \ \ldots)\) where \texttt{command-name} is a symbol.

For example, the following two commands are equivalent:

\begin{verbatim}
,config ,load my-file.scm
,exec (config '(load "my-file.scm"))
\end{verbatim}

The file \texttt{scheme/vm/load-vm.scm} in the source directory contains an example of an \texttt{exec} program.

### 3.9 Building images

\begin{verbatim}
,dump filename [identification]
\end{verbatim}

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, \texttt{identification} should be a string (written with double quotes); this string will be part of the greeting message as the image starts up.

\begin{verbatim}
,build exp filename
\end{verbatim}

Like \texttt{dump}, except that the image file contains the value of \texttt{exp}, which should be a procedure of one argument, instead of the command processor. When \texttt{filename} is resumed, that procedure will be invoked on the VM's \texttt{-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 \texttt{,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 \texttt{,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

\begin{verbatim}
,time exp
\end{verbatim}

Measure execution time.

\begin{verbatim}
,collect
\end{verbatim}

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

\begin{verbatim}
,keep kind
\end{verbatim}
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.

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

\texttt{,go \textit{exp}}

This is like \texttt{,exit \textit{exp}} except that the evaluation of \textit{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 \textit{exp}. If an error occurs Scheme 48 will exit with a non-zero value.

\texttt{,translate \textit{from} to}

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

\texttt{,translate /usr/gjc/ /zu/gjc/}

will cause (\texttt{load "/usr/gjc/foo.scm"}) to have the same effect as (\texttt{load "/zu/gjc/foo.scm"}).

\texttt{,from-file \textit{filename} form ... ,end}

This is used by the cmuschem48 Emacs library to indicate the file from which the \textit{forms} came. \textit{Filename} is then used by the command processor to determine the package in which the \textit{forms} are to be evaluated.
Chapter 4
Module system

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

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

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

4.1 Introduction

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

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

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

A module imports bindings from other modules by either opening or 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))
```
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 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/)

18
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 \( \langle \text{command} \rangle \) s are applied to the names exported by \( \langle \text{structure} \rangle \) to produce a new set of names for the \( \langle \text{structure} \rangle \) 's bindings. Expose makes only the listed names visible. Hide makes all but the listed names visible. Rename makes each \( \langle \text{name} \rangle_0 \) visible as \( \langle \text{name} \rangle_1 \) name and not visible as \( \langle \text{name} \rangle_0 \), while alias makes each \( \langle \text{name} \rangle_0 \) visible as both \( \langle \text{name} \rangle_0 \) and \( \langle \text{name} \rangle_1 \). Prefix adds \( \langle \text{name} \rangle \) to the beginning of each exported name. The modifiers are applied from right to left. Thus

\[
\text{modify scheme (prefix foo/) (rename (car bus))})
\]

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 Revised5 Scheme report.

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

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

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

4.3 Interfaces

define-interface

An interface can be thought of as the type of a structure. In its basic form it is just a list of variable names, written (export name ...). However, in place of a name one may write (name type), indicating the type of name’s binding. The type field is optional, except that exported macros must be indicated with type :syntax.

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

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

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

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

1. It is important to separate the interface definition from the package definitions when there are multiple distinct structures that have the same interface — that is, multiple implementations of the same abstraction.
2. It is conceptually cleaner, and often useful for documentation purposes, to separate a module’s specification (interface) from its implementation (package).

3. Our experience is that configurations that are separated into interface definitions and package definitions are easier to read; the long lists of exported bindings just get in the way most of the time.

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

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

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

4.4 Macros

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

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

```
(delay exp) => (make-promise (lambda () exp)).
```

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

4.5 Higher-order modules

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

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

4.6 Compiling and linking

Scheme 48 has a static linker that produces stand-alone heap images from module descriptions. The programmer specifies a particular procedure in a particular structure to be the image’s startup procedure (entry point), and the linker traces dependency links as given by open and access clauses to determine the composition of the heap image.
There is not currently any provision for separate compilation; the only input to the static linker is source code. However, it will not be difficult to implement separate compilation. The unit of compilation is one module (not one file). Any opened or accessed structures from which macros are obtained must be processed to the extent of extracting its macro definitions. The compiler knows from the interface of an opened or accessed structure which of its exports are macros. Except for macros, a module may be compiled without any knowledge of the implementation of its opened and accessed structures. However, inter-module optimization may be available as an option.

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

4.7 Semantics of configuration mutation

During program development it is often desirable to make changes to packages and interfaces. In static languages it may be necessary to recompile and re-link a program in order for such changes to be reflected in a running system. Even in interactive Common Lisp implementations, a change to a package's exports often requires reloading clients that have already mentioned names whose bindings change. Once read resolves a use of a name to a symbol, that resolution is fixed, so a change in the way that a name resolves to a symbol can only be reflected by re-reading 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 \( \text{foo} \) that exports a name \( x \), and \( \text{foo} \)'s implementation obtains the binding of \( x \) as an import from some other structure \( \text{bar} \). Then \( C \) will see the binding from \( \text{bar} \). If one then alters \( \text{foo} \) so that it shadows \( \text{bar} \)'s binding of \( x \) with a definition of its own, then procedures in \( C \) that reference \( x \) will automatically see \( \text{foo} \)'s definition instead of the one from \( \text{bar} \) that they saw earlier.

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

### 4.8 Command processor support

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

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

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

```
,open structure
```

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

```
,open (subset foo (bar baz))
```
which only makes the bar and baz bindings from structure foo visible.

`config`

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

`config` command

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

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

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

`config-package-is struct-name`

The `config-package-is struct-name` 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
```
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).

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

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.

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.

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

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.

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

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

### 4.9 Configuration packages

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

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

Note that it does not open scheme.

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

For example:

```scheme
> ,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

```scheme
(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.

```scheme
> ,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 defined) 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 com-
positional. That is, structures are constructed by single syntactic units that compose
existing structures with a body of code. In Scheme Xerox, the set of modules that can
contribute to an interface is open-ended — any module can contribute bindings to any
interface whose name is in scope. The module system implementation is a cross-bar
that channels definitions from modules to interfaces. The module system described
here has simpler semantics and makes dependencies easier to trace. It also allows for
higher-order modules, which Scheme Xerox considers unimportant.
Chapter 5

Libraries

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

5.1 General utilities

These are in the big-util structure.

- \((\text{atom? } \text{value}) \rightarrow \text{boolean}\)

  \((\text{atom? } x)\) is the same as \((\text{not (pair? } x))\).

- \((\text{null-list? } \text{list}) \rightarrow \text{boolean}\)

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

- \((\text{neq? } \text{value value}) \rightarrow \text{boolean}\)

  \((\text{neq? } x \ y)\) is the same as \((\text{not (eq? } x \ y))\).

- \((\text{n= } \text{number number}) \rightarrow \text{boolean}\)

  \((\text{n= } x \ y)\) is the same as \((\text{not (= } x \ y))\).

- \((\text{identity value}) \rightarrow \text{value}\)

- \((\text{no-op value}) \rightarrow \text{value}\)

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

- \((\text{memq? } \text{value list}) \rightarrow \text{boolean}\)

  Returns true if \text{value} is in \text{list}, false otherwise.

- \((\text{any? } \text{predicate list}) \rightarrow \text{boolean}\)

  Returns true if \text{predicate} is true for any element of \text{list}.

- \((\text{every? } \text{predicate list}) \rightarrow \text{boolean}\)

  Returns true if \text{predicate} is true for every element of \text{list}.

- \((\text{any predicate list}) \rightarrow \text{value}\)

- \((\text{first predicate list}) \rightarrow \text{value}\)
Any returns some element of list for which predicate is true, or false if there are none. First does the same except that it returns the first element for which predicate is true.

- \((\text{filter} \ predicate \ list) \rightarrow \text{list}\)
- \((\text{filter!} \ predicate \ list) \rightarrow \text{list}\)

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

- \((\text{filter-map} \ procedure \ list) \rightarrow \text{list}\)

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

- \((\text{partition-list} \ predicate \ list) \rightarrow \text{list} \ \text{list}\)
- \((\text{partition-list!} \ predicate \ list) \rightarrow \text{list} \ \text{list}\)

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

- \((\text{remove-duplicates} \ list) \rightarrow \text{list}\)

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

- \((\text{delq} \ value \ list) \rightarrow \text{list}\)
- \((\text{delq!} \ value \ list) \rightarrow \text{list}\)
- \((\text{delete} \ predicate \ list) \rightarrow \text{list}\)

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

- \((\text{reverse!} \ list) \rightarrow \text{list}\)

Destructively reverses list.

- \((\text{concatenate-symbol} \ value \ldots) \rightarrow \text{symbol}\)

Returns the symbol whose name is produced by concatenating the displayed representations of value ....

\[(\text{concatenate-symbol} \ 'abc \ "-" \ 4) \Rightarrow 'abc-4\]

5.2 Pretty-printing

These are in the pp structure.

- \((\text{p} \ value)\)
- \((\text{p} \ value \ output-port)\)
- \((\text{pretty-print} \ value \ output-port \ position)\)

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

These are in the structure ascii.

- (char->ascii char) → integer
- (ascii->char integer) → char

These are identical to char->integer and integer->char except that they use the ASCII 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).

5.4 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) → integer
- (bitwise-ior integer integer) → integer
- (bitwise-xor integer integer) → integer
- (bitwise-not integer) → 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) → 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) → integer

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

5.5 Byte vectors

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

- (byte-vector? value) → boolean
- (make-byte-vector k fill) → byte-vector
- (byte-vector b ...) → byte-vector
- (byte-vector-length byte-vector) → integer
- (byte-vector-ref byte-vector k) → integer
- (byte-vector-set! byte-vector k b)
5.6 Sparse vectors

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

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

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

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

5.7 Cells

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

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

5.8 Queues

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

- `(make-queue) → queue`
- `(queue? value) → boolean`
- `(queue-empty? queue) → boolean`
- `(enqueue! queue value)`
- `(dequeue! queue) → 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) → integer`
- `(queue->list queue) → values`
- `(list->queue values) → queue`
- `(delete-from-queue! queue value) → boolean`
Queue-length returns the number of values in queue. Queue->list returns the values in queue as a list, in the order in which the values were added. 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.

5.9 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 dimension0 ...) → array
- (array dimensions element0 ...) → array
- (copy-array array) → array

Make-array makes a new array with the given dimensions, each of which must be a non-negative integer. Every element is initially set to value. Array Returns a new array with the given dimensions and elements. Dimensions must be a list of 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) → boolean

Returns #t if value is an array.

- (array-ref array index0 ...) → value
- (array-set! array value index0 ...)
- (array->vector array) → vector
- (array-dimensions array) → list

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

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

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

- (make-shared-array array linear-map dimension0 ...) → 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 dimensions given and must return a list of non-negative integers that are valid indexes into *array*. i

\[
(array-ref (make-shared-array a f i0 i1 ...) j0 j1 ...)
\]

is equivalent to

\[
(apply array-ref a (f j0 j1 ...))
\]

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

\[
(define (transpose array)
  (let ((dimensions (array-dimensions array)))
    (make-shared-array array
      (lambda (x y)
        (list y x))
      (cadr dimensions)
      (car dimensions))))
\]

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

### 5.10 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
- (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-names 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-tags 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

```scheme
(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:

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

### 5.10.1 Low-level access to records

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

These procedures are in the structure records.

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

These the same as the standard vector- procedures except that they operate on records. The value returned by record-length includes the slot holding the record’s type. (record-type x) is equivalent to (record-ref x 0).
5.10.2 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 returns true when passed a record of type `record-type` and false otherwise. `record-accessor` and `record-modifier` return procedures that reference and set the given field in records of the appropriate type.

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

`record-types` is the initial exporter of `define-record-discloser` (re-exported by `define-record-types` described above) and `define-record-resumer` (re-exported by `external-calls` (section 7.9.3)).

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

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

is (semantically) equivalent to

```scheme
(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.

5.11 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:

```scheme
(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.

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

```scheme
(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:

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

### 5.12 Sets over finite types

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

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

This defines `id` to be syntax for constructing sets, `type-name` to be a value representing the type, `predicate` to be a predicate for those sets, and `constructor` a procedure for constructing one from a list.
Element-syntax must be the name of a macro for constructing set elements from names (akin to the tag argument to define-enumerated-type). Element-predicate must be a predicate for the element type, all-elements a vector of all values of the element type, and element-index-ref must return the index of an element within the all-elements vector.

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

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

Here is an example. Given an enumerated type:

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

we can define sets of colors:

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

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

5.13 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.

### 5.14 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)

40
• (limit-output output-port n procedure)

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

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

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

(define p (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)

5.15 Fluid bindings

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

• (make-fluid value) → fluid
• (fluid fluid) → value
• (let-fluid fluid value thunk) → value(s)
• (let-fluids fluid0 value0 fluid1 value1 ... thunk) → value(s)
Make-fluid returns a new fluid with value as its initial top-level value. Fluid returns fluid’s current value. Let-fluid calls thunk, with fluid bound to value until thunk returns. Using a continuation to throw out of the call to thunk causes fluid to revert to its original value, while throwing back in causes fluid to be rebound to value. Let-fluid returns the value(s) returned by thunk. Let-fluids is identical to let-fluid except that it binds an arbitrary number of fluids to new values.

```lisp
(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
```

```lisp
(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
```

### 5.16 Shell commands

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

- (have-system?) → boolean
- (system string) → integer

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

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

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

- \( (\text{open-socket}) \rightarrow \text{socket} \)
- \( (\text{open-socket port-number}) \rightarrow \text{socket} \)
- \( (\text{socket-port-number socket}) \rightarrow \text{integer} \)
- \( (\text{close-socket socket}) \)
- \( (\text{socket-accept socket}) \rightarrow \text{input-port output-port} \)
- \( (\text{get-host-name}) \rightarrow \text{string} \)

Open-socket creates a new socket. If no \textit{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.

- \( (\text{socket-client host-name port-number}) \rightarrow \text{input-port output-port} \)

Socket-client connects to the server at \textit{port-number} on the machine named \textit{host-name}. Socket-client blocks until the server accepts the connection.

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

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

(define (get-id machine port-number)
  (call-with-values
   (lambda ()
    (socket-client machine port-number))
   (lambda (in out)
    (let ((id (read in)))
      (close-input-port in)
      (close-output-port out)
      id))))
```
5.18 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.

5.18.1 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-variables 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-variables, 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-variables, or returns some other value(s).

The loop-name and the state-variables and initial-values 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-types 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-variables are visible in final-expression but the sequence-variables 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).
5.18.2 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-variables 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:

\[
\begin{align*}
\text{(reduce} & \left(\begin{array}{c}
\text{(sequence-type element-variable sequence-data \ldots) } \\
\text{\ldots)} \\
\text{(state-variable initial-value)} \\
\text{\ldots)} \\
\text{body-expression} \\
\text{[final-expression]} \end{array}\right) \\
\end{align*}
\]

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.

\[
\begin{align*}
\text{(iterate loop} & \left(\begin{array}{c}
\text{(sequence-type element-variable sequence-data \ldots) } \\
\text{\ldots)} \\
\text{(state-variable initial-value)} \\
\text{\ldots)} \\
\text{(call-with-values (lambda ()} \\
\text{body-expression)} \\
\text{loop)} \\
\text{[final-expression]} \end{array}\right) \\
\end{align*}
\]

5.18.3 Sequence types

The predefined sequence types are:

- (list* elt-var list)   \hspace{1cm} \text{syntax}
- (vector* elt-var vector) \hspace{1cm} \text{syntax}
- (string* elt-var string) \hspace{1cm} \text{syntax}
- (count* elt-var start [end [step]]) \hspace{1cm} \text{syntax}
- (input* elt-var input-port read-procedure) \hspace{1cm} \text{syntax}
- (stream* elt-var procedure initial-data) \hspace{1cm} \text{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.
\[ \text{start, start + step, start + 2step, \ldots, end} \]

inclusive of \text{start} and exclusive of \text{end}. The default \text{step} is 1. The sequence does not terminate if no \text{end} is given or if there is no \( N > 0 \) such that \( \text{end} = \text{start} + N\text{step} \) (= is used to test for termination). For example, \( (\text{count}* i 0 -1) \) doesn’t terminate because it begins past the \text{end} value and \( (\text{count}* i 0 1 2) \) doesn’t terminate because it skips over the \text{end} value.

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

For a stream, the \text{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,

\[(\text{list* elt my-list})\]

is the same as

\[(\text{stream* elt list->stream my-list})\]

where \text{list->stream} is

\[(\lambda (\text{list})
   (\text{if} \ ((\text{null? list})
      (\text{values 'ignored #f})
      (\text{values (car list) (cdr list)}))))\]

### 5.18.4 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:

- \( (\text{list}^\% \text{ elt-var list}) \) syntax
- \( (\text{vector}^\% \text{ elt-var vector}) \) syntax
- \( (\text{string}^\% \text{ elt-var string}) \) syntax
- \( (\text{count}^\% \text{ elt-var start end [step]}) \) syntax
- \( (\text{input}^\% \text{ elt-var input-port read-procedure}) \) syntax
- \( (\text{stream}^\% \text{ elt-var procedure initial-data}) \) syntax

Note that the synchronous \text{count}^\% must have an \text{end}, unlike the nonsynchronous \text{count}^\%. 

46
5.18.5 Examples

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

\[
\begin{align*}
\text{(lambda} \ (\text{my-list} \ \text{predicate}) \\
\quad \text{(reduce} \ ((\text{list* elt my-list}) \\
\quad \quad \text{(count* i 0)}) \\
\quad \quad \text{(hits ‘())}) \\
\quad \text{(if} \ (\text{predicate elt}) \\
\quad \quad \text{(cons i hits)} \\
\quad \text{hits}) \\
\quad \text{(reverse hits)})
\end{align*}
\]

Looking for the index of an element of a list.

\[
\begin{align*}
\text{(lambda} \ (\text{my-list} \ \text{predicate}) \\
\quad \text{(iterate} \ \text{loop} \\
\quad \quad ((\text{list* elt my-list}) \\
\quad \quad \text{(count* i 0)}) \\
\quad \quad \text{())} \quad \text{; no state} \\
\quad \text{(if} \ (\text{predicate elt}) \\
\quad \quad i \\
\quad \quad \text{) (loop)))))
\end{align*}
\]

Reading one line.

\[
\begin{align*}
\text{(define} \ (\text{read-line} \ \text{port}) \\
\quad \text{(iterate} \ \text{loop} \\
\quad \quad ((\text{input* c port read-char}) \\
\quad \quad \text{(chars ‘())}) \\
\quad \quad \text{(if} \ (\text{char=}? \ c \ #\newline) \\
\quad \quad \text{(list->string} \ (\text{reverse} \ \text{chars})) \\
\quad \quad \text{(loop} \ (\text{cons c chars}))) \\
\quad \quad \text{(if} \ (\text{null?} \ \text{chars}) \\
\quad \quad \text{(eof-object)} \\
\quad \quad \text{; no newline at end of file} \\
\quad \quad \text{(list->string} \ (\text{reverse} \ \text{chars}))))
\end{align*}
\]

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

\[
\begin{align*}
\text{(define} \ (\text{line-count} \ \text{name}) \\
\quad \text{(call-with-input-file} \ \text{name} \\
\quad \quad \text{(lambda} \ (\text{in}) \\
\quad \quad \text{(reduce} \ ((\text{input*} \ 1 \ \text{in read-line}) \\
\quad \quad \quad ((i 0)) \\
\quad \quad \quad (+ \ i 1)))))
\end{align*}
\]
5.18.6 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 argu-
ments. 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

```scheme
(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 termina-
tion test to be used when some other synchronized method terminates the loop.

```scheme
(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 ...))))
```

5.18.7 Expanded code

The expansion of

```scheme
(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.

5.19 Sorting lists and vectors

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

The sort libraries in Scheme 48 include

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

5.19.1 Design rules

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

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

Detailed  you specify the actual algorithm (quick, heap, insert, merge). Different algorithms have different properties, both semantic and pragmatic, so these exports are necessary.
It is necessarily the case that the specifications of these procedures make statements about execution “pragmatics.” For example, the sole distinction between heap sort and quick sort—both of which are provided by this library—is one of execution time, which is not a “semantic” distinction. Similar resource-use statements are made about “iterative” procedures, meaning that they can execute on input of arbitrary size in a constant number of stack frames.

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

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

\[(\text{sort } < \text{ list})\]

not

\[(\text{sort list } <)\]

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

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

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

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

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

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

\[(\lambda x\ y\ (<\ (\lambda\ x\ y\ (\abs\ x)\ (\abs\ y)))\)]

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

\[\ldots 3, \ldots, -3 \ldots\]

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

\[\ldots 3, -3 \ldots\]
Choosing $<$ for the comparison procedure instead of $\leq$ affects how stability is coded. Given an adjacent pair $x, y$, $(< y x)$ means “$x$ should be moved in front of $x$”—otherwise, leave things as they are. So using a $\leq$ procedure where a $<$ procedure is expected will invert stability.

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

$$(\text{and} \ (\text{not} \ (< \ x \ y)) \ \ (\text{not} \ (< \ y \ x)))$$

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

$$(\text{and} \ (\leq \ x \ y) \ \ (\leq \ y \ x))$$

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

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

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

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

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

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

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

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

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

```
(lambdax (x y) (not (< x x)))
```

if need be.

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

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

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

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

51
Required vs. allowed side-effects List-sort! and List-stable-sort! are allowed, but not required, to alter their arguments’ cons cells to construct the result list. This is consistent with the what-not-how character of the group of procedures to which they belong (the sorting structure).

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

5.19.2 Procedure specification

<table>
<thead>
<tr>
<th>Structure name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorting</td>
<td>General sorting for lists and vectors</td>
</tr>
<tr>
<td>sorted</td>
<td>Sorted predicates for lists and vectors</td>
</tr>
<tr>
<td>list-merge-sort</td>
<td>List merge sort</td>
</tr>
<tr>
<td>vector-merge-sort</td>
<td>Vector merge sort</td>
</tr>
<tr>
<td>vector-heap-sort</td>
<td>Vector heap sort</td>
</tr>
<tr>
<td>vector-insert-sort</td>
<td>Vector insertion sort</td>
</tr>
<tr>
<td>delete-neighbor-duplicates</td>
<td>List and vector delete neighbor duplicates</td>
</tr>
<tr>
<td>binary-searches</td>
<td>Vector binary search</td>
</tr>
</tbody>
</table>

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

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

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

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

- A < or = parameter is a procedure accepting two arguments taken from the specified procedure’s data set(s), and returning a boolean;

- Start and end parameters are exact, non-negative integers that serve as vector indices selecting a subrange of some associated vector. When specified, they must satisfy the relation

\[ 0 \leq start \leq end \leq (\text{vector-length } \text{vector}) \]
where `vector` is the associated vector.

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

If a procedure is said to return “unspecified,” this means that nothing at all is said about what the procedure returns, not even the number of return values. Such a procedure is not even required to be consistent from call to call in the nature or number of its return values. It is simply required to return a value (or values) that may be passed to a command continuation, e.g. as the value of an expression appearing as a non-terminal subform of a `begin` expression. Note that in R5RS, this restricts such a procedure to returning a single value; non-R5RS systems may not even provide this restriction.

**sorting—general sorting package**

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

- `(list-sorted? < list) → boolean`
- `(list-merge < list₁ list₂) → list`
- `(list-merge! < list₁ list₂) → list`
- `(list-sort < lis) → list`
- `(list-sort! < lis) → list`
- `(list-stable-sort < list) → list`
- `(list-stable-sort! < list) → list`
- `(list-delete-neighbor-dups = list) → list`
- `(vector-sorted? < v [start [end]]) → boolean`
- `(vector-merge < v₁ v₂ [start₁ [end₁] [start₂ [end₂]])] → vector`
- `(vector-merge! < v v₁ v₂ [start [start₁ [end₁] [start₂ [end₂]])]) → vector`
- `(vector-sort < v [start [end]]) → vector`
- `(vector-sort! < v [start [end]]) → vector`
- `(vector-stable-sort < v [start [end]]) → vector`
- `(vector-stable-sort! < v [start [end]]) → vector`
- `(vector-delete-neighbor-dups = v [start [end]]) → vector`

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Suggested algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>list-sort</td>
<td>vector heap or quick</td>
</tr>
<tr>
<td>list-sort!</td>
<td>list merge sort</td>
</tr>
<tr>
<td>list-stable-sort</td>
<td>vector merge sort</td>
</tr>
<tr>
<td>list-stable-sort!</td>
<td>list merge sort</td>
</tr>
<tr>
<td>vector-sort</td>
<td>heap or quick sort</td>
</tr>
<tr>
<td>vector-sort! or quick sort</td>
<td>vector merge sort</td>
</tr>
<tr>
<td>vector-stable-sort</td>
<td>vector merge sort</td>
</tr>
<tr>
<td>vector-stable-sort!</td>
<td>merge sort</td>
</tr>
</tbody>
</table>

`List-Sorted?` and `vector-sorted?` return true if their input list or vector is in sorted order, as determined by their `<` comparison parameter.
All four merge operations are stable: an element of the initial list \( \text{list}_1 \) or vector \( \text{vector}_1 \) will come before an equal-comparing element in the second list \( \text{list}_2 \) or vector \( \text{vector}_2 \) in the result.

The procedures

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

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

The procedure

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

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

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

The vector procedures

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

do not alter their inputs, but allocate a fresh vector for their result, of length \( \text{end} - \text{start} \).

The vector procedures

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

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

Vector-merge returns a vector of length \( \text{end}_1 - \text{start}_1 + (\text{end}_2 - \text{start}_2) \).

Vector-merge! writes its result into vector \( v \) beginning at index \( \text{start} \), for indices less than \( \text{end} = \text{start} + (\text{end}_1 - \text{start}_1) + (\text{end}_2 - \text{start}_2) \). The target subvector \( v[\text{start}, \text{end}) \) may not overlap either source subvector \( \text{vector}_1[\text{start}_1, \text{end}_1) \) or \( \text{vector}_2[\text{start}_2, \text{end}_2) \).
The ...-delete-neighbor-dups-... procedures: These procedures delete adjacent duplicate elements from a list or a vector, using a given element-equality procedure. The first/leftmost element of a run of equal elements is the one that survives. The list or vector is not otherwise disordered.

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

The comparison procedure $= \!: x \: y$ where $x$ comes before $y$ in the containing list or vector.

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

Examples:

(list-delete-neighbor-dups $\!=$(1 1 2 7 7 0 -2 -2))
\[\Longrightarrow (1 \ 2 \ 7 \ 0 \ -2)\]

(vector-delete-neighbor-dups $\!=(#(1 1 2 7 7 0 -2 -2))$
\[\Longrightarrow #(1 \ 2 \ 7 \ 0 \ -2)\]

(vector-delete-neighbor-dups $\!=(#(1 1 2 7 7 0 -2 -2) \ 3 \ 7)$
\[\Longrightarrow #(7 \ 0 \ -2)\]

Algorithm-specific sorting packages

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

**sorted**—sorted predicates

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

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

- \((\text{list-merge-sort} < \text{list}) \rightarrow \text{list}\)
- \((\text{list-merge-sort!} < \text{list}) \rightarrow \text{list}\)
- \((\text{list-merge} \text{list}_1 < \text{list}_2) \rightarrow \text{list}\)
- \((\text{list-merge!} \text{list}_1 < \text{list}_2) \rightarrow \text{list}\)

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

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

The merge operations are stable: an element of \(\text{list}_1\) will come before an equal-comparing element in \(\text{list}_2\) in the result list.

vector-merge-sort—vector merge sort

- \((\text{vector-merge-sort} < \text{vector} \text{[start [end [temp]]]})) \rightarrow \text{vector}\)
- \((\text{vector-merge-sort!} < \text{vector} \text{[start [end [temp]]]})) \rightarrow \text{vector}\)
- \((\text{vector-merge} < \text{vector}_1 \text{vector}_2 \text{[start}_1 \text{[end}_1 \text{[start}_2 \text{[end}_2]]]})) \rightarrow \text{vector}\)
- \((\text{vector-merge!} < \text{vector} \text{vector}_1 \text{vector}_2 \text{[start}_1 \text{[end}_1 \text{[start}_2 \text{[end}_2]]]})) \rightarrow \text{vector}\)

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

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

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

The merge operations are stable: an element of \(\text{vector}_1\) will come before an equal-comparing element in \(\text{vector}_2\) in the result vector.

- Vector-merge-sort! leaves its result in vector[start,end).
- Vector-merge-sort returns a vector of length end – start.
- Vector-merge returns a vector of length \((\text{end}_1 – \text{start}_1) + (\text{end}_2 – \text{start}_2)\).
- Vector-merge! writes its result into vector, beginning at index start, for indices less than end = start + \((\text{end}_1 – \text{start}_1) + (\text{end}_2 – \text{start}_2)\). The target subvector

\[ \text{vector}[\text{start}, \text{end}] \]

may not overlap either source subvector

\[ \text{vector}_1[\text{start}_1, \text{end}_1), \text{ or vector}_2[\text{start}_2, \text{end}_2). \]
**vector-heap-sort**—vector heap sort

- (vector-heap-sort < vector [start [end]]) → vector
- (vector-heap-sort! < vector [start [end]])

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

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

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

- (vector-insert-sort < vector [start [end]]) → vector
- (vector-insert-sort! < vector [start [end]])

These procedures stably sort their data using insertion sort.

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

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

- (list-delete-neighbor-dups = list) → list
- (list-delete-neighbor-dups! = list) → list
- (vector-delete-neighbor-dups = vector [start [end]]) → vector
- (vector-delete-neighbor-dups! = vector [start [end]]) → end’

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

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

The comparison procedure = passed to these procedures is always applied

\[
(= \; x \; y)
\]

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

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

Examples:

\[
\begin{align*}
\text{(list-delete-neighbor-dups} &= \text{'(1 1 2 7 7 0 -2 -2))} \\
\implies (1 2 7 0 -2)
\end{align*}
\]

\[
\begin{align*}
\text{(vector-delete-neighbor-dups} &= \text{'#(1 1 2 7 7 0 -2 -2))} \\
\implies #(1 2 7 0 -2)
\end{align*}
\]

\[
\begin{align*}
\text{(vector-delete-neighbor-dups} &= \text{'#(1 1 2 7 7 0 -2 -2) 3 7)} \\
\implies #(7 0 -2)
\end{align*}
\]

;; Result left in \(v[3,9)\):

\[
\begin{align*}
\text{(let ((v (vector 0 0 0 1 1 2 2 3 3 4 4 5 5 6 6)))} \\
\text{(cons (vector-delete-neighbor-dups! = v 3) v))} \\
\implies (9 . #(0 0 1 2 3 4 5 6 4 4 5 5 6 6))
\end{align*}
\]

**binary-searches**—vector binary search

• **(vector-binary-search** \(<\) \(elt\rightarrow\text{key}\) \(key\) \(vector\) \([start [end]])\) \(\rightarrow\) integer or \#f
• **(vector-binary-search3** \(\text{compare-proc}\) \(vector\) \([start [end]])\) \(\rightarrow\) integer or \#f

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

\[
\begin{align*}
\text{(.vector-sorted? (lambda (x y) (< (elt\rightarrow\text{key} x) (elt\rightarrow\text{key} y))))} \\
\text{vector start end) } \implies \text{true}
\end{align*}
\]

An element \(e\) of \(vector\) is a match for \(key\) if it’s neither less nor greater than the key:

\[
\begin{align*}
\text{(and (not (< (elt\rightarrow\text{key} e) key))} \\
\text{(not (< key (elt\rightarrow\text{key} e))))}
\end{align*}
\]

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

\[
\begin{align*}
\text{(vector-binary-search < car 4 '#((1 . one) (3 . three) (4 . four) (25 . twenty-five)))} \\
\implies 2
\end{align*}
\]
(vector-binary-search < car 7 '(((1 . one) (3 . three) (4 . four) (25 . twenty-five)))
⇒ #f

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

(compare-proc x) < 0 ⇒ x < search-key
(compare-proc x) = 0 ⇒ x = search-key
(compare-proc x) > 0 ⇒ x > search-key

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

5.19.3 Algorithmic properties

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

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

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

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

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

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

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

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

Pure list merge sort Stable and fast—O(n log(n)) worst-case, and possibly O(n), depending upon the input list (see discussion above).
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Stable?</th>
<th>Worst case</th>
<th>Average case</th>
<th>In-place</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector insert</td>
<td>Yes</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>Yes</td>
</tr>
<tr>
<td>Vector quick</td>
<td>No</td>
<td>$O(n^2)$</td>
<td>$O(n \log(n))$</td>
<td>Yes</td>
</tr>
<tr>
<td>Vector heap</td>
<td>No</td>
<td>$O(n \log(n))$</td>
<td>$O(n \log(n))$</td>
<td>Yes</td>
</tr>
<tr>
<td>Vector merge</td>
<td>Yes</td>
<td>$O(n \log(n))$</td>
<td>$O(n \log(n))$</td>
<td>No</td>
</tr>
<tr>
<td>List merge</td>
<td>Yes</td>
<td>$O(n \log(n))$</td>
<td>$O(n \log(n))$</td>
<td>Either</td>
</tr>
</tbody>
</table>

## 5.20 Regular expressions

This section describes a functional interface for building regular expressions and matching them against strings. The matching is done using the POSIX regular expression package. Regular expressions are in the structure `regexps`. A regular expression is either a character set, which matches any character in the set, or a composite expression containing one or more subexpressions. A regular expression can be matched against a string to determine success or failure, and to determine the substrings matched by particular subexpressions.

### 5.20.1 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 "!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~")

60
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.

5.20.2 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.

5.20.3 Composite expressions

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

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

- (text string) → reg-exp

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

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

Repeat returns a regular expression that matches zero or more occurrences of its reg-exp argument. With no count the result will match any number of times (reg-exp*). With a single count the returned expression will match reg-exp exactly that number of times. The final case will match from min to max repetitions, inclusive. Max may be #f, in which case there is no maximum number of matches. Count and min should be exact, non-negative integers; max should either be an exact non-negative integer or #f.

5.20.4 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".

```lisp
(text "ab")
```
while

```lisp
(ignore-case (text "ab"))
```
matches "ab", "aB", "Ab", and "AB" and

```lisp
(ignore-case (sequence (text "a")
  (use-case (text "b"))))
```
matches "ab" and "Ab" but not "aB" or "AB".

### 5.20.5 Submatches and matching

A subexpression within a larger expression can be marked as a submatch. When an expression is matched against a string, the success or failure of each submatch within that expression is reported, as well as the location of the substring matched be each successful submatch.

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

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

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

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

`Match` returns `#f` if `reg-exp` does not match `string` and a match record if it does match. A match record contains three values: the beginning and end of the substring that matched the pattern and an a-list of submatch keys and corresponding match records for any submatches that also matched. `Match-start` returns the index of the first character in the matching substring and `match-end` gives index of the first character after the matching substring. `Match-submatches` returns an alist of submatch keys and match records. Only the top match record returned by `match` has a submatch alist.
Matching occurs according to POSIX. The match returned is the one with the lowest starting index in `string`. If there is more than one such match, the longest is returned. Within that match the longest possible submatches are returned.

All three matching procedures cache a compiled version of `reg-exp`. Subsequent calls with the same `reg-exp` will be more efficient.

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

```scheme
(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)))

5.21 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
• SRFI 11 – Syntax for receiving multiple values
• SRFI 13 – String Library
• SRFI 14 – Character-Set Library (see note below)
• SRFI 16 – Syntax for procedures of variable arity
• SRFI 17 – Generalized set!
• SRFI 22 – Running Scheme Scripts on Unix
• SRFI 23 – Error reporting mechanism
• SRFI 25 – Multi-dimensional Array Primitives
• SRFI 26 – Notation for Specializing Parameters without Currying
• SRFI 27 – Sources of Random Bits
• SRFI 28 – Basic Format Strings
• SRFI 31 – A special form rec for recursive evaluation
• SRFI 37 – args-fold: a program argument processor
• SRFI 42 – Eager Comprehensions
• SRFI 45 – Primitives for Expressing Iterative Lazy Algorithms

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

SRFI 14 includes the procedure ->char-set which is not a standard Scheme identifier (in R5RS the only required identifier starting with – is – itself). In the Scheme 48 version of SRFI 14 we have renamed ->char-set as x->char-set.

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

Threads

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

6.1 Creating and controlling threads

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

- (spawn thunk) → thread
- (spawn thunk name) → thread

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

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

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

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

- (current-thread) → thread
- (thread? thing) → boolean
- (thread-name thread) → name
- (thread-uid thread) → integer

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

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

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

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

6.3 Debugging multithreaded programs

Debugging multithreaded programs can be difficult.

As described in section 3.11, when any thread signals an error, Scheme 48 stops running all of the threads at that command level.

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

- `(debug-message element0 ...)`

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>pair</td>
<td>(...</td>
</tr>
<tr>
<td>vector</td>
<td>#(...)</td>
</tr>
<tr>
<td>procedure</td>
<td>#{procedure}</td>
</tr>
<tr>
<td>record</td>
<td>#{&lt;name of record type&gt;}</td>
</tr>
<tr>
<td>all others</td>
<td>???</td>
</tr>
</tbody>
</table>

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

6.4 Optimistic concurrency

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

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

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

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

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

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

- (provisional-car pair) → value
- (provisional-cdr pair) → value
- (provisional-set-car! pair value)
- (provisional-set-cdr! pair value)
- (provisional-cell-ref cell) → value
- (provisional-cell-set! cell value)
- (provisional-vector-ref vector i) → value
- (provisional-vector-set! vector i value)
- (provisional-string-ref vector i) → char
- (provisional-string-set! vector i char)
- (provisional-byte-vector-ref vector i k)
- (provisional-byte-vector-set! vector i k)

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

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

(define (make-counter)
  (let ((value 0)))
Here is the same procedure using a proposal to ensure that each increment operation happens atomically. The value of the counter is kept in a cell (see section 5.7 to allow the use of logging operations.

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

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

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

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

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

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

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

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

\[
\cdot (\text{with-new-proposal (lose) exp ...}) \rightarrow \text{value} \dots
\]

Syntax

With-new-proposal saves the current proposal, installs a new one, executes the forms in the body, and returns whatever they return. It also binds lose to a thunk repeating the procedure of installing a new procedure and running the body. Typically, the body will call maybe-commit and, if that fails, call lose to try again.

The following procedures give access to the low-level proposal mechanism. They are defined in the low-proposals structure.

\[
\cdot (\text{maybe-commit}) \rightarrow \text{boolean} \\
\cdot (\text{make-proposal}) \rightarrow \text{proposal} \\
\cdot (\text{current-proposal}) \rightarrow \text{proposal} \\
\cdot (\text{set-current-proposal! proposal})
\]

Maybe-commit verifies that any reads logged in the current proposal are still valid and, if so, performs any writes that it contains. A logged read is valid if, at the time of the commit, the location read contains the same value it had at the time of the original read (note that this does not mean that no change occurred, simply that the value now is the same as the value then). Maybe-commit returns #t if the commit succeeds and #f if it fails.

Make-proposal creates a new proposal. Current-proposal and set-current-proposal! access and set the current thread’s proposal. It is an error to pass to set-current-proposal! a proposal that is already in use.

6.5 Condition variables

Condition variables (defined in the condvars structure) allow threads to perform condition synchronization: It allows threads to block, waiting for a specified condition—associated with a condition variable—to occur, and other threads to wake up the waiting threads when the condition is fulfilled.

Note that, in Scheme 48, condition variables work in conjunction with proposals, not with mutex locks or semaphores, as in most other implementations of this concept.

\[
\cdot (\text{make-condvar}) \rightarrow \text{condvar} \\
\cdot (\text{make-condvar id}) \rightarrow \text{condvar} \\
\cdot (\text{condvar? thing}) \rightarrow \text{boolean} \\
\cdot (\text{set-condvar-has-value?! condvar boolean}) \\
\cdot (\text{condvar-has-value? condvar}) \rightarrow \text{boolean} \\
\cdot (\text{set-condvar-value! condvar value}) \\
\cdot (\text{condvar-value condvar}) \rightarrow \text{value} \\
\cdot (\text{maybe-commit-and-wait-for-condvar condvar}) \rightarrow \text{boolean} \\
\cdot (\text{maybe-commit-and-set-condvar! condvar value}) \rightarrow \text{boolean}
\]
Make-condvar creates a condition variable. (The optional id argument is only for debugging purposes; the discloser for condition variables prints it out if present.) Condvar? is the predicate for condition variables.

Each condition variable has an associated value and a flag has-value? signalling if the condition has already occurred. The accessor for flag is condvar-has-value?; set-condvar-has-value?! sets it. Both are provisional operations and go through the current proposal. Set-condvar-value! sets the value of the condition variable ( provisionally), and condvar-value extracts it.

Maybe-commit-and-wait-for-condvar attempts to commit the current proposal. If the commit succeeds, it suspends the current thread and registers it with the condvar condition variable. Upon waking up again maybe-commit-and-wait-for-condvar returns #t. If the commit fails, maybe-commit-and-set-condvar returns #f.

Maybe-commit-and-set-condvar! sets the value of the condvar condition variable to value, ( provisionally) sets the has-value? flag to #t, and then attempt to commit the current proposal. Upon success, it wakes up all suspended threads registered with condvar and returns #t, otherwise, it returns #f.

6.6 Mutual exclusion

Scheme 48 also has more traditional mutual-exclusion synchronization abstractions, specifically mutex locks and placeholders. Note that typically synchronization via optimistic concurrency is usually preferable: Mutual exclusion often puts the running program into an inconsistent state for the time of the inclusion, which has adverse effects on modularity and interruptibility.

6.6.1 Locks

The locks structure contains bindings that implement standard mutex locks:

- (make-lock) \(\rightarrow\) lock
- (lock? thing) \(\rightarrow\) boolean
- (obtain-lock lock)
- (maybe-obtain-lock lock) \(\rightarrow\) boolean
- (release-lock lock)

Make-lock creates a lock in the “released” state. Lock? is the predicate for locks.

Obtain-lock atomically checks if lock is in the “released” state. If it is, the lock is put into the “obtained” state, and obtain-lock returns immediately. If the lock is in the “obtained” state, the current thread is suspended and registered with the lock. Maybe-obtain-lock, like obtain-lock, checks the state of lock: if it is “released,” the lock is put into the “obtained” state, if it is “obtained,” maybe-obtain-lock returns immediately. Maybe-obtain-lock returns #t if it was able to obtain the lock, and #f otherwise.

Release-lock does nothing if lock is in the “released” state. If it is in the “obtained” state, release-lock causes one of the threads suspended on an obtain-lock lock operation to continue execution. If that thread is the last thread
registered with the lock, the lock is transferred to the “released” state. In any case, release-lock returns immediately.

6.6.2 Placeholders

The placeholders structure contains bindings for placeholders—thread-safe, write-once variables, akin to ID-90 I-structures or CML I-variables.

The typical scenario for placeholders is that, say, a thread A computes a value needed by another thread B at some unspecified time. Both threads share access to a placeholder; when A has computed the value, it places it into the placeholder. When B needs the value, it extracts it from placeholder, blocking if necessary.

- (make-placeholder) → placeholder
- (make-placeholder id) → placeholder
- (placeholder? thing) → boolean
- (placeholder-set! placeholder value)
- (placeholder-value placeholder) → value

Make-placeholder creates an empty placeholder. (The optional id argument is only for debugging purposes; the discloser for placeholders prints it out if present.) Placeholder? is the predicate for placeholders.

Placeholder-set! places a value into a placeholder. Doing this more than once signals an error. Placeholder-value extracts the value from the placeholder and returns it. If the placeholder is empty, it blocks the current thread until it becomes full.

6.7 Writing custom synchronization abstractions

The bindings explained in this section are part of the threads-internal structure. They are concerned with suspending threads and making them runnable again upon some later event.

Typically, a suspended thread needs to be recorded in a queue somewhere for later waking-up. To allow a thread to be recorded in multiple queues (say, when it waits for one of a number of events), such thread queues are ordinary queues containing cells that, in turn, contain the thread objects themselves. Each thread has at most one such cell associated with it which is shared among all queues (or other data structures) holding on to the suspended thread. The cell is cleared when the thread is woken up.

- (thread-queue-empty? thread-queue) → boolean
- (maybe-dequeue-thread! thread-queue) → boolean

Thread-queue-empty? atomically checks whether the thread-queue thread queue is empty, i.e., if it does not contain non-empty cells. Maybe-dequeue-thread! provisionally dequeues a thread from thread-queue if it contains one. It returns the dequeued thread or #f if the queue is empty.

- (maybe-commit-and-block cell) → boolean
- (maybe-commit-and-block-on-queue thread-queue) → boolean
- (maybe-commit-and-make-ready thread-or-queue) → boolean
Maybe-commit-and-block attempts to commit the current proposal. If this succeeds, the current thread is blocked, the thread’s cell is set to cell, and #t is returned. Otherwise, #f is returned. Maybe-commit-and-block-on-queue is like maybe-commit-and-block, excepts that it creates a fresh cell for the thread and enqueues it in thread-queue if the commit succeeds.

Maybe-commit-and-make-ready accepts either a thread object or a thread queue as an argument. In either case, maybe-commit-and-make-ready tries to commit the current proposal. If that succeeds, maybe-commit-and-make-ready makes its argument runnable: if thread-or-queue is a thread, that thread is made runnable, if it is a thread queue, all threads on the queue are made runnable. (In the latter case, none of the threads actually runs until all have been made runnable.) Maybe-commit-and-make-ready returns #t if it succeeded, and #f otherwise.
Chapter 7
Mixing Scheme 48 and C

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.

7.1 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.

7.1.1 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 7.5, and shared-bindings, which has the additional shared-binding functions described in Section 7.2.3.

7.1.2 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.

7.1.3 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 collector. See section 7.9 for more information.

7.2 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.

7.2.1 Exporting Scheme values to C

- (define-exported-binding name value) → 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.

7.2.2 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

```scheme
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 7.9.

- `(import-definition name)` syntax
- `(import-definition name c-name)` syntax

These macros simplify importing definitions from C to Scheme. They expand into

```scheme
(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

```scheme
(define my-foo (lookup-imported-binding "my_foo"))
```

### 7.2.3 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)`
### 7.3 Calling C functions from Scheme

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

- \( \text{(call-imported-binding binding arg}_0 \ldots) \to \text{value} \)
- \( \text{(call-external external arg}_0 \ldots) \to \text{value} \)
- \( \text{(call-external-value value name arg}_0 \ldots) \to \text{value} \)

Each of these applies its first argument, a C function, to the rest of the arguments. For \text{call-imported-binding} the function argument must be an imported binding. For \text{call-external} the function argument must be an external bound in the current process (see Section 7.5). For \text{call-external-value} \text{value} must be a byte vector whose contents is a pointer to a C function and \text{name} should be a string naming the function. The \text{name} argument is used only for printing error messages.

For all of these, the C function is passed the \text{arg}_i \text{ 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).

- \( \text{(import-lambda-definition name (formal \ldots))} \) syntax
- \( \text{(import-lambda-definition name (formal \ldots) c-name)} \) syntax

These macros simplify importing functions from C. They define \text{name} to be a function with the given formals that applies those formals to the corresponding C binding. \text{C-name}, if supplied, should be a string. These expand into

\[
\begin{align*}
\text{(define temp (lookup-imported-binding c-name))} \\
\text{(define name (lambda (formal \ldots)) (call-imported-binding temp formal \ldots))}
\end{align*}
\]

If \text{c-name} is not supplied, it is derived from \text{name} by converting all letters to lowercase and replacing ‘-’ with ‘.’.

### 7.4 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 (\text{scheme48vm}). \text{EXTERNALOBJECTS} lists the object files to be included in \text{scheme48vm}, \text{EXTERNALFLAGS} is a list of ld flags to be used when creating \text{scheme48vm}, and \text{EXTERNALINITIALIZERS} is a list of C procedures to be called on startup. The procedures listed in \text{EXTERNALINITIALIZERS} should
7.5 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 if any are unbound.

- (call-external external arg0 ...) → value

An external whose value is a C procedure can be called using call-external. See Section 7.3 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.
7.6 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.

7.7 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.

7.7.1 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).

7.7.2 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_EXTRACTBOOLEAN is false if its argument is #f and true otherwise. S48_ENTERBOOLEAN 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 7.9.

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).

### 7.7.3 C versions of Scheme procedures

The following macros and procedures are C versions ofScheme 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)
- 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)
- int S48_STRING_P(s48_value)
7.8 Calling Scheme functions from C

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

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

This calls the Scheme procedure \( p \) on \( \text{nargs} \) arguments, which are passed as additional arguments to \( \text{s48_call_scheme} \). 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 \text{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.
7.9 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.

7.9.1 Registering objects with the GC

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

- void S48_GC_PROTECT(n)
- void S48_GC_PROTECT(n) (s48_value1, ..., s48_value_n)
- void S48_GC_UNPROTECT()

S48_DECLARE_GC_PROTECT(n), where 1 ≤ n ≤ 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.

7.9.2 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 \texttt{S48\_EXTRACT\_VALUE\_POINTER} is valid only until the next garbage collection.
\texttt{S48\_SET\_VALUE} stores value into the byte vector.

7.9.3 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)

\texttt{Define-record-resumer} defines \texttt{procedure}, which should accept one argument, to be the resumer for \texttt{record-type}. The order in which resumer procedures are called is not specified.

The \texttt{procedure} argument to \texttt{define-record-resumer} may be \texttt{#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.

7.10 Using Scheme records in C code

External modules can create records and access their slots positionally.

* \texttt{s48\_value s48\_make\_record(s48\_value)} \hspace{1cm} (may GC)
* \texttt{int S48\_RECORD\_P(s48\_value)}
* \texttt{s48\_value S48\_RECORD\_TYPE(s48\_value)}
* \texttt{s48\_value S48\_RECORD\_REF(s48\_value, long)}
* \texttt{void S48\_RECORD\_SET(s48\_value, long, s48\_value)}

The argument to \texttt{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

\begin{verbatim}
(define-record-type thing :thing
  (make-thing a b)
  thing?
  (a thing-a)
  (b thing-b))
\end{verbatim}
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:

```c
static s48_value
    thing_record_type_binding = S48_FALSE;

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

s48_value make_thing(s48_value a, s48_value b)
{
    s48_value thing;
    s48_DECLARE_GC_PROTECT(2);
    S48_GC_PROTECT_2(a, b);
    thing = s48_make_record(thing_record_type_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 occurring during the call to s48_make_record().

### 7.11 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(s48_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)

7.12 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 S48Unsafe_extract_char(s48_value)
• char * S48Unsafe_extract_string(s48_value)
• long S48Unsafe_extract_integer(s48_value)
• long S48Unsafe_extract_double(s48_value)
• long S48Unsafe_extract_fixnum(s48_value)
• s48_value S48Unsafe_enter_fixnum(long)
• s48_value S48Unsafe_car(s48_value)
• s48_value S48Unsafe_cdr(s48_value)
• void S48Unsafe_set_car(s48_value, s48_value)
• void S48Unsafe_set_cdr(s48_value, s48_value)
• long S48Unsafe_vector_length(s48_value)
• s48_value S48Unsafe_vector_ref(s48_value, long)
• void S48Unsafe_vector_set(s48_value, long, s48_value)
• long S48Unsafe_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 8

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 8.10 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
- posix-process-data
- posix-files
- posix-i/o
- posix-time
- posix-users
- posix-regexps
- posix

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.

8.1 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.

8.1.1 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.

- \((\text{process-id? } x) \rightarrow \text{boolean}\)
- \((\text{process-id=}? \text{process-id0} \text{process-id1}) \rightarrow \text{boolean}\)
- \((\text{process-id->integer} \text{process-id}) \rightarrow \text{integer}\)
- \((\text{integer->process-id} \text{integer}) \rightarrow \text{process-id}\)

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.

- \((\text{process-id-exit-status} \text{process-id}) \rightarrow \text{integer or } \#f\)
- \((\text{process-id-terminating-signal} \text{process-id}) \rightarrow \text{signal or } \#f\)
- \((\text{wait-for-child-process} \text{process-id})\)

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.

- \((\text{exit} \text{status})\)

Terminates the current process with the integer status as its exit status.

8.1.2 Exec

- \((\text{exec} \text{program-name} \text{arg0} \ldots)\)
- \((\text{exec-with-environment} \text{program-name} \text{env} \text{arg0} \ldots)\)
- \((\text{exec-file} \text{filename} \text{arg0} \ldots)\)
- \((\text{exec-file-with-environment} \text{filename} \text{env} \text{arg0} \ldots)\)
- \((\text{exec-with-alias} \text{name} \text{lookup}? \text{maybe-env} \text{arguments})\)

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 (exec-with-environment). Program-name and filename and any arg, 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).

8.2 Signals

There are two varieties of signals available, named and anonymous. A named signal is one for which we have a symbolic name, such as kill or pipe. Anonymous signals, for which we only have the current operating system’s signal number, have no meaning in other operating systems. Named signals preserve their meaning in image files. Not all named signals are available from all OS’s and there may be multiple names for a single OS signal number.

- (signal signal-name) → signal  
  syntax
- (name->signal symbol) → signal or \#f
- (integer->signal integer) → signal
- (signal? x) → boolean
- (signal-name signal) → symbol or \#f
- (signal-os-number signal) → integer
- (signal=? signal0 signal1) → boolean

The syntax signal returns a (named) signal associated with signal-name. Name->signal returns a (named) signal or \#f if the the signal name is not supported by the operating system. The signal returned by integer->signal is a named signal if integer corresponds to a named signal in the current operating system; otherwise it returns an anonymous signal. Signal-name returns a symbol if signal is named and \#f if it is anonymous. Signal=? returns \#t if signal0 and signal1 have the same operating system number and \#f if they do not.

8.2.1 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

8.2.2 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 stack fault on coprocessor
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)
vtalarm 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
8.2.3 Sending signals

- (signal-process process-id signal)

Send signal to the process corresponding to process-id.

8.2.4 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 occurrences 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.

8.3 Process environment

These are in structures posix-process-data and posix.
8.3.1 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 8.1.1 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.

8.3.2 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.

8.4 Users and groups

User-ids and group-ids 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.

### 8.5 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

### 8.6 Files and directories

These procedures are in structures posix-files and posix.

#### 8.6.1 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.

8.6.2 Working directory

• (working-directory) → string
• (set-working-directory! string)

These return and set the working directory.

8.6.3 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-names` are: `read`, `write`, `execute`, `exists`. 

95
8.6.4 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

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.
8.6.5 File modes

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

- `(file-mode permission-name ...) → file-mode`  
  - `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.

<table>
<thead>
<tr>
<th>Permission name</th>
<th>Bit mask</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set-uid</td>
<td>#o4000</td>
<td>set user id when executing</td>
</tr>
<tr>
<td>set-gid</td>
<td>#o2000</td>
<td>set group id when executing</td>
</tr>
<tr>
<td>owner-read</td>
<td>#o0400</td>
<td>read by owner</td>
</tr>
<tr>
<td>owner-write</td>
<td>#o0200</td>
<td>write by owner</td>
</tr>
<tr>
<td>owner-exec</td>
<td>#o0100</td>
<td>execute (or search) by owner</td>
</tr>
<tr>
<td>group-read</td>
<td>#o0040</td>
<td>read by group</td>
</tr>
<tr>
<td>group-write</td>
<td>#o0020</td>
<td>write by group</td>
</tr>
<tr>
<td>group-exec</td>
<td>#o0010</td>
<td>execute (or search) by group</td>
</tr>
<tr>
<td>other-read</td>
<td>#o004</td>
<td>read by others</td>
</tr>
<tr>
<td>other-write</td>
<td>#o002</td>
<td>write by others</td>
</tr>
<tr>
<td>other-exec</td>
<td>#o001</td>
<td>execute (or search) by others</td>
</tr>
<tr>
<td>read</td>
<td>#o0444</td>
<td>read by anyone</td>
</tr>
<tr>
<td>write</td>
<td>#o0222</td>
<td>write by anyone</td>
</tr>
<tr>
<td>exec</td>
<td>#o0111</td>
<td>execute by anyone</td>
</tr>
<tr>
<td>all</td>
<td>#o0777</td>
<td>anything by anyone</td>
</tr>
</tbody>
</table>

Names for sets of permissions

<table>
<thead>
<tr>
<th>Name</th>
<th>Bit mask</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>#o0700</td>
<td>read, write, and execute by owner</td>
</tr>
<tr>
<td>group</td>
<td>#o0070</td>
<td>read, write, and execute by group</td>
</tr>
<tr>
<td>other</td>
<td>#o007</td>
<td>read, write, and execute by others</td>
</tr>
<tr>
<td>read</td>
<td>#o0444</td>
<td>read by anyone</td>
</tr>
<tr>
<td>write</td>
<td>#o0222</td>
<td>write by anyone</td>
</tr>
<tr>
<td>exec</td>
<td>#o0111</td>
<td>execute by anyone</td>
</tr>
<tr>
<td>all</td>
<td>#o0777</td>
<td>anything by anyone</td>
</tr>
</tbody>
</table>
8.7 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 times, with make-time’s using its argument while current-time’s has the current time. Time? is a predicate that recognizes times 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) → string

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

"Wed Jun 30 21:49:08 1993"

8.8 I/O

These procedures are in structures posix-i/o and posix.

- (open-pipe) → input-port + output-port

Open-pipe creates a new pipe and returns the two ends as an input port and an output port.

A file descriptor port (or fd-port) is a port that reads to or writes from an OS file descriptor. Fd-ports are returned by open-input-file, open-output-file, open-file, open-pipe, and other procedures.

- (fd-port? port) → boolean
- (port->fd port) → integer or #f

Fd-port? returns true if its argument is an fd-port. Port->fd returns the file descriptor associated with or #f if port is not an fd-port.

- (remap-file-descriptors fd-spec ...)
(remap-file-descriptors (current-output-port) 
  #f 
  (current-input-port))

moves the current output port to file descriptor 0 and the current input port to file descriptor 2.

- (dup fd-port) → fd-port
- (dup-switching-mode fd-port) → fd-port
- (dup2 fd-port file-descriptor) → fd-port

These change fd-port’s file descriptor and return a new port that uses ports’s old file descriptor. Dup uses the lowest unused file descriptor and dup2 uses the one provided. Dup-switching-mode is the same as dup except that the returned port is an input port if the argument was an output port and vice versa. If any existing port uses the file descriptor passed to dup2, that port is closed.

- (close-all-but port ...)

Close-all-but closes all file descriptors whose associated ports are not passed to it as arguments.

- (close-on-exec? port) → boolean
- (set-close-on-exec?! port boolean)

Close-on-exec? returns true if port will be closed when a new program is exec’ed. Set-close-on-exec?! sets port’s close-on-exec flag.

- (i/o-flags port) → file-options
- (set-i/o-flags! port file-options)

These two procedures read and write various options for port. The options that can be read are append, nonblocking, read-only, write-only, and read/write. Only the append and nonblocking can be written.

- (port-is-a-terminal? port) → boolean
- (port-terminal-name port) → string

Port-is-a-terminal? returns true if port has an underlying file descriptor that is associated with a terminal. For such ports port-terminal-name returns the name of the terminal, for all others it returns #f.

8.9 Regular expressions

The procedures in this section provide access to POSIX regular expression matching. The regular expression syntax and semantics are far too complex to be described here. 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 5.20.

- (make-regexp string . regexp-options) → regexp
• (regexp-option option-name) → regexp-option

Make-regexp makes a new regular expression, using string as the pattern. The possible option names are:

- extended: use the extended patterns
- ignore-case: ignore case when matching
- submatches: report submatches
- newline: treat newlines specially

The regular expression is not compiled until it matched against a string, so any errors in the pattern string will not be reported until that point.

• (regexp? x) → boolean

This is a predicate for regular expressions.

• (regexp-match regexp string start submatches? starts-line? ends-line?) → boolean or list of matches
• (match? x) → boolean
• (match-start match) → integer
• (match-end match) → integer

Regexp-match matches the regular expression against the characters in string, starting at position start. If the string does not match the regular expression, regexp-match returns #f. If the string does match, then a list of match records is returned if submatches? is true, or #t is returned if it is not. Each match record contains the index of the character at the beginning of the match and one more than the index of the character at the end. The first match record gives the location of the substring that matched regexp. If the pattern in regexp contained submatches, then the results of these are returned in order, with a match records reporting submatches that succeeded and #f in place of those that did not.

Starts-line? should be true if string starts at the beginning of a line and ends-line? should be true if it ends one.

### 8.10 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.

<table>
<thead>
<tr>
<th>C procedure</th>
<th>Scheme procedure(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>accessible?</td>
</tr>
<tr>
<td>chdir</td>
<td>set-working-directory!</td>
</tr>
<tr>
<td>close</td>
<td>close-input-port, close-output-port, close-channel, close-socket</td>
</tr>
<tr>
<td>closedir</td>
<td>close-directory-stream</td>
</tr>
<tr>
<td>creat</td>
<td>open-file</td>
</tr>
<tr>
<td>ctime</td>
<td>time-&gt;string</td>
</tr>
<tr>
<td>dup</td>
<td>dup, dup-switching-mode</td>
</tr>
<tr>
<td>dup2</td>
<td>dup2</td>
</tr>
<tr>
<td>C procedure</td>
<td>Scheme procedure(s)</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>_exit</td>
<td>exit</td>
</tr>
<tr>
<td>fcntl</td>
<td>io-flags, set-io-flags!, close-on-exec, set-close-on-exec!</td>
</tr>
<tr>
<td>fork</td>
<td>fork, fork-and-forget</td>
</tr>
<tr>
<td>fstat</td>
<td>get-port-info</td>
</tr>
<tr>
<td>getcwd</td>
<td>working-directory</td>
</tr>
<tr>
<td>getegid</td>
<td>get-effective-group-id</td>
</tr>
<tr>
<td>getenv</td>
<td>lookup-environment-variable, environment-alist</td>
</tr>
<tr>
<td>geteuid</td>
<td>get-effective-user-id</td>
</tr>
<tr>
<td>getgid</td>
<td>get-group-id</td>
</tr>
<tr>
<td>getgroups</td>
<td>get-groups</td>
</tr>
<tr>
<td>getlogin</td>
<td>get-login-name</td>
</tr>
<tr>
<td>getpid</td>
<td>get-process-id</td>
</tr>
<tr>
<td>getppid</td>
<td>get-parent-process-id</td>
</tr>
<tr>
<td>getuid</td>
<td>get-user-id</td>
</tr>
<tr>
<td>isatty</td>
<td>port-is-a-terminal?</td>
</tr>
<tr>
<td>link</td>
<td>link</td>
</tr>
<tr>
<td>lstat</td>
<td>get-file/link-info</td>
</tr>
<tr>
<td>mkdir</td>
<td>make-directory</td>
</tr>
<tr>
<td>mkfifo</td>
<td>make-fifo</td>
</tr>
<tr>
<td>open</td>
<td>open-file</td>
</tr>
<tr>
<td>opendir</td>
<td>open-directory-stream</td>
</tr>
<tr>
<td>pipe</td>
<td>open-pipe</td>
</tr>
<tr>
<td>read</td>
<td>read-char, read-block</td>
</tr>
<tr>
<td>readdir</td>
<td>read-directory-stream</td>
</tr>
<tr>
<td>rename</td>
<td>rename</td>
</tr>
<tr>
<td>rmdir</td>
<td>remove-directory</td>
</tr>
<tr>
<td>setgid</td>
<td>set-group-id!</td>
</tr>
<tr>
<td>setuid</td>
<td>set-user-id!</td>
</tr>
<tr>
<td>stat</td>
<td>get-file-info</td>
</tr>
<tr>
<td>time</td>
<td>current-time</td>
</tr>
<tr>
<td>ttyname</td>
<td>port-terminal-name</td>
</tr>
<tr>
<td>umask</td>
<td>set-file-creation-mask!</td>
</tr>
<tr>
<td>uname</td>
<td>os-name, os-node-name, os-release-name, os-version-name, machine-name</td>
</tr>
<tr>
<td>unlink</td>
<td>unlink</td>
</tr>
<tr>
<td>waitpid</td>
<td>wait-for-child-process</td>
</tr>
<tr>
<td>write</td>
<td>write-char, write-block</td>
</tr>
</tbody>
</table>
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 5.3) give access to the ASCII encoding.

<table>
<thead>
<tr>
<th>n_{10}</th>
<th>n_{16}</th>
<th>Unicode name</th>
<th>n_{10}</th>
<th>n_{16}</th>
<th>Unicode name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>NUL null</td>
<td>64</td>
<td>40</td>
<td>@ commercial at</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>SOH start of heading</td>
<td>65</td>
<td>41</td>
<td>A latin capital letter a</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>STX start of text</td>
<td>66</td>
<td>42</td>
<td>B latin capital letter b</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>ETX end of text</td>
<td>67</td>
<td>43</td>
<td>C latin capital letter c</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>EOT end of transmission</td>
<td>68</td>
<td>44</td>
<td>D latin capital letter d</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>ENQ enquiry</td>
<td>69</td>
<td>45</td>
<td>E latin capital letter e</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>ACK acknowledge</td>
<td>70</td>
<td>46</td>
<td>F latin capital letter f</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>BEL bell</td>
<td>71</td>
<td>47</td>
<td>G latin capital letter g</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>BS backspace</td>
<td>72</td>
<td>48</td>
<td>H latin capital letter h</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>HT horizontal tabulation</td>
<td>73</td>
<td>49</td>
<td>I latin capital letter i</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>LF line feed</td>
<td>74</td>
<td>4A</td>
<td>J latin capital letter j</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>VT vertical tabulation</td>
<td>75</td>
<td>4B</td>
<td>K latin capital letter k</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>FF form feed</td>
<td>76</td>
<td>4C</td>
<td>L latin capital letter l</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>CR carriage return</td>
<td>77</td>
<td>4D</td>
<td>M latin capital letter m</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>SO shift out</td>
<td>78</td>
<td>4E</td>
<td>N latin capital letter n</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>SI shift in</td>
<td>79</td>
<td>4F</td>
<td>O latin capital letter o</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>DLE data link escape</td>
<td>80</td>
<td>50</td>
<td>P latin capital letter p</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>DC1 device control one</td>
<td>81</td>
<td>51</td>
<td>Q latin capital letter q</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>DC2 device control two</td>
<td>82</td>
<td>52</td>
<td>R latin capital letter r</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>DC3 device control three</td>
<td>83</td>
<td>53</td>
<td>S latin capital letter s</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>DC4 device control four</td>
<td>84</td>
<td>54</td>
<td>T latin capital letter t</td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>NAK negative acknowledge</td>
<td>85</td>
<td>55</td>
<td>U latin capital letter u</td>
</tr>
<tr>
<td>22</td>
<td></td>
<td>SYN synchronous idle</td>
<td>86</td>
<td>56</td>
<td>V latin capital letter v</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td>ETB end of transmission block</td>
<td>87</td>
<td>57</td>
<td>W latin capital letter w</td>
</tr>
<tr>
<td>24</td>
<td></td>
<td>CAN cancel</td>
<td>88</td>
<td>58</td>
<td>X latin capital letter x</td>
</tr>
<tr>
<td>25</td>
<td></td>
<td>EM end of medium</td>
<td>89</td>
<td>59</td>
<td>Y latin capital letter y</td>
</tr>
<tr>
<td>26</td>
<td></td>
<td>SUB substitute</td>
<td>90</td>
<td>5A</td>
<td>Z latin capital letter z</td>
</tr>
<tr>
<td>27</td>
<td></td>
<td>ESC escape</td>
<td>91</td>
<td>5B</td>
<td>[ left square bracket</td>
</tr>
<tr>
<td>28</td>
<td></td>
<td>FS file separator</td>
<td>92</td>
<td>5C</td>
<td>\ reverse solidus</td>
</tr>
<tr>
<td>29</td>
<td></td>
<td>GS group separator</td>
<td>93</td>
<td>5D</td>
<td>] right square bracket</td>
</tr>
<tr>
<td>( n_{10} )</td>
<td>( n_{16} )</td>
<td>Unicode name</td>
<td>( n_{10} )</td>
<td>( n_{16} )</td>
<td>Unicode name</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>RS record separator</td>
<td>94</td>
<td>5E</td>
<td>~ circumflex accent</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>US unit separator</td>
<td>95</td>
<td>5F</td>
<td>_ low line</td>
</tr>
<tr>
<td>32</td>
<td>20</td>
<td>SPACE space</td>
<td>96</td>
<td>60</td>
<td>' grave accent</td>
</tr>
<tr>
<td>33</td>
<td>21</td>
<td>! exclamation mark</td>
<td>97</td>
<td>61</td>
<td>a latin small letter a</td>
</tr>
<tr>
<td>34</td>
<td>22</td>
<td>&quot; quotation mark</td>
<td>98</td>
<td>62</td>
<td>b latin small letter b</td>
</tr>
<tr>
<td>35</td>
<td>23</td>
<td># number sign</td>
<td>99</td>
<td>63</td>
<td>c latin small letter c</td>
</tr>
<tr>
<td>36</td>
<td>24</td>
<td>$ dollar sign</td>
<td>100</td>
<td>64</td>
<td>d latin small letter d</td>
</tr>
<tr>
<td>37</td>
<td>25</td>
<td>% percent sign</td>
<td>101</td>
<td>65</td>
<td>e latin small letter e</td>
</tr>
<tr>
<td>38</td>
<td>26</td>
<td>&amp; ampersand</td>
<td>102</td>
<td>66</td>
<td>f latin small letter f</td>
</tr>
<tr>
<td>39</td>
<td>27</td>
<td>' apostrophe</td>
<td>103</td>
<td>67</td>
<td>g latin small letter g</td>
</tr>
<tr>
<td>40</td>
<td>28</td>
<td>( left parenthesis</td>
<td>104</td>
<td>68</td>
<td>h latin small letter h</td>
</tr>
<tr>
<td>41</td>
<td>29</td>
<td>) right parenthesis</td>
<td>105</td>
<td>69</td>
<td>i latin small letter i</td>
</tr>
<tr>
<td>42</td>
<td>2A</td>
<td>* asterisk</td>
<td>106</td>
<td>6A</td>
<td>j latin small letter j</td>
</tr>
<tr>
<td>43</td>
<td>2B</td>
<td>+ plus sign</td>
<td>107</td>
<td>6B</td>
<td>k latin small letter k</td>
</tr>
<tr>
<td>44</td>
<td>2C</td>
<td>, comma</td>
<td>108</td>
<td>6C</td>
<td>l latin small letter l</td>
</tr>
<tr>
<td>45</td>
<td>2D</td>
<td>- hyphen-minus</td>
<td>109</td>
<td>6D</td>
<td>m latin small letter m</td>
</tr>
<tr>
<td>46</td>
<td>2E</td>
<td>. full stop</td>
<td>110</td>
<td>6E</td>
<td>n latin small letter n</td>
</tr>
<tr>
<td>47</td>
<td>2F</td>
<td>/ solidus</td>
<td>111</td>
<td>6F</td>
<td>o latin small letter o</td>
</tr>
<tr>
<td>48</td>
<td>30</td>
<td>0 digit zero</td>
<td>112</td>
<td>70</td>
<td>p latin small letter p</td>
</tr>
<tr>
<td>49</td>
<td>31</td>
<td>1 digit one</td>
<td>113</td>
<td>71</td>
<td>q latin small letter q</td>
</tr>
<tr>
<td>50</td>
<td>32</td>
<td>2 digit two</td>
<td>114</td>
<td>72</td>
<td>r latin small letter r</td>
</tr>
<tr>
<td>51</td>
<td>33</td>
<td>3 digit three</td>
<td>115</td>
<td>73</td>
<td>s latin small letter s</td>
</tr>
<tr>
<td>52</td>
<td>34</td>
<td>4 digit four</td>
<td>116</td>
<td>74</td>
<td>t latin small letter t</td>
</tr>
<tr>
<td>53</td>
<td>35</td>
<td>5 digit five</td>
<td>117</td>
<td>75</td>
<td>u latin small letter u</td>
</tr>
<tr>
<td>54</td>
<td>36</td>
<td>6 digit six</td>
<td>118</td>
<td>76</td>
<td>v latin small letter v</td>
</tr>
<tr>
<td>55</td>
<td>37</td>
<td>7 digit seven</td>
<td>119</td>
<td>77</td>
<td>w latin small letter w</td>
</tr>
<tr>
<td>56</td>
<td>38</td>
<td>8 digit eight</td>
<td>120</td>
<td>78</td>
<td>x latin small letter x</td>
</tr>
<tr>
<td>57</td>
<td>39</td>
<td>9 digit nine</td>
<td>121</td>
<td>79</td>
<td>y latin small letter y</td>
</tr>
<tr>
<td>58</td>
<td>3A</td>
<td>: colon</td>
<td>122</td>
<td>7A</td>
<td>z latin small letter z</td>
</tr>
<tr>
<td>59</td>
<td>3B</td>
<td>; semicolon</td>
<td>123</td>
<td>7B</td>
<td>{ left curly bracket</td>
</tr>
<tr>
<td>60</td>
<td>3C</td>
<td>&lt; less-than sign</td>
<td>124</td>
<td>7C</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>3D</td>
<td>= equals sign</td>
<td>125</td>
<td>7D</td>
<td>} right curly bracket</td>
</tr>
<tr>
<td>62</td>
<td>3E</td>
<td>&gt; greater-than sign</td>
<td>126</td>
<td>7E</td>
<td>~ tilde</td>
</tr>
<tr>
<td>63</td>
<td>3F</td>
<td>? question mark</td>
<td>127</td>
<td>7F</td>
<td>DEL delete</td>
</tr>
</tbody>
</table>
Bibliography


Index

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

accessible? 95
add-signal-queue-signal! 91
any 29
any-match? 62
any? 29
arithmetic-shift 31
array 33
array->vector 33
array-dimensions 33
array-ref 33
array-set! 33
array? 33
ascii->char 31
ascii-limit 31
ascii-range 60
ascii-ranges 60
ascii-whitespaces 31
atom? 29
atomically! 69

bit-count 31
bitwise-and 31
bitwise-ior 31
bitwise-not 31
bitwise-xor 31
byte-vector 31
byte-vector-length 31
byte-vector-ref 31
byte-vector-set! 31
byte-vector? 31
call-atomically 69
call-atomically! 69
call-ensuring-atomicity 68
call-ensuring-atomicity! 68
call-external 78;77
call-external-value 77
call-imported-binding 77
cell-ref 32
cell-set! 32
cell? 32
char->ascii 31
close-all-but 99
close-directory-stream 94
close-on-exec? 99
close-socket 43
compound-interface 21
concatenate-symbol 30
condvar-has-value? 70
condvar-value 70
condvar? 70
copy-array 33
current-column 41
current-proposal 70
current-row 41
current-thread 66
current-time 98
debug-message 67
default-hash-function 40
define-exported-binding 75
define-imported-binding 76
define-interface 20
define-record-discloser 36;35
define-record-resumer 83;36
define-structure 18
delete 30
delete-from-queue! 32
delq 30
delq! 30
dequeue! 32
dequeue-signal! 91
directory-stream? 93
dup 99
let-fluids 41
limit-output 41
link 95
list->queue 32
list-delete-neighbor-dups 57; 53
list-delete-neighbor-dups! 57
list-directory 94
list-merge 56; 53
list-merge! 56; 53
list-merge-sort 56
list-merge-sort! 56
list-sort 53
list-sort! 53
list-sorted? 55; 53
list-stable-sort 53
list-stable-sort! 53
lock? 71
lookup-all-externals 78
lookup-environment-variable 92
lookup-exported-binding 76
lookup-external 78
lookup-imported-binding 75
machine-name 93
make-array 33
make-byte-vector 31
make-cell 32
make-directory 70
make-fifo 95
make-fluid 41
make-integer-table 40
make-lock 71
make-placeholder 72
make-proposal 70
make-queue 32
make-record 35
make-record-type 36
make-regexp 99
make-shared-array 33
make-signal-queue 91
make-sparse-vector 32
make-string-input-port 40
make-string-output-port 40
make-string-table 40
make-symbol-table 39
make-table 39
make-table-immutable! 40
make-table-maker 40
make-time 98
make-tracking-input-port 41
make-tracking-output-port 41
match 62
match-end 100; 62
match-start 100; 62
match-submatches 62
match? 100
maybe-commit 70
maybe-commit-and-block 72
maybe-commit-and-block-on-queue 72
maybe-commit-and-make-ready 72
maybe-commit-and-set-condvar! 70
maybe-commit-and-wait-for-condvar 70
maybe-dequeue-signal! 91
maybe-dequeue-thread! 72
maybe-obtain-lock 71
memq? 29
modify 18
n= 29
name->group-info 93
name->signal 89
name->user-info 93
negate 60
neq? 29
no-op 29
no-submatches 62
null-list? 29
obtain-lock 71
one-of 61
open-directory-stream 93
open-file 94
open-pipe 98
open-socket 43
os-name 93
os-node-name 93
os-release-name 93
os-version-name 93
p 30
partition-list 30
partition-list! 30
placeholder-set! 72
placeholder-value 72
placeholder? 72
port->fd 98
port-is-a-terminal? 99
port-terminal-name 99
prefix 18
pretty-print 30
process-id->integer 88
process-id-exit-status 88
process-id-terminating-signal 88
process-id=? 88
process-id? 88
provisional-byte-vector-ref 68
provisional-byte-vector-set! 68
provisional-car 68
provisional-cdr 68
provisional-cell-ref 68
provisional-cell-set! 68
provisional-set-car! 68
provisional-set-cdr! 68
provisional-string-ref 68
provisional-string-set! 68
provisional-vector-ref 68
provisional-vector-set! 68
queue->list 32
queue-empty? 32
queue-length 32
queue? 32
range 60
ranges 60
read-directory-stream 93
record 35
record-accessor 36
record-constructor 36
record-length 35
record-modifier 36
record-predicate 36
record-ref 35
record-set! 35
record-type? 36
record? 35
regexp-match 100
regexp? 100
release-lock 71
relinquish-timeslice 66
remap-file-descriptors 98
remove-directory 95
remove-duplicates 30
remove-signal-queue-signal! 91
rename 95
repeat 61
reverse! 30
sequence 61
set 60
set-close-on-exec?! 99
set-condvar-has-value?! 70
set-condvar-value! 70
set-current-proposal! 70
set-file-creation-mask! 95
set-group-id! 92
set-i/o-flags! 99
set-user-id! 92
set-working-directory! 94
shared-binding-is-import? 76
shared-binding-name 76
shared-binding-ref 75
shared-binding-set! 76
shared-binding? 76
signal-name 89
signal-os-number 89
signal-process 91
signal-queue-monitored-signals 91
signal-queue? 91
signal=? 89
signal? 89
sleep 66
socket-accept 43
socket-client 43
socket-port-number 43
sparse-vector->list 32
sparse-vector-ref 32
sparse-vector-set! 32
spawn 66
string-end 61
string-hash 40
string-output-port-output 40
string-start 61
submatch 62
subset 18
subtract 60
system 42
table-ref 40
table-set! 40
table-walk 40
table? 40
terminate-current-thread 66
terminate-thread! 67
text 61
thread-name 66
thread-queue-empty? 72
thread-uid 66
thread? 66
time->string 98
time-seconds 98
time<=? 98
time<? 98
time=? 98
time>=? 98
time>? 98
time? 98

undefine-exported-binding 76
undefine-imported-binding 76
union 60
unlink 95
use-case 61
user-id->integer 92
user-id->user-info 92
user-id=? 92
user-id? 92
user-info-group 93
user-info-home-directory 93
user-info-id 93
user-info-name 93
user-info-shell 93
user-info? 93

vector-binary-search 58
vector-binary-search3 58
vector-delete-neighbor-dups! 57
vector-heap-sort 57
vector-heap-sort! 57
vector-insert-sort 57
vector-insert-sort! 57
vector-merge 56; 53
vector-merge! 56; 53
vector-merge-sort 56
vector-merge-sort! 56
vector-sort 53
vector-sort! 53
vector-sorted? 55; 53
vector-stable-sort 53
vector-stable-sort! 53

wait-for-child-process 88
working-directory 94