Scsh Reference Manual

For Scsh release 0.3
December 25, 1994

Olin Shivers and Brian D. Carlstrom

Acknowledgements

Who should I thank? My so-called “colleagues,” who laugh at me behind my back, all the while becoming famous on *my* work? My worthless graduate students, whose computer skills appear to be limited to downloading bitmaps off of netnews? My parents, who are still waiting for me to quit “fooling around with computers,” go to med school, and become a radiologist? My department chairman, a manager who gives one new insight into and sympathy for disgruntled postal workers?

My God, no one could blame me—no one!—if I went off the edge and just lost it completely one day. I couldn’t get through the day as it is without the Prozac and Jack Daniels I keep on the shelf, behind my Tops-20 JSYS manuals. I start getting the shakes real bad around 10am, right before my advisor meetings. A 10 oz. Jack ’n Zac helps me get through the meetings without one of my students winding up with his severed head in a bowling-ball bag. They look at me funny; they think I twitch a lot. I’m not twitching. I’m controlling my impulse to snag my 9mm Sig-Sauer out from my day-pack and make a few strong points about the quality of undergraduate education in Amerika.

If I thought anyone cared, if I thought anyone would even be reading this, I’d probably make an effort to keep up appearances until the last possible moment. But no one does, and no one will. So I can pretty much say exactly what I think.

Oh, yes, the *acknowledgements.* I think not. I did it. I did it all, by myself.

Olin Shivers
Cambridge
September 4, 1994

Contents

Chapter 1

Introduction

This is a draft manual for scsh, a Unix shell that is embedded within Scheme. Scsh comes built on top of Scheme 48, and it has two components: a process notation for running programs and setting up pipelines and redirections, and a complete syscall library for low-level access to the OS. This manual gives a complete description of scsh. A general discussion of the design principles behind scsh can be found in a companion paper, "A Scheme Shell."

1.1 Caveats

It is important to note what scsh is *not*, as well as what it is. Scsh, in the current release, is primarily designed for the writing of shell scripts—programming. It is not a very comfortable system for interactive command use: the current release lacks job control, command-line editing, a terse, convenient command syntax, and it can not be made to read in an initialisation file analogous to `.login` or `.profile`. We hope to address all of these problems in future releases; we even have designs for several of these features; but the system as-released does not currently address these issues.

As a first release, the system has some rough edges. It is quite slow to start up; we hope to fix that by providing a static-heap linker in the next release. For now, the initial image load takes about a cpu second.

This manual is very, very rough: incomplete, inconsistent, and misleading. At some point, we will polish it up, finish it off, and re-typeset it using markup, so we can generate html, info nodes, and TeX output from the single source without having to deal with Texinfo. But it's all there is, for now.

1.2 Naming conventions

Scsh follows a general naming scheme that consistently employs a set of abbreviations. This is intended to make it easier to remember the names of things. Some of the common ones are:

`fdes` Means "file descriptor," a small integer used in Unix to represent I/O channels.

`...*` A given bit of functionality sometimes comes in two related forms, the first being a *special form* that contains a body of Scheme code to be executed in some context, and the other being a *procedure* that takes a procedural argument (a "thunk") to be called in the same context. The procedure variant is named by taking the name of the special form, and appending an asterisk. For example:

```
;;; Special form:
(with-cwd "/etc"
  (for-each print-file (directory-files))
  (display "All done"))

;;; Procedure:
(with-cwd* "/etc"
  (lambda ()
    (for-each print-file (directory-files))
    (display "All done")))
```

action/*modifier* The infix "/" is pronounced "with," as in `exec/env`—"exec with environment."

`call/...` Procedures that call their argument on some computed value are usually named "`call/...`," *e.g.*, (`call/fdes` *port proc*), which calls *proc* on *port*'s file descriptor, returning whatever *proc* returns. The abbreviated name means "call with file descriptor."

`with-...` Procedures that call their argument, and special forms that execute their bodies in some special dynamic context frequently have names of the form `with-...`. For example, (`with-env` *env* $body_1$ ...) and (`with-env*` *env thunk*). These forms set the process environment body, execute their body or thunk, and then return after resetting the environment to its original state.

`create-` Procedures that create objects in the file system (files, directories, temp files, fifos, etc), begin with `create-...`.

`delete-` Procedures that delete objects from the file system (files, directories, temp files, fifos, etc), begin with `delete-`....

record`:`*field* Procedures that access fields of a record are usually written with a colon between the name of the record and the name of the field, as in `user-info:home-dir`.

`%`... A percent sign is used to prefix lower-level scsh primitives that are not commonly used.

`-info` Data structures packaging up information about various OS entities frequently end in ...`-info`. Examples: `user-info`, `file-info`, `group-info`, and `host-info`.

Enumerated constants from some set *s* are usually named *s*/$const_1$, *s*/$const_2$, ... For example, the various Unix signal integers have the names `signal/cont`, `signal/kill`, `signal/int`, `signal/hup`, and so forth.

1.3 Lexical issues

Scsh's lexical syntax is just R4RS Scheme, with the following exceptions.

Scsh differs from R4RS Scheme in the following ways:

- In scsh, symbol case is preserved by `read` and is significant on symbol comparison. This means

  ```
  (run (less Readme))
  ```

  displays the right file.

- "`-`" and "`+`" are allowed to begin symbols. So the following are legitimate symbols:

  ```
  -O2 -geometry +Wn
  ```

Scsh also extends R4RS lexical syntax in the following ways:

- "`|`" and "`.`" are symbol constituents. This allows `|` for the pipe symbol, and `..` for the parent-directory symbol. (Of course, "`.`" alone is not a symbol, but a dotted-pair marker.)

- A symbol may begin with a digit. So the following are legitimate symbols:

  ```
  9x15 80x36-3+440
  ```

- Strings are allowed to contain the ANSI C escape sequences such as `\n` and `\161`.

- `#!` is a comment read-macro similar to `;`. This is used to write shell scripts. When the reader encounters `#!`, it skips characters until it finds the sequence newline/exclamation-point/sharp-sign/newline.

It is unfortunate that the single-dot token, "`.`", is both a fundamental Unix file name and a deep, primitive syntactic token in Scheme—it means the following will not parse correctly in scsh:

```
(run/strings (find . -name *.c -print))
```

You must instead quote the dot:

```
(run/strings (find "." -name *.c -print))
```

1.4 A word about Unix standards

"The wonderful thing about Unix standards is that there are so many to choose from." You may be totally bewildered about the multitude of various standards that exist. Rest assured that this nowhere in this manual will you encounter an attempt to spell it all out for you; you could not read and internalise such a twisted account without bleeding from the nose and ears.

However, you might keep in mind the following simple fact: of all the standards, POSIX, as far as I have been able to determine, is the least common denominator. So when this manual repeatedly refers to POSIX, the point is "the thing we are describing should be portable just about anywhere." Scsh sticks to POSIX when at all possible; it's major departure is symbolic links, which aren't in POSIX (see—it really *is* a least common denominator).

However, just because POSIX is the l.c.d. standard doesn't mean everyone supports all of it. The guerilla PC Unix implementations that have been springing up on the net (*e.g.*, NetBSD, Linux, FreeBSD, and so forth) are only recently coming into compliance with the standard—although they are getting there. We've found a few small problems with NeXTSTEP's POSIX support that we had to work around.

Chapter 2

Process notation

Scsh has a notation for controlling Unix processes that takes the form of s-expressions; this notation can then be embedded inside of standard Scheme code. The basic elements of this notation are *process forms*, *extended process forms*, and *redirections*.

2.1 Extended process forms and i/o redirections

An *extended process form* is a specification of a Unix process to run, in a particular I/O environment:

epf ::= (*pf* *redir*$_1$ ... *redir*$_n$)

where *pf* is a process form and the *redir*$_i$ are redirection specs. A *redirection spec* is one of:

(< *[fdes] file-name*)	`; Open file for read.`
(> *[fdes] file-name*)	`; Open file create/truncate.`
(<< *[fdes] object*)	`; Use` *object*`'s printed rep.`
(>> *[fdes] file-name*)	`; Open file for append.`
(= *fdes fdes/port*)	`; Dup2`
(- *fdes/port*)	`; Close` *fdes/port*.
`stdports`	`; 0,1,2 dup'd from standard ports.`

The input redirections default to file descriptor 0; the output redirections default to file descriptor 1.

The subforms of a redirection are implicitly backquoted, and symbols stand for their print-names. So `(> ,x)` means "output to the file named by Scheme variable `x`," and `(< /usr/shivers/.login)` means "read from `/usr/shivers/.login`."

Here are two more examples of i/o redirection:

```
(< ,(vector-ref fv i))
(>> 2 /tmp/buf)
```

These two redirections cause the file `fv[i]` to be opened on stdin, and `/tmp/buf` to be opened for append writes on stderr.

The redirection `(<<` *object*`)` causes input to come from the printed representation of *object*. For example,

```
(<< "The quick brown fox jumped over the lazy dog.")
```

causes reads from stdin to produce the characters of the above string. The object is converted to its printed representation using the `display` procedure, so

```
(<< (A five element list))
```

is the same as

```
(<< "(A five element list)")
```

is the same as

```
(<< ,(reverse '(list element five A))).
```

(Here we use the implicit backquoting feature to compute the list to be printed.)

The redirection `(=` *fdes fdes/port*`)` causes *fdes/port* to be dup'd into file descriptor *fdes*. For example, the redirection

```
(= 2 1)
```

causes stderr to be the same as stdout. *fdes/port* can also be a port, for example:

```
(= 2 ,(current-output-port))
```

causes stderr to be dup'd from the current output port. In this case, it is an error if the port is not a file port (*e.g.*, a string port). More complex redirections can be accomplished using the `begin` process form, discussed below, which gives the programmer full control of i/o redirection from Scheme.

2.1.1 Port and file descriptor sync

It's important to remember that rebinding Scheme's current I/O ports (*e.g.*, using `call-with-input-file` to rebind the value of `(current-input-port)`) does *not* automatically "rebind" the file referenced by the Unix stdio file descriptors 0, 1, and 2. This is impossible to do in general, since some Scheme ports are not representable as Unix file descriptors. For example, many Scheme implementations provide "string ports," that is, ports that collect characters sent to them into memory buffers. The accumulated string can later be retrieved from the port as a string. If a user were to bind `(current-output-port)` to such a port, it would be impossible to associate file descriptor 1 with this port, as it cannot be represented in Unix. So, if the user subsequently forked off some other program as a subprocess, that program would of course not see the Scheme string port as its standard output.

To keep stdio synced with the values of Scheme's current i/o ports, use the special redirection `stdports`. This causes 0, 1, 2 to be redirected from the current Scheme standard ports. It is equivalent to the three redirections:

```
(= 0 ,(current-input-port))
(= 1 ,(current-output-port))
(= 2 ,(error-output-port))
```

The redirections are done in the indicated order. This will cause an error if the one of current i/o ports isn't a Unix port (*e.g.*, if one is a string port). This Scheme/Unix i/o synchronisation can also be had in Scheme code (as opposed to a redirection spec) with the `(stdports->stdio)` procedure.

2.2 Process forms

A *process form* specifies a computation to perform as an independent Unix process. It can be one of the following:

```
(begin . scheme-code)           ; Run scheme-code in a fork.
(| pf1 ... pfn)                 ; Simple pipeline
(|+ connect-list pf1 ... pfn)   ; Complex pipeline
(epf . epf)                     ; An extended process form.
(prog arg1 ... argn)            ; Default: exec the program.
```

The default case (*prog* arg_1 ... arg_n) is also implicitly backquoted. That is, it is equivalent to:

```
(begin (apply exec-path `(prog arg1 ... argn)))
```

`Exec-path` is the version of the `exec()` system call that uses scsh's path list to search for an executable. The program and the arguments must be either strings, symbols, or integers. Symbols and integers are coerced to strings. A symbol's print-name is used. Integers are converted to strings in base 10. Using symbols instead of strings is convenient, since it suppresses the clutter of the surrounding `"..."` quotation marks. To aid this purpose, scsh reads symbols in a case-sensitive manner, so that you can say

```
(more Readme)
```

and get the right file.

A *connect-list* is a specification of how two processes are to be wired together by pipes. It has the form `((`$from_1$ $from_2$ ... *to*`) ...)` and is implicitly backquoted. For example,

```
(|+ ((1 2 0) (3 1)) pf₁ pf₂)
```

runs pf_1 and pf_2. The first clause `(1 2 0)` causes pf_1's stdout (1) and stderr (2) to be connected via pipe to pf_2's stdin (0). The second clause `(3 1)` causes pf_1's file descriptor 3 to be connected to pf_2's file descriptor 1.

Note that R4RS does not specify whether or not `|` and `|+` are readable symbols. Scsh does.

2.3 Using extended process forms in Scheme

Process forms and extended process forms are *not* Scheme. They are a different notation for expressing computation that, like Scheme, is based upon s-expressions. Extended process forms are used in Scheme programs by embedding them inside special Scheme forms. There are three basic Scheme forms that use extended process forms: `exec-epf`, `&`, and `run`.

`(exec-epf` . *epf*`)` ⟶ *no return value* — syntax
`(&` . *epf*`)` ⟶ *integer* — syntax
`(run` . *epf*`)` ⟶ *integer* — syntax

> The `(exec-epf .` *epf*`)` form nukes the current process: it establishes the i/o redirections and then overlays the current process with the requested computation.
>
> The `(& .` *epf*`)` form is similar, except that the process is forked off in background. The form returns the subprocess' pid.
>
> The `(run .` *epf*`)` form runs the process in foreground: after forking off the computation, it waits for the subprocess to exit, and returns its exit status.
>
> These special forms are macros that expand into the equivalent series of system calls. The definition of the `exec-epf` macro is non-trivial, as it produces the code to handle i/o redirections and set up pipelines. However, the definitions of the `&` and `run` macros are very simple:
>
> ```
> (& . epf)     ≡  (fork (λ () (exec-epf . epf)))
> (run . epf)   ≡  (wait (& . epf))
> ```

2.3.1 Procedures and special forms

It is a general design principle in scsh that all functionality made available through special syntax is also available in a straightforward procedural form. So there are procedural equivalents for all of the process notation. In this way, the programmer is not restricted by the particular details of the syntax. Here are some of the syntax/procedure equivalents:

Notation	Procedure
`|`	`fork/pipe`
`|+`	`fork/pipe+`
`exec-epf`	`exec-path`
redirection	`open`, `dup`
`&`	`fork`
`run`	`wait` + `fork`

Having a solid procedural foundation also allows for general notational experimentation using Scheme's macros. For example, the programmer can build his own pipeline notation on top of the `fork` and `fork/pipe` procedures. Chapter 3 gives the full story on all the procedures in the syscall library.

2.3.2 Interfacing process output to Scheme

There is a family of procedures and special forms that can be used to capture the output of processes as Scheme data.

`(run/port` . *epf*`)` ⟶ *port* — syntax
`(run/file` . *epf*`)` ⟶ *string* — syntax
`(run/string` . *epf*`)` ⟶ *string* — syntax
`(run/strings` . *epf*`)` ⟶ *string list* — syntax
`(run/sexp` . *epf*`)` ⟶ *object* — syntax
`(run/sexps` . *epf*`)` ⟶ *list* — syntax

These forms all fork off subprocesses, collecting the process' output to stdout in some form or another.

`run/port`	Value is a port open on process's stdout. Returns immediately after forking child.
`run/file`	Value is name of a temp file containing process's output. Returns when process exits.
`run/string`	Value is a string containing process' output. Returns when eof read.
`run/strings`	Splits process' output into a list of newline-delimited strings. Returns when eof read.
`run/sexp`	Reads a single object from process' stdout with `read`. Returns as soon as the read completes.
`run/sexps`	Repeatedly reads objects from process' stdout with `read`. Returns accumulated list upon eof.

The delimiting newlines are not included in the strings returned by `run/strings`.

These special forms just expand into calls to the following analogous procedures.

(`run/port*` *thunk*) ⟶ *port* procedure
(`run/file*` *thunk*) ⟶ *string* procedure
(`run/string*` *thunk*) ⟶ *string* procedure
(`run/strings*` *thunk*) ⟶ *string list* procedure
(`run/sexp*` *thunk*) ⟶ *object* procedure
(`run/sexps*` *thunk*) ⟶ *object list* procedure

For example, `(run/port . ` *epf*`)` expands into

```
(run/port* (λ () (exec-epf . epf))).
```

The following procedures are also of utility for generally parsing input streams in scsh:

(`port->string` *port*) ⟶ *string* procedure
(`port->sexp-list` *port*) ⟶ *list* procedure
(`port->string-list` *port*) ⟶ *string list* procedure
(`port->list` *reader port*) ⟶ *list* procedure

`Port->string` reads the port until eof, then returns the accumulated string. `Port->sexp-list` repeatedly reads data from the port until eof, then returns the accumulated list of items. `Port->string-list` repeatedly reads newline-terminated strings from the port until eof, then returns the accumulated list of strings. The delimiting newlines are not part of the returned strings. `Port->list` generalises these two procedures. It uses *reader* to repeatedly read objects from a port. It accumulates these objects into a list, which is returned upon eof. The `port->string-list` and `port->sexp-list` procedures are trivial to define, being merely `port->list` curried with the appropriate parsers:

```
(port->string-list port) ≡ (port->list read-line port)
(port->sexp-list   port) ≡ (port->list read port)
```

The following compositions also hold:

```
run/string*  ≡  port->string      ∘ run/port*
run/strings* ≡  port->string-list ∘ run/port*
run/sexp*    ≡  read              ∘ run/port*
run/sexps*   ≡  port->sexp-list   ∘ run/port*
```

(`reduce-port` *port reader op . seeds*) ⟶ *object** procedure

This procedure can be used to perform a variety of iterative operations over an input stream. It repeatedly uses *reader* to read an object from *port*. If the first read returns eof, then the entire `reduce-port` operation returns the seeds as multiple values. If the first read operation returns some other value v, then *op* is applied to v and the seeds: `(`*op v . seeds*`)`. This should return a new set of seed values, and the reduction then loops, reading a new value from the port, and so forth. (If multiple seed values are used, then *op* must return multiple values.)

For example, `(port->list` *reader port*`)` could be defined as

```
(reverse (reduce-port port reader cons '()))
```

An imperative way to look at `reduce-port` is to say that it abstracts the idea of a loop over a stream of values read from some port, where the seed values express the loop state.

2.4 More complex process operations

The procedures and special forms in the previous section provide for the common case, where the programmer is only interested in the output of the process. These special forms and procedures provide more complicated facilities for manipulating processes.

2.4.1 Pids and ports together

`(run/port+pid . `*epf*`)` $\longrightarrow$ *[port fixnum]* syntax
`(run/port+pid* `*thunk*`)` $\longrightarrow$ *[port fixnum]* procedure

This special form and its analogous procedure can be used if the programmer also wishes access to the process' pid, exit status, or other information. They both fork off a subprocess, returning two values: a port open on the process' stdout, and the subprocess's pid.

For example, to uncompress a tech report, reading the uncompressed data into scsh, and also be able to track the exit status of the decompression process, use the following:

```
(receive (port pid) (run/port+pid (zcat tr91-145.tex.Z))
  (let* ((paper (port->string port))
         (status (wait pid)))
    ...use paper, status, and pid here...))
```

Note that you must *first* do the `port->string` and *then* do the wait—the other way around may lock up when the zcat fills up its output pipe buffer.

2.4.2 Multiple stream capture

Occasionally, the programmer may want to capture multiple distinct output streams from a process. For instance, he may wish to read the stdout and stderr streams into two distinct strings. This is accomplished with the `run/collecting` form and its analogous procedure, `run/collecting*`.

`(run/collecting` *fds . epf*`)` $\longrightarrow$ *[port. . .]* syntax
`(run/collecting*` *fds thunk*`)` $\longrightarrow$ *[port. . .]* procedure

`Run/collecting` and `run/collecting*` run processes that produce multiple output streams and return ports open on these streams. To avoid issues of deadlock, `run/collecting` doesn't use pipes. Instead, it first runs the process with output to temp files, then returns ports open on the temp files. For example,

```
(run/collecting (1 2) (ls))
```

runs `ls` with stdout (fd 1) and stderr (fd 2) redirected to temporary files. When the `ls` is done, `run/collecting` returns three values: the `ls` process' exit status, and two ports open on the temporary files. The files are deleted before `run/collecting` returns, so when the ports are closed, they vanish. The `fds` list of file descriptors is implicitly backquoted by the special-form version.

For example, if Kaiming has his mailbox protected, then

```
(receive (status out err)
         (run/collecting (1 2) (cat /usr/kmshea/mbox))
  (list status (port->string out) (port->string err)))
```

might produce the list

```
(256 "" "cat: /usr/kmshea/mbox: Permission denied")
```

What is the deadlock hazard that causes `run/collecting` to use temp files? Processes with multiple output streams can lock up if they use pipes to communicate with Scheme i/o readers. For example, suppose some Unix program `myprog` does the following:

1. First, outputs a single "`(`" to stderr.
2. Then, outputs a megabyte of data to stdout.
3. Finally, outputs a single "`)`" to stderr, and exits.

Our scsh programmer decides to run `myprog` with stdout and stderr redirected *via Unix pipes* to the ports `port1` and `port2`, respectively. He gets into trouble when he subsequently says `(read port2)`. The Scheme `read` routine reads the open paren, and then hangs in a `read()` system call trying to read a matching close paren. But before `myprog` sends the close paren down the stderr pipe, it first tries to write a megabyte of data to the stdout pipe. However, Scheme is not reading that pipe—it's stuck waiting for input on stderr. So the stdout pipe quickly fills up, and `myprog` hangs, waiting for the pipe to drain. The `myprog` child is stuck in a stdout/`port1` write; the Scheme parent is stuck in a stderr/`port2` read. Deadlock.

Here's a concrete example that does exactly the above:

```
(receive (status port1 port2)
         (run/collecting (1 2)
             (begin
               ;; Write an open paren to stderr.
               (run (echo "(") (= 1 2))
               ;; Copy a lot of stuff to stdout.
               (run (cat /usr/dict/words))
               ;; Write a close paren to stderr.
               (run (echo ")") (= 1 2))))

   ;; OK. Here, I have a port PORT1 built over a pipe
   ;; connected to the BEGIN subproc's stdout, and
   ;; PORT2 built over a pipe connected to the BEGIN
   ;; subproc's stderr.
   (read port2) ; Should return the empty list.
   (port->string port1)) ; Should return a big string.
```

In order to avoid this problem, `run/collecting` and `run/collecting*` first run the child process to completion, buffering all the output streams in temp files (using the `temp-file-channel` procedure, see below). When the child process exits, ports open on the buffered output are returned. This approach has two disadvantages over using pipes:

- The total output from the child output is temporarily written to the disk before returning from `run/collecting`. If this output is some large intermediate result, the disk could fill up.
- The child producer and Scheme consumer are serialised; there is no concurrency overlap in their execution.

However, it remains a simple solution that avoids deadlock. More sophisticated solutions can easily be programmed up as needed—`run/collecting*` itself is only 12 lines of simple code.

See `temp-file-channel` for more information on creating temp files as communication channels.

2.5 Conditional process sequencing forms

These forms allow conditional execution of a sequence of processes.

(|| $pf_1 \ldots pfn$) $\longrightarrow$ *boolean* syntax

Run each proc until one completes successfully (*i.e.*, exit status zero). Return true if some proc completes successfully; otherwise `#f`.

(&& $pf_1 \ldots pfn$) $\longrightarrow$ *boolean* syntax

Run each proc until one fails (*i.e.*, exit status non-zero). Return true if all procs complete successfully; otherwise `#f`.

2.6 Process filters

These procedures are useful for forking off processes to filter text streams.

(`char-filter` *filter*) $\longrightarrow$ *procedure* procedure

The *filter* argument is a character→character procedure. Returns a procedure that when called, repeatedly reads a character from the current input port, applies *filter* to the character, and writes the result to the current output port. The procedure returns upon reaching eof on the input port.

For example, to downcase a stream of text in a spell-checking pipeline, instead of using the Unix `tr A-Z a-z` command, we can say:

```
(run (| (delatex)
        (begin ((char-filter char-downcase))) ; tr A-Z a-z
        (spell)
        (sort)
        (uniq))
     (< scsh.tex)
     (> spell-errors.txt))
```

(`string-filter` *filter [buflen]*) $\longrightarrow$ *procedure* procedure

The *filter* argument is a string→string procedure. Returns a procedure that when called, repeatedly reads a string from the current input port, applies *filter* to the string, and writes the result to the current output port. The procedure returns upon reaching eof on the input port.

The optional *buflen* argument controls the number of characters each internal read operation requests; this means that *filter* will never be applied to a string longer than *buflen* chars. The default *buflen* value is 1024.

Chapter 3

System Calls

Scsh provides (almost) complete access to the basic Unix kernel services: processes, files, signals and so forth. These procedures comprise a first cut at a Scheme binding for POSIX, with a few extras thrown in (*e.g.*, symbolic links, `fchown`, `fstat`). A few have been punted for the current release (tty control, ioctl, and a few others.)

3.1 Errors

Scsh syscalls never return error codes, and do not use a global `errno` variable to report errors. Errors are consistently reported by raising exceptions. This frees up the procedures to return useful values, and allows the programmer to assume that *if a syscall returns, it succeeded.* This greatly simplifies the flow of the code from the programmer's point of view.

Since Scheme does not yet have a standard exception system, the scsh definition remains somewhat vague on the actual form of exceptions and exception handlers. When a standard exception system is defined, scsh will move to it. For now, scsh uses the Scheme 48 exception system, with a simple sugaring on top to hide the details in the common case.

System call error exceptions contain the Unix `errno` code reported by the system call. Unlike C, the `errno` value is a part of the exception packet, it is *not* accessed through a global variable.

For reference purposes, the Unix `errno` numbers are bound to the variables `errno/perm`, `errno/noent`, *etc.* System calls never return `error/intr`—they automatically retry. (Currently only true for I/O calls.)

(`errno-error` *errno syscall . data*) $\longrightarrow$ *no return value* procedure

Raises a Unix error exception for Unix error number *errno*. The *syscall* and *data* arguments are packaged up in the exception packet passed to the exception handler.

`(with-errno-handler*` *handler thunk*`)` $\longrightarrow$ *value(s) of thunk* procedure
`(with-errno-handler` *handler-spec . body*`)` $\longrightarrow$ *value of body* syntax

Unix syscalls raise error exceptions by calling `errno-error`. Programs can use `with-errno-handler*` to establish handlers for these exceptions.

If a Unix error arises while *thunk* is executing, *handler* is called on two arguments:

(*handler errno packet*)

packet is a list of the form

$$packet = (errno\text{-}msg\ \ syscall\ \ .\ \ data),$$

where *errno-msg* is the standard Unix error message for the error, *syscall* is the procedure that generated the error, and *data* is a list of information generated by the error, which varies from syscall to syscall.

If *handler* returns, the handler search continues upwards. *Handler* can acquire the exception by invoking a saved continuation. This procedure can be sugared over with the following syntax:

```
(with-errno-handler
    ((errno packet) clause ...)
  body1
  body2
  ...)
```

This form executes the body forms with a particular errno handler installed. When an errno error is raised, the handler search machinery will bind variable *errno* to the error's integer code, and variable *packet* to the error's auxiliary data packet. Then, the clauses will be checked for a match. The first clause that matches is executed, and its value is the value of the entire `with-errno-handler` form. If no clause matches, the handler search continues.

Error clauses have two forms

```
((errno ...) body ...)
(else body ...)
```

In the first type of clause, the *errno* forms are integer expressions. They are evaluated and compared to the error's errno value. An `else` clause matches any errno value. Note that the *errno* and *data* variables are lexically visible to the error clauses.

Example:

```
(with-errno-handler
    ((errno packet) ; Only handle 3 particular errors.
     ((errno/wouldblock errno/again)
      (loop))
     ((errno/acces)
      (format #t "Not allowed access!")
      #f))

  (foo frobbotz)
  (blatz garglemumph))
```

It is not defined what dynamic context the handler executes in, so fluid variables cannot reliably be referenced.

Note that Scsh system calls always retry when interrupted, so that the `errno/intr` exception is never raised. If the programmer wishes to abort a system call on an interrupt, he should have the interrupt handler explicitly raise an exception or invoke a stored continuation to throw out of the system call.

> *Remark:* This is not strictly true in the current implementation—only some of the i/o syscalls loop. But BSD variants never return `EINTR` anyway, unless you explicitly request it, so we'll live w/it for now.

3.1.1 Interactive mode and error handling

Scsh runs in two modes: interactive and script mode. It starts up in interactive mode if the scsh interpreter is started up with no script argument. Otherwise, scsh starts up in script mode. The mode determines whether scsh prints prompts in between reading and evaluating forms, and it affects the default error handler. In interactive mode, the default error handler will report the error, and generate an interactive breakpoint so that the user can interact with the system to examine, fix, or dismiss from the error. In script mode, the default error handler causes the scsh process to exit.

When scsh forks a child with `(fork)`, the child resets to script mode. This can be overridden if the programmer wishes.

3.2 I/O

3.2.1 Standard R4RS I/O procedures

In scsh, most standard R4RS `i/o` operations (such as `display` or `read-char`) work on both integer file descriptors and Scheme ports. When doing `i/o` with a file descriptor, the `i/o` operation is done directly on the file, bypassing any buffered data that may have accumulated in an associated port. Note that character-at-a-time operations (*e.g.*, `read-char` and `read-line`) are likely to be quite slow when performed directly upon file descriptors.

The standard R4RS procedures `read-char`, `char-ready?`, `write`, `display`, `newline`, and `write-char` are all generic, accepting integer file descriptor arguments as well as ports. Scsh also mandates the availability of `format`, and further requires `format` to accept file descriptor arguments as well as ports.

The procedures `peek-char` and `read` do *not* accept file descriptor arguments, since these functions require the ability to read ahead in the input stream, a feature not supported by Unix I/O.

3.2.2 Port manipulation and standard ports

(`close-after` *port consumer*) $\longrightarrow$ *value(s) of consumer* — procedure

Returns (*consumer port*), but closes the port on return. No dynamic-wind magic.

> *Remark:* Is there a less-awkward name?

(`error-output-port`) $\longrightarrow$ *port* — procedure

This procedure is analogous to `current-output-port`, but produces a port used for error messages—the scsh equivalent of stderr.

(`with-current-input-port*` *port thunk*) $\longrightarrow$ *value(s) of thunk* — procedure
(`with-current-output-port*` *port thunk*) $\longrightarrow$ *value(s) of thunk* — procedure
(`with-error-output-port*` *port thunk*) $\longrightarrow$ *value(s) of thunk* — procedure

These procedures install *port* as the current input, current output, and error output port, respectively, for the duration of a call to *thunk*.

(`with-current-input-port` *port . body*) $\longrightarrow$ *value(s) of body* — syntax
(`with-current-output-port` *port . body*) $\longrightarrow$ *value(s) of body* — syntax
(`with-error-output-port` *port . body*) $\longrightarrow$ *value(s) of body* — syntax

These special forms are simply syntactic sugar for the `with-current-input-port*` procedure and friends.

(close *port/fd*) ⟶ *undefined* procedure

Close the port or file descriptor.

If *port/fd* is a file descriptor, and it has a port allocated to it, the port is shifted to a new file descriptor created with (dup port/fd) before closing port/fd. The port then has its revealed count set to zero. This reflects the design criteria that ports are not associated with file descriptors, but with open files.

To close a file descriptor, and any associated port it might have, you must instead say one of (as appropriate):

```
(close (fdes->inport  fd))
(close (fdes->outport fd))
```

(stdports->stdio) ⟶ *undefined* procedure
(stdio->stdports *thunk*) ⟶ *value(s) of thunk* procedure

(stdports->stdio) is exactly equivalent to the series of redirections:[1]

```
(dup (current-input-port)  0)
(dup (current-output-port) 1)
(dup (error-output-port)   2)
```

stdio->stdports binds the standard ports (current-input-port), (current-output-port), and (error-output-port) to be ports on file descriptors 0, 1, 2, and then calls *thunk*. It is equivalent to:

```
(with-current-input-port (fdes->inport 0)
  (with-current-output-port (fdes->inport 1)
    (with-error-output-port (fdes->outport 2)
      (thunk))))
```

3.2.3 String ports

Scheme 48 has string ports, which you can use. Scsh has not committed to the particular interface or names that Scheme 48 uses, so be warned that the interface described herein may be liable to change.

(make-string-input-port *string*) ⟶ *port* procedure

Returns a port that reads characters from the supplied string.

(make-string-output-port) ⟶ *port* procedure
(string-output-port-output *port*) ⟶ *port* procedure

[1] Why not move->fdes? Because the current output port and error port might be the same port.

A string output port is a port collects the characters given to it into a string. The accumulated string is retrieved by applying `string-output-port-output` to the port.

(`call-with-string-output-port` *procedure*) $\longrightarrow$ *string* procedure

The procedure is called on a port. When it returns, `call-with-string-output-port` returns a string containing the characters written to the port.

3.2.4 Revealed ports and file descriptors

The material in this section and the following one is not critical for most applications. You may safely skim or completely skip this section on a first reading.

Dealing with Unix file descriptors in a Scheme environment is difficult. In Unix, open files are part of the process environment, and are referenced by small integers called *file descriptors*. Open file descriptors are the fundamental way i/o redirections are passed to subprocesses, since file descriptors are preserved across fork's and exec's.

Scheme, on the other hand, uses ports for specifying i/o sources. Ports are garbage-collected Scheme objects, not integers. Ports can be garbage collected; when a port is collected, it is also closed. Because file descriptors are just integers, it's impossible to garbage collect them—you wouldn't be able to close file descriptor 3 unless there were no 3's in the system, and you could further prove that your program would never again compute a 3. This is difficult at best.

If a Scheme program only used Scheme ports, and never actually used file descriptors, this would not be a problem. But Scheme code must descend to the file descriptor level in at least two circumstances:

- when interfacing to foreign code
- when interfacing to a subprocess.

This causes a problem. Suppose we have a Scheme port constructed on top of file descriptor 2. We intend to fork off a program that will inherit this file descriptor. If we drop references to the port, the garbage collector may prematurely close file 2 before we fork the subprocess. The interface described below is intended to fix this and other problems arising from the mismatch between ports and file descriptors.

The Scheme kernel maintains a port table that maps a file descriptor to the Scheme port allocated for it (or, `#f` if there is no port allocated for this file descriptor). This is used to ensure that there is at most one open port for each open file descriptor.

The port data structure for file ports has two fields besides the descriptor: revealed and closed?. When a file port is closed with `(close port)`, the port's file descriptor is closed, its entry in the port table is cleared, and the port's closed? field is set to true.

When a file descriptor is closed with `(close fdes)`, any associated port is shifted to a new file descriptor created with `(dup fdes)`. The port has its revealed count reset to zero. See discussion below. To really put a stake through a descriptor's heart, you must say one of

```
(close (fdes->inport fdes))
(close (fdes->output fdes))
```

The revealed field is an aid to garbage collection. It is an integer semaphore. If it is zero, the port's file descriptor can be closed when the port is collected. Essentially, the revealed field reflects whether or not the port's file descriptor has escaped to the Scheme user. If the Scheme user doesn't know what file descriptor is associated with a given port, then he can't possibly retain an "integer handle" on the port after dropping pointers to the port itself, so the garbage collector is free to close the file.

Ports allocated with `open-output-file` and `open-input-file` are unrevealed ports—*i.e.*, revealed is initialised to 0. No one knows the port's file descriptor, so the file descriptor can be closed when the port is collected.

The functions `fdes->output-port`, `fdes->input-port`, `port->fdes` are used to shift back and forth between file descriptors and ports. When `port->fdes` reveals a port's file descriptor, it increments the port's revealed field. When the user is through with the file descriptor, he can call `(release-port-handle port)`, which decrements the count. The function `(call/fdes fdes/port proc)` automates this protocol. `call/fdes` uses `dynamic-wind` to enforce the protocol. If `proc` throws out of the `call/fdes`, unwind handler releases the descriptor handle; if the user subsequently tries to throw *back* into `proc`'s context, the wind handler raises an error. When the user maps a file descriptor to a port with `fdes->outport` or `fdes->inport`, the port has its revealed field incremented.

Not all file descriptors are created by requests to make ports. Some are inherited on process invocation via `exec(2)`, and are simply part of the global environment. Subprocesses may depend upon them, so if a port is later allocated for these file descriptors, is should be considered as a revealed port. For example, when the Scheme shell's process starts up, it opens ports on file descriptors 0, 1, and 2 for the initial values of `(current-input-port)`, `(current-output-port)`, and `(error-output-port)`. These ports are initialised with revealed set to 1, so that stdin, stdout, and stderr are not closed even if the user drops the port. A fine point: the stdin file descriptor is allocated an unbuffered port. Because shells frequently share stdin with subprocesses, if the shell does buffered reads, it might "steal" input intended for a subprocess. For this reason, all shells, including sh, csh, and scsh, read stdin unbuffered. Responsibility for deciding which other files must be opened unbuffered rests with the shell programmer.

Unrevealed file ports have the nice property that they can be closed when all pointers to the port are dropped. This can happen during gc, or at an `exec()`—since all memory is dropped at an `exec()`. No one knows the file descriptor associated with the port, so the exec'd process certainly can't refer to it.

This facility preserves the transparent close-on-collect property for file ports that are used in straightforward ways, yet allows access to the underlying Unix substrate without interference from the garbage collector. This is critical, since shell programming absolutely requires access to the Unix file descriptors, as their numerical values are a critical part of the process interface.

A port's underlying file descriptor can be shifted around with `dup(2)` when convenient. That is, the actual fd on top of which a port is constructed can be shifted around underneath the port by the scsh kernel when necessary. This is important, because when the user is setting up file descriptors prior to a `exec(2)`, he may explicitly use a file descriptor that has already been allocated to some port. In this case, the scsh kernel just shifts the port's file descriptor to some new location with `dup`, freeing up its old descriptor. This prevents errors from happening in the following scenario. Suppose we have a file open on port `f`. Now we want to run a program that reads input on file 0, writes output to file 1, errors to file 2, and logs execution information on file 3. We want to run this program with input from `f`. So we write:

```
(run (/usr/shivers/bin/prog)
     (> 1 output.txt)
     (> 2 error.log)
     (> 3 trace.log)
     (= 0 ,f))
```

Now, suppose by ill chance that, unbeknownst to us, when the operating system opened `f`'s file, it allocated descriptor 3 for it. If we blindly redirect `trace.log` into file descriptor 3, we'll clobber `f`! However, the port-shuffling machinery saves us: when the `run` form tries to dup `trace.log`'s file descriptor to 3, `dup` will notice that file descriptor 3 is already associated with an unrevealed port (*i.e.*, `f`). So, it will first move `f` to some other file descriptor. This keeps `f` alive and well so that it can subsequently be dup'd into descriptor 0 for `prog`'s stdin.

The port-shifting machinery makes the following guarantee: a port is only moved when the underlying file descriptor is closed, either by a `close()` or a `dup2()` operation. Otherwise a port/file-descriptor association is stable.

Under normal circumstances, all this machinery just works behind the scenes to keep things straightened out. The only time the user has to think about it is when he starts accessing file descriptors from ports, which he should almost never have to do. If a user starts asking what file descriptors have been allocated to what ports, he has to take responsibility for managing this information.

3.2.5 Port-mapping machinery

The procedures provided in this section are almost never needed. You may safely skim or completely skip this section on a first reading.

Here are the routines for manipulating ports in scsh. The important points to remember are:

- A file port is associated with an open file, not a particular file descriptor.
- The association between a file port and a particular file descriptor is never changed *except* when the file descriptor is explicitly closed. "Closing" includes being used as the target of a `dup2`, so the set of procedures below that close their targets are `close`, two-argument `dup`, and `move->fdes`. If the target file descriptor of one of these routines has an allocated port, the port will be shifted to another freshly-allocated file descriptor, and marked as unrevealed, thus preserving the port but freeing its old file descriptor.

These rules are what is necessary to "make things work out" with no surprises in the general case.

`(fdes->inport` *fd*`)` ⟶ *port* procedure
`(fdes->outport` *fd*`)` ⟶ *port* procedure
`(port->fdes` *port*`)` ⟶ *fixnum* procedure

These increment the port's revealed count.

`(port-revealed` *port*`)` ⟶ *integer or* `#f` procedure

Return the port's revealed count if positive, otherwise `#f`.

`(release-port-handle` *port*`)` ⟶ *undefined* procedure

Decrement the port's revealed count.

`(call/fdes` *fd/port consumer*`)` ⟶ *value(s) of consumer* procedure

Calls *consumer* on a file descriptor; takes care of revealed bookkeeping. If *fd/port* is a file descriptor, this is just `(`*consumer fd/port*`)`. If *fd/port* is a port, calls *consumer* on its underlying file descriptor. While *consumer* is running, the port's revealed count is incremented.

When `call/fdes` is called with port argument, you are not allowed to throw into *consumer* with a stored continuation, as that would violate the revealed-count bookkeeping.

`(move->fdes` *fd/port target-fd*`)` ⟶ *port or fdes* procedure

Maps fd→fd and port→port.

If *fd/port* is a file-descriptor not equal to *target-fd*, dup it to *target-fd* and close it. Returns *target-fd*.

If *fd/port* is a port, it is shifted to *target-fd*, by duping its underlying file-descriptor if necessary. *Fd/port*'s original file descriptor is closed (if it was different from *target-fd*). Returns the port. This operation resets *fd/port*'s revealed count to 1.

In all cases when *fd/port* is actually shifted, if there is a port already using *target-fd*, it is first relocated to some other file descriptor.

3.2.6 Unix I/O

`(dup` *port/fd [newfd]*`)` ⟶ *port/fd* procedure
`(dup->inport` *port/fd [newfd]*`)` ⟶ *port* procedure
`(dup->outport` *port/fd [newfd]*`)` ⟶ *port* procedure
`(dup->fdes` *port/fd [newfd]*`)` ⟶ *fd* procedure

These procedures subsume the functionality of C's `dup()` and `dup2()`. The different routines return different types of values: `dup->inport`, `dup->outport`, and `dup->fdes` return input ports, output ports, and integer file descriptors, respectively. `dup`'s return value depends on on the type of *port/fd*—it maps fd→fd and port→port.

These procedures use the Unix `dup()` syscall to replicate the file descriptor or file port *port/fd*. If a *newfd* file descriptor is given, it is used as the target of the dup operation, *i.e.*, the operation is a `dup2()`. In this case, procedures that return a port (such as `dup->inport`) will return one with the revealed count set to one. For example, `(dup (current-input-port) 5)` produces a new port with underlying file descriptor 5, whose revealed count is 1. If *newfd* is not specified, then the operating system chooses the file descriptor, and any returned port is marked as unrevealed.

If the *newfd* target is given, and some port is already using that file descriptor, the port is first quietly shifted (with another `dup`) to some other file descriptor (zeroing its revealed count).

Since Scheme doesn't provide read/write ports, `dup->inport` and `dup->outport` can be useful for getting an output version of an input port, or *vice versa*. For example, if `p` is an input port open on a tty, and we would like to do output to that tty, we can simply use `(dup->outport p)` to produce an equivalent output port for the tty.

`(file-seek` *fd/port offset whence*`)` $\longrightarrow$ *undefined* procedure

whence is one of {`seek/set`, `seek/delta`, `seek/end`}.

> *Oops:* The current implementation doesn't handle *offset* arguments that are not immediate integers (*i.e.*, representable in 30 bits).

`(open-file` *fname flags [perms]*`)` $\longrightarrow$ *port* procedure

Perms defaults to `#o666`. *Flags* is an integer bitmask, composed by or'ing together the following constants:

```
open/read               ; You may only
open/write              ; choose one
open/read+write         ; of these three
open/no-control-tty
open/nonblocking
open/append
open/create
open/truncate
open/exclusive
      .                 ; Your Unix may have
      .                 ; a few more.
```

Returns a port. The port is an input port if the `flags` permit it, otherwise an output port. R4RS/Scheme 48/scsh do not have input/output ports, so it's one or the other. This should be fixed. (You can hack simultaneous i/o on a file by opening it r/w, taking the result input port, and duping it to an output port with `dup->outport`.)

`(open-input-file` *fname [flags]*`)` $\longrightarrow$ *port* procedure
`(open-output-file` *fname [flags perms]*`)` $\longrightarrow$ *port* procedure

These are equivalent to `open-file`, after first setting the read/write bits of the *flags* argument to `open/read` or `open/write`, respectively. *Flags* defaults to zero for `open-input-file`, and

```
(bitwise-ior open/create open/truncate)
```

for `open-output-file`. These defaults make the procedures backwards-compatible with their unary R4RS definitions.

(`open-fdes` *fname flags [perms]*) $\longrightarrow$ *integer* — procedure

Returns a file descriptor.

(`pipe`) $\longrightarrow$ *[rport wport]* — procedure

Returns two ports, the read and write end-points of a Unix pipe.

(`read-line` *[fd/port retain-newline?]*) $\longrightarrow$ *string or eof-object* — procedure

Reads and returns one line of text; on eof, returns the eof object. A line is terminated by newline or eof.

retain-newline? defaults to `#f`; if true, a terminating newline is included in the result string, otherwise it is trimmed. Using this argument allows one to tell whether or not the last line of input in a file is newline terminated.

(`read-string` *nbytes [fd/port]*) $\longrightarrow$ *string or* `#f` — procedure
(`read-string!` *str [fd/port start end]*) $\longrightarrow$ *nread or* `#f` — procedure

These calls read exactly as much data as you requested, unless there is not enough data (eof). `read-string!` reads the data into string *str* at the indices in the half-open interval $[start, end)$; the default interval is the whole string: $start = 0$ and $end =$ (`string-length` *string*). They will persistently retry on partial reads and when interrupted until (1) error, (2) eof, or (3) the input request is completely satisfied. Partial reads can occur when reading from an intermittent source, such as a pipe or tty.

`read-string` returns the string read; `read-string!` returns the number of characters read. They both return false at eof. A request to read zero bytes returns immediately, with no eof check.

The values of *start* and *end* must specify a well-defined interval in *str*, *i.e.*, $0 \leq start \leq end \leq$ (`string-length` *str*).

Any partially-read data is included in the error exception packet. Error returns on non-blocking input are considered an error.

(`read-string/partial` *nbytes [fd/port]*) $\longrightarrow$ *string or* `#f` — procedure
(`read-string!/partial` *str [fd/port start end]*) $\longrightarrow$ *nread or* `#f` — procedure

These are atomic best-effort/forward-progress calls. Best effort: they may read less than you request if there is a lesser amount of data immediately available (*e.g.*, because you are reading from a pipe or a tty). Forward progress: if no data is immediately available (*e.g.*, empty pipe), they will block. Therefore, if you request an $n > 0$ byte read, while you may not get everything you asked for, you will always get something (barring eof).

There is one case in which the forward-progress guarantee is cancelled: when the programmer explicitly sets the port to non-blocking `i/o`. In this case, if no data is immediately available, the procedure will not block, but will immediately return a zero-byte read.

`read-string/partial` reads the data into a freshly allocated string, which it returns as its value. `read-string!/partial` reads the data into string *str* at the indices in the half-open interval $[start, end)$; the default interval is the whole string: $start = 0$ and $end =$ `(string-length` *string*`)`. The values of *start* and *end* must specify a well-defined interval in *str*, *i.e.*, $0 \leq start \leq end \leq$ `(string-length` *str*`)`. It returns the number of bytes read.

A request to read zero bytes returns immediatedly, with no eof check.

In sum, there are only three ways you can get a zero-byte read: (1) you request one, (2) you turn on non-blocking `i/o`, or (3) you try to read at eof.

These are the routines to use for non-blocking input. They are also useful when you wish to efficiently process data in large blocks, and your algorithm is insensitive to the block size of any particular read operation.

`(select` *readfds writefds exceptfds timeout*`)` $\longrightarrow$ *rfds wfds efds* procedure

> *Remark:* Unimplemented. Should we implement a set-of abstraction first, Or just use a twos-complement bitvector encoding with bignums?

`(write-string` *string [fd/port start end]*`)` $\longrightarrow$ *undefined* procedure

This procedure writes all the data requested. If the procedure cannot perform the write with a single kernel call (due to interrupts or partial writes), it will perform multiple write operations until all the data is written or an error has occurred. A non-blocking `i/o` error is considered an error. (Error exception packets for this syscall include the amount of data partially transferred before the error occurred.)

The data written are the characters of *string* in the half-open interval $[start, end)$. The default interval is the whole string: $start = 0$ and $end =$ `(string-length` *string*`)`. The values of *start* and *end* must specify a well-defined interval in *str*, *i.e.*, $0 \leq start \leq end \leq$ `(string-length` *str*`)`. A zero-byte write returns immediately, with no error.

Output to buffered ports: `write-string`'s efforts end as soon as all the data has been placed in the output buffer. Errors and true output may not happen until a later time, of course.

(`write-string/partial` *string [fd/port start end]*) $\longrightarrow$ *nwritten* procedure

This routine is the atomic best-effort/forward-progress analog to `write-string`. It returns the number of bytes written, which may be less than you asked for. Partial writes can occur when (1) we write off the physical end of the media, (2) the write is interrrupted, or (3) the file descriptor is set for non-blocking i/o.

If the file descriptor is not set up for non-blocking i/o, then a successful return from these procedures makes a forward progress guarantee—that is, a partial write took place of at least one byte:

- If we are at the end of physical media, and no write takes place, an error exception is raised. So a return implies we wrote *something*.
- If the call is interrupted after a partial transfer, it returns immediately. But if the call is interrupted before any data transfer, then the write is retried.

If we request a zero-byte write, then the call immediately returns 0. If the file descriptor is set for non-blocking i/o, then the call may return 0 if it was unable to immediately write anything (*e.g.*, full pipe). Barring these two cases, a write either returns $nwritten > 0$, or raises an error exception.

Non-blocking i/o is only available on file descriptors and unbuffered ports. Doing non-blocking i/o to a buffered port is not well-defined, and is an error (the problem is the subsequent flush operation).

(`force-output` *[fd/port]*) $\longrightarrow$ *no return value* procedure

This procedure does nothing when applied to an integer file descriptor or unbuffered port. It flushes buffered output when applied to a buffered port, and raises a write-error exception on error. Returns no value.

3.3 File system

Besides the following procedures, which allow access to the computer's file system, scsh also provides a set of procedures which manipulate file *names*. These string-processing procedures are documented in section 5.1.3.

(`create-directory` *fname [perms override?]*) ⟶ *undefined* procedure
(`create-fifo` *fname [perms override?]*) ⟶ *undefined* procedure
(`create-hard-link` *oldname newname [override?]*) ⟶ *undefined* procedure
(`create-symlink` *old-name new-name [override?]*) ⟶ *undefined* procedure

These procedures create objects of various kinds in the file system.

The *override?* argument controls the action if there is already an object in the file system with the new name:

`#f`	signal an error (default)
`'query`	prompt the user
other	delete the old object (with `delete-file` or `delete-directory`, as appropriate) before creating the new object.

Perms defaults to `#o777` (but is masked by the current umask).

> *Remark:* Currently, if you try to create a hard or symbolic link from a file to itself, you will error out with *override?* false, and simply delete your file with *override?* true. Catching this will require some sort of true-name procedure, which I currently do not have.

(`delete-directory` *fname*) ⟶ *undefined* procedure
(`delete-file` *fname*) ⟶ *undefined* procedure
(`delete-filesys-object` *fname*) ⟶ *undefined* procedure

These procedures delete objects from the file system. The `delete-filesys-object` procedure will delete an object of any type from the file system: files, (empty) directories, symlinks, fifos, *etc.*.

(`read-symlink` *fname*) ⟶ *string* procedure

Return the filename referenced by symbolic link `fname`.

(`rename-file` *old-fname new-fname [override?]*) ⟶ *undefined* procedure

If you override an existing object, then *old-fname* and *new-fname* must type-match—either both directories, or both non-directories. This is required by the semantics of Unix `rename()`.

> *Remark:* There is an unfortunate atomicity problem with the `rename-file` procedure: if you specify no-override, but create file `new-fname` sometime between `rename-file`'s existence check and the actual rename operation, your file will be clobbered with `old-fname`. There is no way to fix this problem, given the semantics of Unix `rename()`; at least it is highly unlikely to occur in practice.

(`set-file-mode` *fname/fd/port mode*) $\longrightarrow$ *undefined* procedure
(`set-file-owner` *fname/fd/port uid*) $\longrightarrow$ *undefined* procedure
(`set-file-group` *fname/fd/port gid*) $\longrightarrow$ *undefined* procedure

These procedures set the permission bits, owner id, and group id of a file, respectively. The file can be specified by giving the file name, or either an integer file descriptor or a port open on the file. Setting file user or group ownership usually requires root privileges.

(`sync-file` *fd/port*) $\longrightarrow$ *undefined* procedure
(`sync-file-system`) $\longrightarrow$ *undefined* procedure

Calling `sync-file` causes Unix to update the disk data structures for a given file. If *fd/port* is a port, any buffered data it may have is first flushed. Calling `sync-file-system` synchronises the kernel's entire file system with the disk.

These procedures are not POSIX. Interestingly enough, `sync-file-system` doesn't actually do what it is claimed to do. We just threw it in for humor value. See the `sync(2)` man page for Unix enlightenment.

(`truncate-file` *fname/fd/port len*) $\longrightarrow$ *undefined* procedure

The specified file is truncated to *len* bytes in length.

(`file-attributes` *fname/fd/port [chase?]*) $\longrightarrow$ *file-info-record* procedure

The `file-attributes` procedure returns a record structure containing everything there is to know about a file. If the *chase?* flag is true (the default), then the procedure chases symlinks and reports on the files to which they refer. If *chase?* is false, then the procedure checks the actual file itself, even if it's a symlink. The *chase?* flag is ignored if the file argument is a file descriptor or port.

The value returned is a *file-info record*, defined to have the following structure:

```
(define-record file-info
  type      ; {block-special, char-special, directory,
            ;     fifo, regular, socket, symlink}
  device    ; Device file resides on.
  inode     ; File's inode.
  mode      ; File's mode bits: permissions, setuid, setgid
  nlinks    ; Number of hard links to this file.
  uid       ; Owner of file.
  gid       ; File's group id.
  size      ; Size of file, in bytes.
  atime     ; Last access time.
  mtime     ; Last status-change time.
  ctime)    ; Creation time.
```

The uid field of a file-info record is accessed with the procedure

```
(file-info:uid x)
```

and similarly for the other fields. The `type` field is a symbol; all other fields are integers. A file-info record is discriminated with the `file-info?` predicate.

The following procedures all return selected information about a file; they are built on top of `file-attributes`, and are called with the same arguments that are passed to it.

Procedure	returns
`file-type`	type
`file-inode`	inode
`file-mode`	mode
`file-nlinks`	nlinks
`file-owner`	uid
`file-group`	gid
`file-size`	size
`file-last-access`	atime
`file-last-mod`	mtime
`file-last-status-change`	ctime

Example:

```
;; All my files in /usr/tmp:
(filter (λ (f) (= (file-owner f) (user-uid)))
        (directory-files "/usr/tmp")))
```

(`file-directory?` *fname/fd/port [chase?]*) $\longrightarrow$ *boolean* procedure
(`file-fifo?` *fname/fd/port [chase?]*) $\longrightarrow$ *boolean* procedure

(`file-regular?` *fname/fd/port [chase?]*) ⟶ *boolean* procedure
(`file-socket?` *fname/fd/port [chase?]*) ⟶ *boolean* procedure
(`file-special?` *fname/fd/port [chase?]*) ⟶ *boolean* procedure
(`file-symlink?` *fname/fd/port*) ⟶ *boolean* procedure

These procedures are file-type predicates that test the type of a given file. The are applied to the same arguments to which `file-attributes` is applied; the sole exception is `file-symlink?`, which does not take the optional *chase?* second argument.

For example,

```
(file-directory? "/usr/dalbertz")    ⟹    #t
```

(`file-not-readable?` *fname*) ⟶ *boolean* procedure
(`file-not-writeable?` *fname*) ⟶ *boolean* procedure
(`file-not-executable?` *fname*) ⟶ *boolean* procedure

Returns:

Value	meaning
`#f`	Access permitted
`'search-denied`	Can't stat—a protected directory is blocking access.
`'permission`	Permission denied.
`'no-directory`	Some directory doesn't exist.
`'nonexistent`	File doesn't exist.

A file is considered writeable if either (1) it exists and is writeable or (2) it doesn't exist and the directory is writeable. Since symlink permission bits are ignored by the filesystem, these calls do not take a *chase?* flag.

> *Oops:* `file-not-writeable?` does not currently do the directory check.

(`file-readable?` *fname*) ⟶ *boolean* procedure
(`file-writable?` *fname*) ⟶ *boolean* procedure
(`file-executable?` *fname*) ⟶ *boolean* procedure

These procedures are the logical negation of the preceding `file-not-...?` procedures.

(`file-not-exists?` *fname [chase?]*) ⟶ *object* procedure

Returns:

`#f`	Exists.
`#t`	Doesn't exist.
`'search-denied`	Some protected directory is blocking the search.

`(file-exists?` *fname [chase?]*`)` $\longrightarrow$ *boolean* procedure

This is simply `(not (file-not-exists?` *fname [chase?]*`))`

`(directory-files` *[dir dotfiles?]*`)` $\longrightarrow$ *string list* procedure

Return the list of files in directory *dir*, which defaults to the current working directory. The *dotfiles?* flag (default `#f`) causes dot files to be included in the list. Regardless of the value of *dotfiles?*, the two files `.` and `..` are *never* returned.

The directory *dir* is not prepended to each file name in the result list. That is,

```
(directory-files "/etc")
```

returns

```
("chown" "exports" "fstab" ...)
```

not

```
("/etc/chown" "/etc/exports" "/etc/fstab" ...)
```

To use the files in returned list, the programmer can either manually prepend the directory:

```
(map (λ (f) (string-append dir "/" f)) files)
```

or cd to the directory before using the file names:

```
(with-cwd dir
  (for-each delete-file (directory-files)))
```

or use the `glob` procedure, defined below.

A directory list can be generated by `(run/strings (ls))`, but this is unreliable, as filenames with whitespace in their names will be split into separate entries. Using `directory-files` is reliable.

`(glob` pat_1 `...)` $\longrightarrow$ *string list* procedure

Glob each pattern against the filesystem and return the sorted list. Duplicates are not removed. Patterns matching nothing are not included literally.[2] C shell `{a,b,c}` patterns are expanded. Backslash quotes characters, turning off the special meaning of `{`, `}`, `*`, `[`, `]`, and `?`.

Note that the rules of backslash for Scheme strings and glob patterns work together to require four backslashes in a row to specify a single literal backslash. Fortunately, this should be a rare occurrence.

A glob subpattern will not match against dot files unless the first character of the subpattern is a literal "`.`". Further, a dot subpattern will not match the files `.` or `..` unless it is a constant pattern, as in `(glob "../*/*.c")`. So a directory's dot files can be reliably generated with the simple glob pattern `".*"`.

Some examples:

```
(glob "*.c" "*.h")
    ;; All the C and #include files in my directory.

(glob "*.c" "*/*.c")
    ;; All the C files in this directory and
    ;; its immediate subdirectories.

(glob "lexer/*.c" "parser/*.c")
(glob "{lexer,parser}/*.c")
    ;; All the C files in the lexer and parser dirs.

(glob "\\{lexer,parser\\}/*.c")
    ;; All the C files in the strange
    ;; directory "{lexer,parser}".

(glob "*\\*")
    ;; All the files ending in "*", e.g.
    ;; ("foo*" "bar*")

(glob "*lexer*")
    ("mylexer.c" "lexer1.notes")
    ;; All files containing the string "lexer".

(glob "lexer")
    ;; Either ("lexer") or ().
```

If the first character of the pattern (after expanding braces) is a slash, the search begins at root; otherwise, the search begins in the current working directory.

[2]Why bother to mention such a silly possibility? Because that is what sh does.

If the last character of the pattern (after expanding braces) is a slash, then the result matches must be directories, *e.g.*,

```
(glob "/usr/man/man?/") ⟹
        ("/usr/man/man1/" "/usr/man/man2/" ...)
```

Globbing can sometimes be useful when we need a list of a directory's files where each element in the list includes the pathname for the file. Compare:

```
(directory-files "../include") ⟹
    ("cig.h" "decls.h" ...)

(glob "../include/*") ⟹
    ("../include/cig.h" "../include/decls.h" ...)
```

(`glob-quote` *str*) ⟶ *string* procedure

Returns a constant glob pattern that exactly matches *str*. All wild-card characters in *str* are quoted with a backslash.

```
(glob-quote "Any *.c files?")
    ⟹ "Any \*.c files\?"
```

(`file-match` *root dot-files?* pat_1 pat_2 ... pat_n) ⟶ *string list* procedure

`file-match` provides a more powerful file-matching service, at the expense of a less convenient notation. It is intermediate in power between most shell matching machinery and recursive `find(1)`.

Each pattern is a regexp. The procedure searches from *root*, matching the first-level files against pattern pat_1, the second-level files against pat_2, and so forth. The list of files matching the whole path pattern is returned, in sorted order. The matcher uses Spencer's regular expression package.

The files `.` and `..` are never matched. Other dot files are only matched if the *dot-files?* argument is `#t`.

A given pat_i pattern is matched as a regexp, so it is not forced to match the entire file name. *E.g.*, pattern `"t"` matches any file containing a "t" in its name, while pattern `"^t$"` matches only a file whose entire name is "`t`".

The pat_i patterns can be more general than stated above.

- A single pattern can specify multiple levels of the path by embedding `/` characters within the pattern. For example, the pattern `"a/b/c"` gives a match equivalent to the list of patterns `"a" "b" "c"`.

- A pat_i pattern can be a procedure, which is used as a match predicate. It will be repeatedly called with a candidate file-name to test. The file-name will be the entire path accumulated.

Some examples:

```
(file-match "/usr/lib" #f "m$" "^tab") ⟹
    ("/usr/lib/term/tab300" "/usr/lib/term/tab300-12" ...)

(file-match "." #f  "^lex|parse|codegen$" "\\.c$") ⟹
    ("lex/lex.c" "lex/lexinit.c" "lex/test.c"
     "parse/actions.c" "parse/error.c" parse/test.c"
     "codegen/io.c" "codegen/walk.c")

(file-match "." #f  "^lex|parse|codegen$/\\.c$")
     ;; The same.

(file-match "." #f  file-directory?)
    ;; Return all subdirs of the current directory.

(file-match "/" #f  file-directory?) ⟹
    ("/bin" "/dev" "/etc" "/tmp" "/usr")
    ;; All subdirs of root.

(file-match "." #f  "\\.c")
    ;; All the C files in my directory.

(define (ext extension)
  (λ (fn) (string-suffix? fn extension)))

(define (true . x) #t)

(file-match "." #f  "./\\.c")
(file-match "." #f  "" "\\.c")
(file-match "." #f  true "\\.c")
(file-match "." #f  true (ext "c"))
    ;; All the C files of all my immediate subdirs.

(file-match "." #f "lexer") ⟹
    ("mylexer.c" "lexer.notes")
    ;; Compare with (glob "lexer"), above.
```

Note that when *root* is the current working directory ("."), when it is converted to directory form, it becomes "", and doesn't show up in the result file-names.

It is regrettable that the regexp wild card char, "`.`", is such an important file name literal, as dot-file prefix and extension delimiter.

`(create-temp-file` *[prefix]*`)` $\longrightarrow$ *string* procedure

`Create-temp-file` creates a new temporary file and return its name. The optional argument specifies the filename prefix to use, and defaults to `"/usr/tmp/`*pid*`"`, where *pid* is the current process' id. The procedure generates a sequence of filenames that have *prefix* as a common prefix, looking for a filename that doesn't already exist in the file system. When it finds one, it creates it, with permission `#o600` and returns the filename. (The file permission can be changed to a more permissive permission with `set-file-mode` after being created).

This file is guaranteed to be brand new. No other process will have it open. This procedure does not simply return a filename that is very likely to be unused. It returns a filename that definitely did not exist at the moment `create-temp-file` created it.

It is not necessary for the process' pid to be a part of the filename for the uniqueness guarantees to hold. The pid component of the default prefix simply serves to scatter the name searches into sparse regions, so that collisions are less likely to occur. This speeds things up, but does not affect correctness.

Security note: doing i/o to files created this way in `/usr/tmp/` is not necessarily secure. General users have write access to `/usr/tmp/`, so even if an attacker cannot access the new temp file, he can delete it and replace it with one of his own. A subsequent open of this filename will then give you his file, to which he has access rights. There are several ways to defeat this attack,

1. Use `temp-file-iterate`, below, to return the file descriptor allocated when the file is opened. This will work if the file only needs to be opened once.
2. If the file needs to be opened twice or more, create it in a protected directory, , `$HOME`.
3. Ensure that `/usr/tmp` has its sticky bit set. This requires system administrator privileges.

The actual default prefix used is controlled by the dynamic variable `*temp-file-template*`, and can be overridden for increased security. See `temp-file-iterate`.

`(temp-file-iterate` *maker [template]*`)` $\longrightarrow$ *object*$^{+}$ procedure `*temp-file-template*` *string*

This procedure can be used to perform certain atomic transactions on the file system involving filenames. Some examples:

- Linking a file to a fresh backup temp name.
- Creating and opening an unused, secure temp file.
- Creating an unused temporary directory.

This procedure uses *template* to generate a series of trial file names. *Template* is a `format` control string, and defaults to

```
"/usr/tmp/pid.~a"
```

where *pid* is the current process' process id. File names are generated by calling `format` to instantiate the template's `~a` field with a varying string.

Maker is a procedure which is serially called on each file name generated. It must return at least one value; it may return multiple values. If the first return value is `#f` or if *maker* raises the `errno/exist` errno exception, `temp-file-iterate` will loop, generating a new file name and calling *maker* again. If the first return value is true, the loop is terminated, returning whatever value(s) *maker* returned.

After a number of unsuccessful trials, `temp-file-iterate` may give up and signal an error.

Thus, if we ignore its optional *prefix* argument, `create-temp-file` could be defined as:

```
(define (create-temp-file)
  (let ((flags (bitwise-ior open/create open/exclusive)))
    (temp-file-iterate
        (λ (f)
          (close (open-output-file f flags #o600))
          f))))
```

To rename a file to a temporary name:

```
(temp-file-iterate (λ (backup)
                     (create-hard-link old-file backup)
                     backup)
                   ".#temp.~a") ; Keep link in cwd.
(delete-file old-file)
```

Recall that scsh reports syscall failure by raising an error exception, not by returning an error code. This is critical to to this example—the programmer can assume that if the `temp-file-iterate` call returns, it returns successully. So the following `delete-file` call can be reliably invoked, safe in the knowledge that the backup link has definitely been established.

To create a unique temporary directory:

```
(temp-file-iterate (λ (dir) (create-directory dir) dir)
                   "/usr/tmp/tempdir.~a")
```

Similar operations can be used to generate unique symlinks and fifos, or to return values other than the new filename (*e.g.*, an open file descriptor or port).

The default template is in fact taken from the value of the dynamic variable `*temp-file-template*`, which itself defaults to `"/usr/tmp/`*pid*`.~a"`, where *pid* is the scsh process' pid. For increased security, a user may wish to change the template to use a directory not allowing world write access (*e.g.*, his home directory).

`(temp-file-channel)` $\longrightarrow$ *[inp outp]* procedure

This procedure can be used to provide an interprocess communications channel with arbitrary-sized buffering. It returns two values, an input port and an output port, both open on a new temp file. The temp file itself is deleted from the Unix file tree before `temp-file-channel` returns, so the file is essentially unnamed, and its disk storage is reclaimed as soon as the two ports are closed.

`Temp-file-channel` is analogous to `port-pipe` with two exceptions:

- If the writer process gets ahead of the reader process, it will not hang waiting for some small pipe buffer to drain. It will simply buffer the data on disk. This is good.
- If the reader process gets ahead of the writer process, it will also not hang waiting for data from the writer process. It will simply see and report an end of file. This is bad.
  In order to ensure that an end-of-file returned to the reader is legitimate, the reader and writer must serialise their `i/o`. The simplest way to do this is for the reader to delay doing input until the writer has completely finished doing output, or exited.

3.4 Processes

`(exec` *prog arg1 … argn*`)` $\longrightarrow$ *no return value* procedure
`(exec-path` *prog arg1 … argn*`)` $\longrightarrow$ *no return value* procedure
`(exec/env` *prog env arg1 … argn*`)` $\longrightarrow$ *no return value* procedure
`(exec-path/env` *prog env arg1 … argn*`)` $\longrightarrow$ *no return value* procedure

The …`/env` variants take an environment specified as a string→string alist. An environment of `#t` is taken to mean the current process' environment (*i.e.*, the value of the external char `**environ`).

[Rationale: `#f` is a more convenient marker for the current environment than `#t`, but would cause an ambiguity on Schemes that identify `#f` and `()`.]

The path-searching variants search the directories in the list `exec-path-list` for the program. A path-search is not performed if the program name contains a slash character—it is used directly. So a program with a name like `"bin/prog"` always executes the program `bin/prog` in the current working directory. See `$path` and `exec-path-list`, below.

Note that there is no analog to the C function `execv()`. To get the effect just do

```
(apply exec prog arglist)
```

All of these procedures flush buffered output and close unrevealed ports before executing the new binary. To avoid flushing buffered output, see `%exec` below.

Note that the C `exec()` procedure allows the zeroth element of the argument vector to be different from the file being executed, *e.g.*

```
char *argv[] = {"-", "-f", 0};
exec("/bin/csh", argv, envp);
```

The scsh `exec`, `exec-path`, `exec/env`, and `exec-path/env` procedures do not give this functionality—element 0 of the arg vector is always identical to the `prog` argument. In the rare case the user wishes to differentiate these two items, he can use the low-level `%exec` and `exec-path-search` procedures. These procedures never return under any circumstances. As with any other system call, if there is an error, they raise an exception.

(`%exec` *prog arglist env*) $\longrightarrow$ *undefined* procedure
(`exec-path-search` *fname pathlist*) $\longrightarrow$ *string* procedure

Arglist is a list of arguments; *env* is either a string→string alist or `#t`. The new program's `argv[0]` will be taken from (`car` *arglist*), *not* from *prog*. An environment of `#t` means the current process' environment. `%exec` does not flush buffered output (see `flush-all-ports`).

`exec-path-search` searches the directories of *pathlist* looking for an occurrence of file `fname`. If no executable file is found, it returns `#f`. If `fname` contains a slash character, the path search is short-circuited, but the procedure still checks to ensure that the file exists and is executable—if not, it still returns `#f`.

See `$path` and `exec-path-list`, below.

All exec procedures, including `%exec`, coerce the `prog` and `arg` values to strings using the usual conversion rules: numbers are converted to decimal numerals, and symbols converted to their print-names.

(`exit` *[status]*) ⟶ *no return value* procedure
(`%exit` *[status]*) ⟶ *no return value* procedure

These procedures terminate the current process with a given exit status. The default exit status is 0. The low-level `%exit` procedure immediately terminates the process without flushing buffered output.

(`suspend`) ⟶ *undefined* procedure

Suspend the current process with a SIGSTOP signal.

(`fork` *[thunk]*) ⟶ *pid or* `#f` procedure
(`%fork` *[thunk]*) ⟶ *pid or* `#f` procedure

`fork` with no arguments is like C `fork()`. In the parent process, it returns the child's pid. In the child process, it returns `#f`.

`fork` with an argument only returns in the parent process, returning the child pid. The child process calls *thunk* and then exits.

`fork` flushes buffered output before forking, and sets the child process to non-interactive. `%fork` does not perform this bookkeeping; it simply forks.

(`fork/pipe` *[thunk]*) ⟶ *pid or* `#f` procedure
(`%fork/pipe` *[thunk]*) ⟶ *pid or* `#f` procedure

Like `fork` and `%fork`, but the parent and child communicate via a pipe connecting the parent's stdin to the child's stdout. These procedures side-effect the parent by changing his stdin.

In effect, `fork/pipe` splices a process into the data stream immediately upstream of the current process. This is the basic function for creating pipelines. Long pipelines are built by performing a sequence of `fork/pipe` calls. For example, to create a background two-process pipe `a | b`, we write:

```
(fork (λ () (fork/pipe a) (b)))
```

which returns the pid of `b`'s process.

To create a background three-process pipe `a | b | c`, we write:

```
(fork (λ () (fork/pipe a)
            (fork/pipe b)
            (c)))
```

which returns the pid of `c`'s process.

`(fork/pipe+` *conns [thunk]*`)` $\longrightarrow$ *pid or* `#f` procedure
`(%fork/pipe+` *conns [thunk]*`)` $\longrightarrow$ *pid or* `#f` procedure

Like `fork/pipe`, but the pipe connections between the child and parent are specified by the connection list *conns*. See the

`(|+` *conns* $pf_1 \;\ldots\; pf_n$`)`

process form for a description of connection lists.

`(wait` *[pid]*`)` $\longrightarrow$ *status [pid]* procedure

Simply calling `(wait)` will wait for any child to die, then return the child's exit status and pid as multiple values.

With an argument, `(wait` *pid*`)` waits for that specific process, then returns its exit status as a single value.

If a candidate child has already exited but not yet been waited for, `wait` returns immediately.

> *Remark:* Describe the way that wait reaps defunct processes into the internal table. Document all the architected wait machinery.

When a child process dies, its parent can call the `wait` procedure to recover the exit status of the child. The exit status is a small integer that can be encodes information describing how the child terminated. The bit-level format of the exit status is not defined by POSIX (you must use the following three functions to decode one). However, if a child terminates normally with exit code 0, POSIX does require `wait` to return an exit status that is exactly zero. So `(zero?` *status*`)` is a correct way to test for non-error, normal termination.

`(status:exit-val` *status*`)` $\longrightarrow$ *integer or* `#f` procedure
`(status:stop-sig` *status*`)` $\longrightarrow$ *integer or* `#f` procedure
`(status:term-sig` *status*`)` $\longrightarrow$ *integer or* `#f` procedure

For a given status value produced by calling `wait`, exactly one of these routines will return a true value.

If the child process exited normally, `status:exit-val` returns the exit code for the child process (*i.e.*, the value the child passed to `exit` or returned from `main`). Otherwise, this function returns false.

If the child process was suspended by a signal, `status:stop-sig` returns the signal that suspended the child. Otherwise, this function returns false.

If the child process terminated abnormally, `status:term-sig` returns the signal that terminated the child. Otherwise, this function returns false.

`(call-terminally` *thunk*`)` $\longrightarrow$ *no return value* procedure `call-terminally` calls its thunk. When the thunk returns, the process exits. Although `call-terminally` could be implemented as

```
(λ (thunk) (thunk) (exit 0))
```

an implementation can take advantage of the fact that this procedure never returns. For example, the runtime can start with a fresh stack and also start with a fresh dynamic environment, where shadowed bindings are discarded. This can allow the old stack and dynamic environment to be collected (assuming this data is not reachable through some live continuation).

3.5 Process state

`(umask)` $\longrightarrow$ *fixnum* procedure
`(set-umask` *perms*`)` $\longrightarrow$ *undefined* procedure
`(with-umask*` *perms thunk*`)` $\longrightarrow$ *values of thunk* procedure
`(with-umask` *perms . body*`)` $\longrightarrow$ *values of body* syntax

The process' current umask is retrieved with `umask`, and set with `(set-umask` *perms*`)`. Calling `with-umask*` changes the umask to *perms* for the duration of the call to *thunk*. If the program throws out of *thunk* by invoking a continuation, the umask is reset to its external value. If the program throws back into *thunk* by calling a stored continuation, the umask is restored to the *perms* value. The special form `with-umask` is equivalent in effect to the procedure `with-umask*`, but does not require the programmer to explicitly wrap a `(λ () ...)` around the body of the code to be executed.

`(chdir` *[fname]*`)` $\longrightarrow$ *undefined* procedure
`(cwd)` $\longrightarrow$ *string* procedure
`(with-cwd*` *fname thunk*`)` $\longrightarrow$ *value(s) of thunk* procedure
`(with-cwd` *fname . body*`)` $\longrightarrow$ *value(s) of body* syntax

These forms manipulate the current working directory. The cwd can be changed with `chdir` (although in most cases, `with-cwd` is preferrable). If `chdir` is called with no arguments, it changes the cwd to the user's home directory. The `with-cwd*` procedure calls `thunk` with the cwd temporarily set to *fname*; when *thunk* returns, or is exited in a non-local fashion (*e.g.*, by raising an exception or by invoking a continuation), the cwd is returned to its original value. The special form `with-cwd` is simply syntactic sugar for `with-cwd*`.

`(pid)` ⟶ *fixnum* procedure
`(parent-pid)` ⟶ *fixnum* procedure
`(process-group` *[pid]*`)` ⟶ *fixnum* procedure
`(set-process-group` *[pid] pgrp*`)` ⟶ *undefined* procedure

`(pid)` and `(parent-pid)` retrieve the process id for the current process and its parent. If the OS supports process groups, a process' process group can be retrieved and set with `process-group` and `set-process-group`. The affected process for these two procedures defaults to the current process.

`(set-priority` *which who priority*`)` ⟶ *undefined* procedure
`(priority` *which who*`)` ⟶ *fixnum* procedure
`(nice` *[pid delta]*`)` ⟶ *undefined* procedure

These procedures set and access the priority of processes. I can't remember how `set-priority` and `priority` work, so no documentation, and besides, they aren't implemented yet, anyway.

`(user-login-name)` ⟶ *string* procedure
`(user-uid)` ⟶ *fixnum* procedure
`(user-effective-uid)` ⟶ *fixnum* procedure
`(user-gid)` ⟶ *fixnum* procedure
`(user-effective-gid)` ⟶ *fixnum* procedure
`(user-supplementary-gids)` ⟶ *fixnum list* procedure
`(set-uid` *uid*`)` ⟶ *undefined* procedure
`(set-gid` *gid*`)` ⟶ *undefined* procedure

These routines get and set the effective and real user and group ids. The `set-uid` and `set-gid` routines correspond to the POSIX `setuid()` and `setgid()` procedures.

`(process-times)` ⟶ *[fixnum fixnum fixnum fixnum]* procedure

Returns four values:
  user CPU time in clock-ticks
  system CPU time in clock-ticks
  user CPU time of all descendant processes
  system CPU time of all descendant processes

3.6 User and group db access

These procedures are used to access the user and group database (*e.g.*, the ones traditionally stored in `/etc/passwd` and `/etc/group`.)

`(user-info` *uid/name*`)` ⟶ *record* procedure

Return a `user-info` record giving the recorded information for a particular user:

```
(define-record user-info
  name uid gid home-dir shell)
```

The *uid/name* argument is either an integer uid or a string user-name.

`(->uid` *uid/name*`)` $\longrightarrow$ *fixnum* procedure
`(->username` *uid/name*`)` $\longrightarrow$ *string* procedure

These two procedures coerce integer uid's and user names to a particular form.

`(group-info` *gid/name*`)` $\longrightarrow$ *record* procedure

Return a `group-info` record giving the recorded information for a particular user:

```
(define-record group-info
  name gid members)
```

The *gid/name* argument is either an integer gid or a string user-name.

3.7 Accessing command-line arguments

`command-line-arguments` *string* list
`(command-line)` $\longrightarrow$ *string list* procedure

The list of strings `command-line-arguments` contains the arguments passed to the scsh process on the command line. Calling `(command-line)` returns the complete `argv` string list, including the program. So if we run a shell script

```
/usr/shivers/bin/myls -CF src
```

then `command-line-arguments` is

```
("-CF" "src")
```

and `(command-line)` returns

```
("/usr/shivers/bin/myls" "-CF" "src")
```

`command-line` returns a fresh list each time it is called. In this way, the programmer can get a fresh copy of the original argument list if `command-line-arguments` has been modified or is lexically shadowed.

(arg *arglist n [default]*) $\longrightarrow$ *string* procedure
(arg* *arglist n [default-thunk]*) $\longrightarrow$ *string* procedure
(argv *n [default]*) $\longrightarrow$ *string* procedure

These procedures are useful for accessing arguments from argument lists. arg returns the n^{th} element of *arglist*. The index is 1-based. If *n* is too large, *default* is returned; if no *default*, then an error is signaled.

arg* is similar, except that the *default-thunk* is called to generate the default value.

(argv *n*) is simply (arg (command-line) (+ *n* 1)). The +1 offset ensures that the two forms

```
(arg command-line-arguments n)
(argv n)
```

return the same argument (assuming the user has not rebound or modified command-line-arguments).

Example:

```
(if (null? command-line-arguments)
    (& (xterm -n ,host -title ,host
              -name ,(string-append "xterm_" host)))
    (let* ((progname (file-name-nondirectory (argv 1)))
           (title (string-append host ":" progname)))
      (& (xterm -n     ,title
                -title ,title
                -e     ,@command-line-arguments))))
```

A subtlety: there are two ways to invoke a scsh program. One is as a simple binary, the other is as an interpreted script via the Unix #! exec(2) feature. When a binary is running with scsh code, (command-line) returns exactly the command line. However, when the scsh interpreter is invoked with a scsh script specified on the command line, then the scsh startup code doctors the list returned by (command-line) to make the shell script itself be the program (*i.e.*, (argv 0)), instead of the string "scsh", or whatever the real (argv 0) value is. In addition, scsh will delete scsh-specific flags from the argument list. So if we have a shell script in file fullecho:

```
#!/usr/local/bin/scsh -s
!#
(for-each (λ (arg) (display arg) (display " "))
          (command-line))
```

and we run the program

```
fullecho hello world
```

the program will print out

```
fullecho hello world
```

not

```
/usr/local/bin/scsh -s fullecho hello world
```

This argument line processing ensures that if a scsh script is subsequently compiled into a standalone executable, that its semantics will be unchanged—the arglist processing is invariant. In effect, the

```
/usr/local/bin/scsh -s
```

is not part of the program; it's a specification for the machine to execute the program on, so it is not properly part of the program's argument list.

> *Remark:* The truth: The above discussion assumes some things that don't exist:
>
> - An implementation of scsh that allows scsh scripts to be compiled to native code binaries.
> - A native code binary implementation of the scsh interpreter.
>
> What there is right now is just the Scheme 48 virtual machine, invoked with a scsh heap image.

3.8 System parameters

`(maximum-fds)` ⟶ *fixnum* — procedure
`(page-size)` ⟶ *fixnum* — procedure
`(system-name)` ⟶ *string* — procedure

Only `system-name` is implemented.

3.9 Signal system

Signal numbers are bound to the variables `signal/hup`, `signal/int`, ...

`(signal-process` *pid sig*`)` ⟶ *undefined* — procedure
`(signal-procgroup` *prgrp sig*`)` ⟶ *undefined* — procedure

These two procedures send signals to a specific process, and all the processes in a specific process group, respectively.

I haven't done signal handlers yet. Should be straightforward: a mechanism to assign procedures to signals.

(itimer *???*) ⟶ *undefined* procedure
(pause-until-interrupt) ⟶ *undefined* procedure

(sleep *secs*) ⟶ *undefined* procedure

Sleeping is defined, but we don't offer a way to sleep for a more precise interval (*e.g.*, a microsecond timer), as this is not in POSIX.

3.10 Time

This time package, does not currently work with NeXTSTEP, as NeXTSTEP does not provide a Posix-compliant time interface that will successfully link.

Scsh's time system is fairly sophisticated, particularly with respect to its careful treatment of time zones. However, casual users shouldn't be intimidated; most of the complexity is optional, and defaulting all the optional arguments reduces the system to a simple interface.

3.10.1 Terminology

"UTC" and "UCT" stand for "universal coordinated time," which is the official name for what is colloquially referred to as "Greenwich Mean Time."

Posix allows a single time zone to specify *two* different offsets from UTC: one standard one, and one for "summer time." Summer time is frequently some sort of daylight savings time.

The scsh time package consistently uses this terminology: we never say "gmt" or "dst;" we always say "utc" and "summer time."

3.10.2 Basic data types

We have two types: *time* and *date*.

A *time* specifies an instant in the history of the universe. It is location and time-zone independent. A time is a real value giving the number of elapsed seconds since the Unix "epoch" (Midnight, January 1, 1970 UTC). Time values provide arbitrary time resolution, limited only by the number system of the underlying Scheme system.

A *date* is a name for an instant in time that is specified relative to some location/time-zone in the world, *e.g.*:
Friday October 31, 1994 3:47:21 pm EST.

Dates provide one-second resolution, and are expressed with the following record type:

```
(define-record date     ; A Posix tm struct
  seconds       ; Seconds after the minute [0-59]
  minute        ; Minutes after the hour [0-59]
  hour          ; Hours since midnight [0-23]
  month-day     ; Day of the month [1-31]
  month         ; Months since January [0-11]
  year          ; Years since 1900
  tz-name       ; Time-zone name: #f or a string.
  tz-secs       ; Time-zone offset: #f or an integer.
  summer?       ; Summer (Daylight Savings) time in effect?
  week-day      ; Days since Sunday [0-6]
  year-day)     ; Days since Jan. 1 [0-365]
```

If the `tz-secs` field is given, it specifies the time-zone's offset from UTC in seconds. If it is specified, the `tz-name` and `summer?` fields are ignored when using the date structure to determine a specific instant in time.

If the `tz-name` field is given, it is a time-zone string such as `"EST"` or `"HKT"` understood by the OS. Since Posix time-zone strings can specify dual standard/summer time-zones (e.g., "EST5EDT" specifies U.S. Eastern Standard/Eastern Daylight Time), the value of the `summer?` field is used to resolve the amiguous boundary cases. For example, on the morning of the Fall daylight savings change-over, 1:00am–2:00am happens twice. Hence the date 1:30 am on this morning can specify two different seconds; the `summer?` flag says which one.

A date with `tz-name` = `tz-secs` = `#f` is a date that is specified in terms of the system's current time zone.

There is redundancy in the `date` data structure. For example, the `year-day` field is redundant with the `month-day` and `month` fields. Either of these implies the values of the `week-day` field. The `summer?` and `tz-name` fields are redundant with the `tz-secs` field in terms of specifying an instant in time. This redundancy is provided because consumers of dates may want it broken out in different ways. The scsh procedures that produce date records fill them out completely. However, when date records produced by the programmer are passed to scsh procedures, the redundancy is resolved by ignoring some of the secondary fields. This is described for each procedure below.

`(make-date` *s min h mday mon y [tzn tzs summ? wday yday]*`)` $\longrightarrow$ *date* procedure

When making a `date` record, the last five elements of the record are optional, and default to `#f`, `#f`, `#f`, 0, and 0 respectively. This is useful when creating a `date` record to pass as an argument to `time`.

3.10.3 Time zones

Several time procedures take time zones as arguments. When optional, the time zone defaults to local time zone. Otherwise the time zone can be one of:

`#f`	Local time
Integer	Seconds of offset from UTC. For example, New York City is -18000 (-5 hours), San Francisco is -28800 (-8 hours).
String	A Posix time zone string understood by the OS (*i.e.*., the sort of time zone assigned to the `$TZ` environment variable).

An integer time zone gives the number of seconds you must add to UTC to get time in that zone. It is *not* "seconds west" of UTC—that flips the sign.

To get UTC time, use a time zone of either 0 or `"UCT0"`.

3.10.4 Procedures

`(time+ticks)` $\longrightarrow$ *[secs ticks]* procedure
`(ticks/sec)` $\longrightarrow$ *real* procedure

The current time, with sub-second resolution. Sub-second resolution is not provided by Posix, but is available on many systems. The time is returned as elapsed seconds since the Unix epoch, plus a number of sub-second "ticks." The length of a tick may vary from implementation to implementation; it can be determined from `(ticks/sec)`.

The system clock is not required to report time at the full resolution given by `(ticks/sec)`. For example, on BSD, time is reported at 1μs resolution, so `(ticks/sec)` is 1,000,000. That doesn't mean the system clock has micro-second resolution.

If the OS does not support sub-second resolution, the *ticks* value is always 0, and `(ticks/sec)` returns 1.

> *Remark:* I chose to represent system clock resolution as ticks/sec instead of sec/tick to increase the odds that the value could be represented as an exact integer, increasing efficiency and making it easier for Scheme implementations that don't have sophisticated numeric support to deal with the quantity.
>
> You can convert seconds and ticks to seconds with the expression
>
> `(+` *secs* `(/` *ticks* `(ticks/sec)))`
>
> Given that, why not have the fine-grain time procedure just return a non-integer real for time? Following Common Lisp, I chose to allow the system clock to report sub-second time in its own units to lower the overhead of determining the time. This would be important for a

system that wanted to precisely time the duration of some event. Time stamps could be collected with little overhead, deferring the overhead of precisely calculating with them until after collection.

This is all a bit academic for the Scheme 48 implementation, where we determine time with a heavyweight system call, but it's nice to plan for the future.

`(date)` $\longrightarrow$ *date-record* — procedure
`(date` *[time tz]*`)` $\longrightarrow$ *date-record* — procedure

Simple `(date)` returns the current date, in the local time zone.

With the optional arguments, `date` converts the time to the date as specified by the time zone *tz*. *Time* defaults to the current time; *tz* defaults to local time, and is as described in the time-zone section.

If the *tz* argument is an integer, the date's `tz-name` field is a Posix time zone of the form "`UTC+`*hh*`:`*mm*`:`*ss*"; the trailing `:`*mm*`:`*ss* portion is deleted if it is zeroes.

`(time)` $\longrightarrow$ *integer* — procedure
`(time` *[date]*`)` $\longrightarrow$ *integer* — procedure

Simple `(time)` returns the current time.

With the optional date argument, `time` converts a date to a time. *Date* defaults to the current date.

Note that the input *date* record is overconstrained. `time` ignores *date*'s `week-day` and `year-day` fields. If the date's `tz-secs` field is set, the `tz-name` and `summer?` fields are ignored.

If the `tz-secs` field is `#f`, then the time-zone is taken from the `tz-name` field. A false `tz-name` means the system's current time zone. When calculating with time-zones, the date's `summer?` field is used to resolve ambiguities:

`#f`	Resolve an ambiguous time in favor of non-summer time.
true	Resolve an ambiguous time in favor of summer time.

This is useful in boundary cases during the change-over. For example, in the Fall, when US daylight savings time changes over at 2:00 am, 1:30 am happens twice—it names two instants in time, an hour apart.

Outside of these boundary cases, the `summer?` flag is ignored. For example, if the standard/summer change-overs happen in the Fall and the Spring, then the value of `summer?` is ignored for a January or July date. A January date would be resolved with standard time, and a July date with summer time, regardless of the `summer?` value.

The `summer?` flag is also ignored if the time zone doesn't have a summer time—for example, simple UTC.

(`date->string` *date*) ⟶ *string* procedure
(`format-date` *fmt date*) ⟶ *string* procedure

`Date->string` formats the date as a 24-character string of the form:

Sun Sep 16 01:03:52 1973

`Format-date` formats the date according to the format string *fmt*. The format string is copied verbatim, except that tilde characters indicate conversion specifiers that are replaced by fields from the date record. Figure 3.1 gives the full set of conversion specifiers supported by `format-date`.

(`fill-in-date!` *date*) ⟶ *date* procedure

This procedure fills in missing, redundant slots in a date record. In decreasing order of priority:

- **year, month, month-day ⇒ year-day**
  If the `year`, `month`, and `month-day` fields are all defined (are all integers), the `year-day` field is set to the corresponding value.
- **year, year-day ⇒ month, month-day**
  If the `month` and `month-day` fields aren't set, but the `year` and `year-day` fields are set, then `month` and `month-day` are calculated.
- **year, month, month-day, year-day ⇒ week-day**
  If either of the above rules is able to determine what day it is, the `week-day` field is then set.
- **tz-secs ⇒ tz-name**
  If `tz-secs` is defined, but `tz-name` is not, it is assigned a time-zone name of the form "`UTC+`*hh*`:`*mm*`:`*ss*"; the trailing `:`*mm*`:`*ss* portion is deleted if it is zeroes.
- **tz-name, date, summer? ⇒ tz-secs, summer?**
  If the date information is provided up to second resolution, `tz-name` is also provided, and `tz-secs` is not set, then `tz-secs` and `summer?` are set to their correct values. Summer-time ambiguities are resolved using the original value of `summer?`. If the time zone doesn't have a summer time variant, then `summer?` is set to `#f`.
- **local time, date, summer? ⇒ tz-name, tz-secs, summer?**
  If the date information is provided up to second resolution, but no time zone information is provided (both `tz-name` and `tz-secs` aren't set), then we proceed as in the above case, except the system's current time zone is used.

These rules allow one particular ambiguity to escape: if both `tz-name` and `tz-secs` are set, they are not brought into agreement. It isn't clear how to do this, nor is it clear which one should take precedence.

Oops: `fill-in-date!` isn't implemented yet.

`~~`	Converted to the ~ character.
`~a`	abbreviated weekday name
`~A`	full weekday name
`~b`	abbreviated month name
`~B`	full month name
`~c`	time and date using the time and date representation for the locale (`~X ~x`)
`~d`	day of the month as a decimal number (01-31)
`~H`	hour based on a 24-hour clock as a decimal number (00-23)
`~I`	hour based on a 12-hour clock as a decimal number (01-12)
`~j`	day of the year as a decimal number (001-366)
`~m`	month as a decimal number (01-12)
`~M`	minute as a decimal number (00-59)
`~p`	AM/PM designation associated with a 12-hour clock
`~S`	second as a decimal number (00-61)
`~U`	week number of the year; Sunday is first day of week (00-53)
`~w`	weekday as a decimal number (0-6), where Sunday is 0
`~W`	week number of the year; Monday is first day of week (00-53)
`~x`	date using the date representation for the locale
`~X`	time using the time representation for the locale
`~y`	year without century (00-99)
`~Y`	year with century (*e.g.*1990)
`~Z`	time zone name or abbreviation, or no characters if no time zone is determinable

Figure 3.1: `format-date` conversion specifiers

3.11 Environment variables

`(setenv` *var val*`)` ⟶ *undefined* procedure
`(getenv` *var*`)` ⟶ *string* procedure

These functions get and set the process environment, stored in the external C variable `char **environ`. An environment variable *var* is a string. If an environment variable is set to a string *val*, then the process' global environment structure is altered with an entry of the form "*var*=*val*". If *val* is `#f`, then any entry for *var* is deleted.

`(env->alist)` ⟶ *string*→*string alist* procedure

The `env->alist` procedure converts the entire environment into an alist, *e.g.*,

```
(("TERM" . "vt100")
 ("SHELL" . "/bin/csh")
 ("EDITOR" . "emacs")
 ...)
```

`(alist->env` *alist*`)` ⟶ *undefined* procedure

Alist must be an alist whose keys are all strings, and whose values are all either strings or string lists. String lists are converted to colon lists (see below). The alist is installed as the current Unix environment (*i.e.*, converted to a null-terminated C vector of "*var*=*val*" strings which is assigned to the global `char **environ`).

The following three functions help the programmer manipulate alist tables in some generally useful ways. They are all defined using `equal?` for key comparison.

`(alist-delete` *key alist*`)` ⟶ *alist* procedure

Delete any entry labelled by value *key*.

`(alist-update` *key val alist*`)` ⟶ *alist* procedure

Delete *key* from *alist*, then cons on a `(`*key* `.` *val*`)` entry.

`(alist-compress` *alist*`)` ⟶ *alist* procedure

Compresses *alist* by removing shadowed entries. Example:

```
;;; Shadowed (1 . c) entry removed.
(alist-compress '( (1 . a) (2 . b) (1 . c) (3 . d) ))
    ⟹  ((1 . a) (2 . b) (3 . d))
```

(`with-env*` *env-alist-delta thunk*) ⟶ *value(s) of thunk* — procedure
(`with-total-env*` *env-alist thunk*) ⟶ *value(s) of thunk* — procedure

These procedures call *thunk* in the context of an altered environment. They return whatever values *thunk* returns. Non-local returns restore the environment to its outer value; throwing back into the thunk by invoking a stored continuation restores the environment back to its inner value.

The *env-alist-delta* argument specifies a *modification* to the current environment—*thunk*'s environment is the original environment overridden with the bindings specified by the alist delta.

The *env-alist* argument specifies a complete environment that is installed for *thunk*.

(`with-env` *env-alist-delta . body*) ⟶ *value(s) of body* — syntax
(`with-total-env` *env-alist . body*) ⟶ *value(s) of body* — syntax

These special forms provide syntactic sugar for `with-env*` and `with-total-env*`. The env alists are not evaluated positions, but are implicitly backquoted. In this way, they tend to resemble binding lists for `let` and `let*` forms.

Example: These four pieces of code all run the mailer with special `$TERM` and `$EDITOR` values.

```
(with-env (("TERM" . "xterm") ("EDITOR" . ,my-editor))
  (run (mail shivers@lcs.mit.edu)))

(with-env* `(("TERM" . "xterm") ("EDITOR" . ,my-editor))
  (λ () (run (mail shivers@csd.hku.hk))))

(run (begin (setenv "TERM" "xterm")      ; Env mutation happens
            (setenv "EDITOR" my-editor) ; in the subshell.
            (exec-epf (mail shivers@research.att.com))))

;; In this example, we compute an alternate environment ENV2
;; as an alist, and install it with an explicit call to the
;; EXEC-PATH/ENV procedure.
(let* ((env (env->alist))           ; Get the current environment,
       (env1 (alist-update env  "TERM" "xterm"))      ; and compute
       (env2 (alist-update env1 "EDITOR" my-editor))) ; the new env.
  (run (begin (exec-path/env "mail" env2 "shivers@cs.cmu.edu"))))
```

3.11.1 Path lists and colon lists

Environment variables such as `$PATH` encode a list of strings by separating the list elements with colon delimiters. Once parsed into actual lists, these ordered lists can be manipulated with the following two functions. To convert between the colon-separated string encoding and the list-of-strings representation, see the `field-reader` and `join-strings` functions in section 6.1.

> *Remark:* An earlier release of scsh provided the `split-colon-list` and `string-list->colon-list` functions. These have been removed from scsh, and are replaced by the more general parsers and unparsers of the field-reader module.

(`add-before` *elt before list*) $\longrightarrow$ *list* — procedure
(`add-after` *elt after list*) $\longrightarrow$ *list* — procedure

These functions are for modifying search-path lists, where element order is significant.

`add-before` adds *elt* to the list immediately before the first occurrence of *before* in the list. If *before* is not in the list, *elt* is added to the end of the list.

`add-after` is similar: *elt* is added after the last occurrence of *after*. If *after* is not found, *elt* is added to the beginning of the list.

Neither function destructively alters the original path-list. The result may share structure with the original list. Both functions use `equal?` for comparing elements.

3.11.2 `$USER`, `$HOME`, and `$PATH`

Like sh and unlike csh, scsh has *no* interactive dependencies on environment variables. It does, however, initialise certain internal values at startup time from the initial process environment, in particular `$HOME` and `$PATH`. Scsh never uses `$USER` at all. It computes `(user-login-name)` from the system call `(user-uid)`.

`home-directory` — *string*
`exec-path-list` — *string* list

Scsh accesses `$HOME` at start-up time, and stores the value in the global variable `home-directory`. It uses this value for ~ lookups and for returning to home on `(chdir)`.

Scsh accesses `$PATH` at start-up time, colon-splits the path list, and stores the value in the global variable `exec-path-list`. This list is used for `exec-path` and `exec-path/env` searches.

Chapter 4

Networking

The Scheme Shell provides a BSD-style sockets interface. There is not an official standard for a network interface for scsh to adopt (this is the subject of the forthcoming Posix.8 standard). However, Berkeley sockets are a *de facto* standard, being found on most Unix workstations and PC operating systems.

Future releases of scsh will contain more high-level support for networking applications. We have Scheme implementations for the ftp, telnet, smtp, finger, and http protocols, as well as an html parser. When this code is included in a future release, this chapter will describe the interfaces. We are also contemplating a tail-recursive RPC mechanism, but have done no development work.

4.1 High-level interface

For convenience, and too avoid some of the messy details of the socket interface, we provide a high level socket interface. These routines attempt to make it easy to write simple clients and servers without having to think of many of the details of initiating socket connections. We welcome suggested improvements to this interface, including better names, which right now are solely descriptions of the procedure's action.. This might be fine for people who already understand sockets, but does not help the new networking programmer.

(`socket-connect` *protocol-family socket-type . args*) $\longrightarrow$ *socket* procedure

> `socket-connect` is intended for creating client applications. *protocol-family* is specified as either the `protocol-family/internet` or `protocol-family/unix`. *socket-type* is specified as either `socket-type/stream` or `socket-type/datagram`. See `socket` for a more complete description of these terms.

The variable *args* list is meant to specify protocol family specific information. For Internet sockets, this consists of two arguments: a host name and a port number. For Unix sockets, this consists of a pathname.

`socket-connect` returns a `socket` which can be used for input and output from a remote server. See `socket` for a description of the *socket record*.

(`bind-listen-accept-loop` *protocol-family proc arg*) ⟶ *does-not-return* procedure

`bind-listen-accept-loop` is intended for creating server applications. *protocol-family* is specified as either the `protocol-family/internet` or `protocol-family/unix`. *proc* is a procedure of two arguments: a `socket` and a socket-address. *arg* specifies a port number for Internet sockets or a pathname for Unix sockets. See `socket` for a more complete description of these terms.

proc is called with a socket and a socket address each time there is a connection from a client application. The socket allows communications with the client. The socket address specifies the address of the remote client.

This procedure does not return, but loops indefinitely accepting connections from client programs.

4.2 Sockets

(`create-socket` *protocol-family type [protocol]*) ⟶ *socket* procedure
(`create-socket-pair` *type*) ⟶ *[socket$_1$ socket$_2$]* procedure
(`close-socket` *socket*) ⟶ *undefined* procedure

A socket is one end of a network connection. Three specific properties of sockets are specified at creation time: the protocol-family, type, and protocol.

The *protocol-family* specifies the protocol family to be used with the socket. This also determines the address family of socket addresses, which are described in more detail below. Scsh currently supports the Unix internal protocols and the Internet protocols using the following constants:

```
protocol-family/unspecified
protocol-family/unix
protocol-family/internet
```

The *type* specifies the style of communication. Examples that your operating system probably provides are stream and datagram sockets. Others maybe available depending on your system. Typical values are:

```
socket-type/stream
socket-type/datagram
socket-type/raw
```

The *protocol* specifies a particular protocol to use within a protocol family and type. Usually only one choice exists, but it's probably safest to set this explicitly. See the protocol database routines for information on looking up protocol constants.

New sockets are typically created with `create-socket`. However, `create-socket-pair` can also be used to create a pair of connected sockets in the `protocol-family/unix` protocol-family. The value of a returned socket is a *socket record*, defined to have the following structure:

```
(define-record socket
  family                                  ; protocol family
  inport                                  ; input-port
  outport)                                ; output-port
```

The `family` specifies the protocol family of the socket. The `inport` and `outport` fields are ports that can be used for input and output, respectively. For a stream socket, they are only usable after a connection has been established via `connect-socket` or `accept-connection`. For a datagram socket, *outport* can be immediately using `send-message`, and *inport* can be used after `bind` has created a local address.

`close-socket` provides a convenient way to close a socket's port. It is preferred to explicitly closing the inport and outport because using `close` on sockets is not currently portable across operating systems.

4.3 Socket addresses

The format of a socket-address depends on the address family of the socket. Address-family-specific routines are provided to convert protocol-specific addresses to socket addresses. The value returned by these routines is a *socket-address record*, defined to have the following visible structure:

```
(define-record socket-address
  family)                                 ; address family
```

The `family` is one of the following constants:

```
address-family/unspecified
address-family/unix
address-family/internet
```

(`unix-address->socket-address` *pathname*) $\longrightarrow$ *socket-address* procedure

> `unix-address->socket-address` returns a *socket-address* based on the string *pathname*. There is a system dependent limit on the length of *pathname*.

(`internet-address->socket-address` *host-address service-port*) $\longrightarrow$ *socket-address* procedure

> `internet-address->socket-address` returns a *socket-address* based on an integer *host-address* and an integer *service-port*. Besides being a 32-bit host address, an Internet host address can also be one of the following constants:
>
> ```
> internet-address/any
> internet-address/loopback
> internet-address/broadcast
> ```
>
> The use of `internet-address/any` is described below in `bind-socket`. `internet-address/loopback` is an address that always specifies the local machine. `internet-address/broadcast` is used for network broadcast communications.
>
> For information on obtaining a host's address, see the `host-info` function.

(`socket-address->unix-address` *socket-address*) $\longrightarrow$ *pathname* procedure
(`socket-address->internet-address` *socket-address*) $\longrightarrow$ *[host-address service-port]* procedure

> The routines `socket-address->internet-address` and `socket-address->unix-address` return the address-family-specific addresses. Be aware that most implementations don't correctly return anything more than an empty string for addresses in the Unix address-family.

4.4 Socket primitives

The procedures in this section are presented in the order in which a typical program will use them. Consult a text on network systems programming for more information on sockets. [1] The last two tutorials are freely available as part of

[1] Some recommended ones are:

- "Unix Network Programming" by W. Richard Stevens
- "An Introductory 4.3BSD Interprocess Communication Tutorial." (reprinted in UNIX Programmer's Supplementary Documents Volume 1, PS1:7)
- "An Advanced 4.3BSD Interprocess Communication Tutorial." (reprinted in UNIX Programmer's Supplementary Documents Volume 1, PS1:8)

BSD. In the absence of these, your Unix manual pages for socket might be a good starting point for information.

(`connect-socket` *socket socket-address*) $\longrightarrow$ *undefined* procedure

`connect-socket` sets up a connection from a *socket* to a remote *socket-address*. A connection has different meanings depending on the socket type. A stream socket must be connected before use. A datagram socket can be connected multiple times, but need not be connected at all if the remote address is specified with each `send-message`, described below. Also, datagram sockets may be disassociated from a remote address by connecting to a null remote address.

(`bind-socket` *socket socket-address*) $\longrightarrow$ *undefined* procedure

`bind-socket` assigns a certain local *socket-address* to a *socket*. Binding a socket reserves the local address. To receive connections after binding the socket, use `listen-socket` for stream sockets and `receive-message` for datagram sockets.

Binding an Internet socket with a host address of `internet-address/any` indicates that the caller does not care to specify from which local network interface connections are received. Binding an Internet socket with a service port number of zero indicates that the caller has no preference as to the port number assigned.

Binding a socket in the Unix address family creates a socket special file in the file system that must be deleted before the address can be reused. See `delete-file`.

(`listen-socket` *socket backlog*) $\longrightarrow$ *undefined* procedure

`listen-socket` allows a stream *socket* to start receiving connections, allowing a queue of up to *backlog* connection requests. Queued connections may be accepted by `accept-connection`.

(`accept-connection` *socket*) $\longrightarrow$ *[new-socket socket-address]* procedure

`accept-connection` receives a connection on a *socket*, returning a new socket that can be used for this connection and the remote socket address associated with the connection.

(`socket-local-address` *socket*) $\longrightarrow$ *socket-address* procedure
(`socket-remote-address` *socket*) $\longrightarrow$ *socket-address* procedure

Sockets can be associated with a local address or a remote address or both. `socket-local-address` returns the local *socket-address* record associated with *socket*. `socket-remote-address` returns the remote *socket-address* record associated with *socket*.

(`shutdown-socket` *socket how-to*) ⟶ *undefined* procedure

`shutdown-socket` shuts down part of a full-duplex socket. The method of shutting done is specified by the *how-to* argument, one of:

```
shutdown/receives
shutdown/sends
shutdown/sends+receives
```

4.5 Performing input and output on sockets

(`receive-message` *socket length [flags]*) ⟶ *[string-or-*`#f`socket-address] procedure
(`receive-message!` *socket string [start] [end] [flags]*) ⟶ *[count-or-*`#f`socket-address] procedure
(`receive-message/partial` *socket length [flags]*) ⟶ *[string-or-*`#f`socket-address] procedure
(`receive-message!/partial` *socket string [start] [end] [flags]*) ⟶ *[count-or-*`#f`socket-address] procedu

(`send-message` *socket string [start] [end] [flags] [socket-address]*) ⟶ *undefined* procedure
(`send-message/partial` *socket string [start] [end] [flags] [socket-address]*) ⟶ *count* procedure

For most uses, standard input and output routines such as `read-string` and `write-string` should suffice. However, in some cases an extended interface is required. The `receive-message` and `send-message` calls parallel the `read-string` and `write-string` calls with a similar naming scheme.

One additional feature of these routines is that `receive-message` returns the remote *socket-address* and *send-message* takes an optional remote `socket-address`. This allows a program to know the source of input from a datagram socket and to use a datagram socket for output without first connecting it.

All of these procedures take an optional *flags* field. This argument is an integer bit-mask, composed by or'ing together the following constants:

```
message/out-of-band
message/peek
message/dont-route
```

See `read-string` and `write-string` for a more detailed description of the arguments and return values.

4.6 Socket options

(`socket-option` *socket level option*) $\longrightarrow$ *value* procedure
(`set-socket-option` *socket level option value*) $\longrightarrow$ *undefined* procedure

`socket-option` and `set-socket-option` allow the inspection and modification, respectively, of several options available on sockets. The *level* argument specifies what protocol level is to be examined or affected. A level of `level/socket` specifies the highest possible level that is available on all socket types. A specific protocol number can also be used as provided by `protocol-info`, described below.

There are several different classes of socket options. The first class consists of boolean options which can be either true or false. Examples of this option type are:

```
socket/debug
socket/accept-connect
socket/reuse-address
socket/keep-alive
socket/dont-route
socket/broadcast
socket/use-loop-back
socket/oob-inline
socket/use-privileged
socket/cant-signal
tcp/no-delay
```

Value options are another category of socket options. Options of this type are an integer value. Examples of this option type are:

```
socket/send-buffer
socket/receive-buffer
socket/send-low-water
socket/receive-low-water
socket/error
socket/type
ip/time-to-live
tcp/max-segment
```

A third option type specifies how long for data to linger after a socket has been closed. There is only one option of this type: `socket/linger`. It is set with either `#f`to disable it or an integer number of seconds to linger and returns a value of the same type upon inspection.

The fourth and final option type of this time is a timeout option. There are two examples of this option type: `socket/send-timeout` and `socket/receive-timeout`. These are set with a real number of microseconds resolution and returns a value of the same type upon inspection.

4.7 Database-information entries

(`host-info` *name-or-socket-address*) ⟶ *host-info* procedure
(`network-info` *name-or-socket-address*) ⟶ *network-info* procedure
(`service-info` *name-or-number [protocol-name]*) ⟶ *service-info* procedure
(`protocol-info` *name-or-number*) ⟶ *protocol-info* procedure

`host-info` allows a program to look up a host entry based on either its string *name* or *socket-address*. The value returned by this routine is a *host-info record*, defined to have the following structure:

```
(define-record host-info
  name                                        ; Host name
  aliases                                     ; Alternative names
  addresses)                                  ; Host addresses
```

`host-info` could fail and raise an error for one of the following reasons:

```
herror/host-not-found
herror/try-again
herror/no-recovery
herror/no-data
herror/no-address
```

`network-info` allows a program to look up a network entry based on either its string *name* or *socket-address*. The value returned by this routine is a *network-info record*, defined to have the following structure:

```
(define-record network-info
  name                                        ; Network name
  aliases                                     ; Alternative names
  net)                                        ; Network number
```

`service-info` allows a program to look up a service entry based on either its string *name* or integer *port*. The value returned by this routine is a *service-info record*, defined to have the following structure:

```
(define-record service-info
  name                                        ; Service name
  aliases                                     ; Alternative names
  port                                        ; Port number
  protocol)                                   ; Protocol name
```

`protocol-info` allows a program to look up a protocol entry based on either its string *name* or integer *number*. The value returned by this routine is a *protocol-info record*, defined to have the following structure:

```
(define-record protocol-info
  name                                    ; Protocol name
  aliases                                 ; Alternative names
  number)                                 ; Protocol number)
```

Chapter 5

Strings and characters

Scsh provides a set of procedures for processing strings and characters. The procedures provided match regular expressions, search strings, parse file-names, and manipulate sets of characters.

Also see chapter 6 on record I/O, field parsing, and the awk loop. The procedures documented there allow you to read character-delimited records from ports, use regular expressions to split the records into fields (for example, splitting a string at every occurrence of colon or white-space), and loop over streams of these records in a convenient way.

5.1 String manipulation

Strings are the basic communication medium for Unix processes, so a shell language must have reasonable facilities for manipulating them.

5.1.1 Regular expressions

The following functions perform regular expression matching. The code uses Henry Spencer's regular expression package.

(`string-match` *regexp string [start]*) $\longrightarrow$ *match or false* procedure

> Search *string* starting at position *start*, looking for a match for *regexp*. If a match is found, return a match structure describing the match, otherwise `#f`. *Start* defaults to 0.

(`regexp-match?` *obj*) $\longrightarrow$ *boolean* procedure

Is the object a regular expression match?

(`match:start` *match [match-number]*) $\longrightarrow$ *fixnum* procedure

Returns the start position of the match denoted by *match-number* The whole regexp is 0. Each further number represents positions enclosed by `(...)` sections. *Match-number* defaults to 0.

(`match:end` *match [match-number]*) $\longrightarrow$ *fixnum* procedure

Returns the end position of the match denoted by *match-number*. *Match-number* defaults to 0 (the whole match).

(`match:substring` *match [match-number]*) $\longrightarrow$ *string* procedure

Returns the substring matched by match *match-number*. *Match-number* defaults to 0 (the whole match).

Remark: What do these guys do when there is no match corresponding to *match-number*? Return `#f` or signal error? `#f` probably best.

Regular expression matching compiles patterns into special data structures which can be efficiently used to match against strings. The overhead of compiling patterns that will be used for multiple searches can be avoided by these lower-level routines:

(`make-regexp` *str*) $\longrightarrow$ *re* procedure

Generate a compiled regular expression from the given string.

(`regexp?` *obj*) $\longrightarrow$ *boolean* procedure

Is the object a regular expression?

(`regexp-exec` *regexp str [start]*) $\longrightarrow$ *match or false* procedure

Apply the regular expression *regexp* to the string *str* starting at position *start*. If the match succeeds it returns a regexp-match, otherwise `#f`. *Start* defaults to 0.

Remark: The truth: S48 doesn't have the facilities for extending the garbage collector to malloc'd C storage (unlike elk). So we do not really export regular expression compilation. What we currently do is this:

```
(define regexp? string?)
(define (make-regexp str) str)
(define (regexp-exec regexp str [start])
  (string-match regexp str [start]))
```

This could be improved upon in another implementation (like elk).

(`regexp-quote` *str*) $\longrightarrow$ *string* procedure

Returns a regular expression that matches the string *str* exactly. In other words, it quotes the regular expression, prepending backslashes to all the special regexp characters in *str*.

```
(regexp-quote "*Hello* world.")
    ⟹ "\*Hello\* world\."
```

Oops: Scsh regex matching doesn't currently flag un-matched subexpressions in the `match:begin`, `match:end`, and `match:substring` functions. This needs to be fixed.

5.1.2 Other string manipulation facilities

(`index` *string char [start]*) $\longrightarrow$ *fixnum or false* procedure
(`rindex` *string char [start]*) $\longrightarrow$ *fixnum or false* procedure

These procedures search through *string* looking for an occurrence of character *char*. `index` searches left-to-right; `rindex` searches right-to-left.

`index` returns the smallest index i of *string* greater than or equal to *start* such that *string*$[i]$ = *char*. The default for *start* is zero. If there is no such match, `index` returns false.

`rindex` returns the largest index i of *string* less than *start* such that *string*$[i]$ = *char*. The default for *start* is (`string-length` *string*). If there is no such match, `rindex` returns false.

I should probably snarf all the MIT Scheme string functions, and stick them in a package. Unix programs need to mung character strings a lot.

MIT string match commands:

```
[sub]string-match-forward,backward[-ci]
[sub]string-prefix,suffix[-ci]?
[sub]string-find-next,previous-char[-ci]
[sub]string-find-next,previous-char-in-set
[sub]string-replace[!]
...etc.
```

These are not currently provided.

(`substitute-env-vars` *fname*) $\longrightarrow$ *string* procedure

Replace occurrences of environment variables with their values. An environment variable is denoted by a dollar sign followed by alphanumeric chars and underscores, or is surrounded by braces.

```
(substitute-env-vars "$USER/.login")
    ⟹  "shivers/.login"
(substitute-env-vars "${USER}_log")  ⟹  "shivers_log"
```

5.1.3 Manipulating file-names

These procedures do not access the file-system at all; they merely operate on file-name strings. Much of this structure is patterned after the gnu emacs design. Perhaps a more sophisticated system would be better, something like the pathname abstractions of COMMON LISP or MIT Scheme. However, being Unix-specific, we can be a little less general.

Terminology

These procedures carefully adhere to the POSIX standard for file-name resolution, which occasionally entails some slightly odd things. This section will describe these rules, and give some basic terminology.

A *file-name* is either the file-system root ("/"), or a series of slash-terminated directory components, followed by a a file component. Root is the only file-name that may end in slash. Some examples:

File name	Dir components	File component
`src/des/main.c`	`("src" "des")`	`"main.c"`
`/src/des/main.c`	`("" "src" "des")`	`"main.c"`
`main.c`	`()`	`"main.c"`

Note that the relative filename `src/des/main.c` and the absolute filename `/src/des/main.c` are distinguished by the presence of the root component `""` in the absolute path.

Multiple embedded slashes within a path have the same meaning as a single slash. More than two leading slashes at the beginning of a path have the same meaning as a single leading slash—they indicate that the file-name is an absolute one, with the path leading from root. However, POSIX permits the OS to give special meaning to *two* leading slashes. For this reason, the routines in this section do not simplify two leading slashes to a single slash.

A file-name in *directory form* is either a file-name terminated by a slash, *e.g.*, "`/src/des/`", or the empty string, "". The empty string corresponds to the current working directory, who's file-name is dot ("`.`"). Working backwards from the append-a-slash rule, we extend the syntax of POSIX file-names to define the empty string to be a file-name form of the root directory "`/`". (However, "`/`" is also acceptable as a file-name form for root.) So the empty string has two interpretations: as a file-name form, it is the file-system root; as a directory form, it is the current working directory. Slash is also an ambiguous form: `/` is both a directory-form and a file-name form.

The directory form of a file-name is very rarely used. Almost all of the procedures in scsh name directories by giving their file-name form (without the trailing slash), not their directory form. So, you say "`/usr/include`", and "`.`", not "`/usr/include/`" and "". The sole exceptions are `file-name-as-directory` and `directory-as-file-name`, whose jobs are to convert back-and-forth between these forms, and `file-name-directory`, whose job it is to split out the directory portion of a file-name. However, most procedures that expect a directory argument will coerce a file-name in directory form to file-name form if it does not have a trailing slash. Bear in mind that the ambiguous case, empty string, will be interpreted in file-name form, *i.e.*, as root.

Procedures

`(file-name-as-directory` *fname*`)` $\longrightarrow$ *string* procedure

Convert a file-name to directory form. Basically, add a trailing slash if needed:

```
(file-name-as-directory "src/des")   ==>  "src/des/"
(file-name-as-directory "src/des/")  ==>  "src/des/"
```

`.`, `/`, and `""` are special:

```
(file-name-as-directory ".")  ==>  ""
(file-name-as-directory "/")  ==>  "/"
(file-name-as-directory "")   ==>  "/"
```

`(directory-as-file-name` *fname*`)` $\longrightarrow$ *string* procedure

Convert a directory to a simple file-name. Basically, kill a trailing slash if one is present:

```
(directory-as-file-name "foo/bar/")  ==>  "foo/bar"
```

`/` and `""` are special:

```
(directory-as-file-name "/")  ==>  "/"
(directory-as-file-name "")   ==>  "." (i.e., the cwd)
```

`(file-name-absolute?` *fname*`)` $\longrightarrow$ *boolean* procedure

Does *fname* begin with a root or ˜ component? (Recognising ˜ as a home-directory specification is an extension of POSIX rules.)

```
(file-name-absolute? "/usr/shivers")  ⟹  #t
(file-name-absolute? "src/des")       ⟹  #f
(file-name-absolute? "~/src/des")     ⟹  #t
```

Non-obvious case:

```
(file-name-absolute? "")              ⟹  #t (i.e., root)
```

`(file-name-directory` *fname*`)` ⟶ *string or false* procedure

Return the directory component of *fname* in directory form. If the file-name is already in directory form, return it as-is.

```
(file-name-directory "/usr/bdc")     ⟹  "/usr/"
(file-name-directory "/usr/bdc/")    ⟹  "/usr/bdc/"
(file-name-directory "bdc/.login")   ⟹  "bdc/"
(file-name-directory "main.c")       ⟹  ""
```

Root has no directory component:

```
(file-name-directory "/")            ⟹  ""
(file-name-directory "")             ⟹  ""
```

`(file-name-nondirectory` *fname*`)` ⟶ *string* procedure

Return non-directory component of fname.

```
(file-name-nondirectory "/usr/ian")     ⟹  "ian"
(file-name-nondirectory "/usr/ian/")    ⟹  ""
(file-name-nondirectory "ian/.login")   ⟹  ".login"
(file-name-nondirectory "main.c")       ⟹  "main.c"
(file-name-nondirectory "")             ⟹  ""
(file-name-nondirectory "/")            ⟹  "/"
```

`(split-file-name` *fname*`)` ⟶ *string list* procedure

Split a file-name into its components.

```
(split-file-name "src/des/main.c")
    ==>      ("src" "des" "main.c")

(split-file-name "/src/des/main.c")
    ==>   ("" "src" "des" "main.c")

(split-file-name "main.c")
    ==>                  ("main.c")

(split-file-name "/")
    ==>   ("")
```

(path-list->file-name *path-list [dir]*) $\longrightarrow$ *string* procedure

Inverse of `split-file-name`.

```
(path-list->file-name '("src" "des" "main.c"))
    ==>  "src/des/main.c"
(path-list->file-name '("" "src" "des" "main.c"))
    ==>  "/src/des/main.c"
```

Optional *dir* arg anchors relative path-lists:

```
(path-list->file-name '("src" "des" "main.c")
                      "/usr/shivers")
    ==>  "/usr/shivers/src/des/main.c"
```

The optional *dir* argument is usefully `(cwd)`.

(file-name-extension *fname*) $\longrightarrow$ *string* procedure

Return the file-name's extension.

```
(file-name-extension "main.c")        ==>  ".c"
(file-name-extension "main.c.old")    ==>  ".old"
(file-name-extension "/usr/shivers")  ==>  ""
```

Weird cases:

```
(file-name-extension "foo.")   ==>  "."
(file-name-extension "foo..")  ==>  "."
```

Dot files are not extensions:

```
(file-name-extension "/usr/shivers/.login")  ==>  ""
```

(file-name-sans-extension *fname*) $\longrightarrow$ *string* procedure

Return everything but the extension.

```
(file-name-sans-extension "main.c")      ==>  "main"
(file-name-sans-extension "main.c.old")  ==>  "main.c""
(file-name-sans-extension "/usr/shivers")
    ==>  "/usr/shivers"
```

Weird cases:

```
(file-name-sans-extension "foo.")   ==>  "foo"
(file-name-sans-extension "foo..")  ==>  "foo."
```

Dot files are not extensions:

```
(file-name-sans-extension "/usr/shivers/.login")
    ==>  "/usr/shivers/.login
```

Note that appending the results of `file-name-extension` and `file-name-sans-extension` in all cases produces the original file-name.

(`parse-file-name` *fname*) $\longrightarrow$ *[dir name extension]* procedure

Let *f* be (`file-name-nondirectory` *fname*). This function returns the three values:

- (`file-name-directory` *fname*)
- (`file-name-sans-extension` *f*))
- (`file-name-extension` *f*)

The inverse of `parse-file-name`, in all cases, is `string-append`. The boundary case of `/` was chosen to preserve this inverse.

(`replace-extension` *fname ext*) $\longrightarrow$ *string* procedure

This procedure replaces *fname*'s extension with *ext*. It is exactly equivalent to

(`string-append (file-name-sans-extension` *fname*) *ext*)

(`simplify-file-name` *fname*) $\longrightarrow$ *string* procedure

Removes leading and internal occurrences of dot. A trailing dot is left alone, as the parent could be a symlink. Removes internal and trailing double-slashes. A leading double-slash is left alone, in accordance with POSIX. However, triple and more leading slashes are reduced to a single slash, in accordance with POSIX. Double-dots (parent directory) are left alone, in case they come after symlinks or appear in a `/../`*machine*`/...` "super-root" form (which POSIX permits).

(`resolve-file-name` *fname [dir]*) $\longrightarrow$ *string* procedure

- Do ˜ expansion.
- If *dir* is given, convert a relative file-name to an absolute file-name, relative to directory *dir*.

(`expand-file-name` *fname [dir]*) $\longrightarrow$ *string* procedure

Resolve and simplify the file-name.

(`home-dir` *[user]*) $\longrightarrow$ *string* procedure

`home-dir` returns *user*'s home directory. *User* defaults to the current user.

```
(home-dir)            ==>  "/user1/lecturer/shivers"
(home-dir "ctkwan")   ==>  "/user0/research/ctkwan"
```

(`home-file` *[user] fname*) $\longrightarrow$ *string* procedure

Returns file-name *fname* relative to *user*'s home directory; *user* defaults to the current user.

```
(home-file "man")            ==>  "/usr/shivers/man"
(home-file "fcmlau" "man")   ==>  "/usr/fcmlau/man"
```

The general `substitute-env-vars` string procedure, defined in the previous section, is also frequently useful for expanding file-names.

5.2 ASCII encoding

(`char->ascii` *character*) $\longrightarrow$ *integer* procedure
(`ascii->char` *integer*) $\longrightarrow$ *character* procedure

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

5.3 Character sets

Scsh provides a `char-set` type for expressing sets of characters. These sets are used by some of the delimited input procedures (section 6.1). The character set package that scsh uses was taken from Project Mac's MIT Scheme.

(`char-set?` *x*) $\longrightarrow$ *boolean* procedure

Returns true if the object `x` is a character set.

5.3.1 Creating character sets

(`char-set` $char_1 \ldots$) $\longrightarrow$ *char-set* procedure

Return a character set containing the given characters.

(`chars->char-set` *chars*) $\longrightarrow$ *char-set* procedure

Return a character set containing the characters in the list *chars*.

(`string->char-set` *s*) $\longrightarrow$ *char-set* procedure

Return a character set containing the characters in the string *s*.

(`predicate->char-set` *pred*) $\longrightarrow$ *char-set* procedure

Returns a character set containing every character *c* such that (*pred c*) returns true.

(`ascii-range->char-set` *lower upper*) $\longrightarrow$ *char-set* procedure

Returns a character set containing every character whose ASCII code lies in the range [*lower*, *upper*] inclusive.

5.3.2 Querying character sets

(`char-set-members` *char-set*) $\longrightarrow$ *character-list* procedure

This procedure returns a list of the members of *char-set*.

(`char-set-member?` *char char-set*) $\longrightarrow$ *boolean* procedure

This procedure tests *char* for membership in set *char-set*.

5.3.3 Character set algebra

(`char-set-invert` *char-set*) $\longrightarrow$ *char-set* procedure
(`char-set-union` $char\text{-}set_1$ $char\text{-}set_2$) $\longrightarrow$ *char-set* procedure
(`char-set-intersection` $char\text{-}set_1$ $char\text{-}set_2$) $\longrightarrow$ *char-set* procedure
(`char-set-difference` $char\text{-}set_1$ $char\text{-}set_2$) $\longrightarrow$ *char-set* procedure

These procedures implement set complement, union, intersection, and difference for character sets.

5.3.4 Standard character sets

Several character sets are predefined for convenience:

`char-set:upper-case`	A–Z
`char-set:lower-case`	a–z
`char-set:numeric`	0–9
`char-set:whitespace`	space, newline, tab, linefeed, page, return
`char-set:not-whitespace`	Complement of `char-set:whitespace`
`char-set:alphabetic`	A–Z and a–z
`char-set:alphanumeric`	Alphabetic or numeric
`char-set:graphic`	Printing characters and space

(`char-upper-case?` *character*) ⟶ *boolean* procedure
(`char-lower-case?` *character*) ⟶ *boolean* procedure
(`char-numeric?` *character*) ⟶ *boolean* procedure
(`char-whitespace?` *character*) ⟶ *boolean* procedure
(`char-alphabetic?` *character*) ⟶ *boolean* procedure
(`char-alphanumeric?` *character*) ⟶ *boolean* procedure
(`char-graphic?` *character*) ⟶ *boolean* procedure

These predicates are defined in terms of the above character sets.

Chapter 6

Awk, record I/O, and field parsing

Unix programs frequently process streams of records, where each record is delimited by a newline, and records are broken into fields with other delimiters (for example, the colon character in `/etc/passwd`). Scsh has procedures that allow the programmer to easily do this kind of processing. Scsh's field parsers can also be used to parse other kinds of delimited strings, such as colon-separated `$PATH` lists. These routines can be used with scsh's `awk` loop construct to conveniently perform pattern-directed computation over streams of records.

6.1 Record I/O and field parsing

The procedures in this section are used to read records from I/O streams and parse them into fields. A record is defined as text terminated by some delimiter (usually a newline). A record can be split into fields by using regular expressions in one of several ways: to *match* fields, to *separate* fields, or to *terminate* fields. The field parsers can be applied to arbitrary strings (one common use is splitting environment variables such as `$PATH` at colons into its component elements).

6.1.1 Reading delimited strings

These procedures read in strings from ports delimited by characters belonging to a specific set. See section 5.3 for information on character set manipulation.

(`read-delimited` *char-set [port]*) $\longrightarrow$ *string or eof* procedure

Read until we encounter one of the chars in *char-set* or eof. The terminating character is not included in the string returned, nor is it removed from the input stream; the next input operation will encounter it. If we get a string back, then `(eof-object? (peek-char))` tells if the string was terminated by a delimiter or eof.

The *char-set* argument may be a charset, a string, a character, or a character predicate; it is coerced to a charset.

This operation is likely to be implemented very efficiently. In the Scheme 48 implementation, the Unix port case is implemented directly in C, and is much faster than the equivalent operation performed in Scheme with `peek-char` and `read-char`.

(`read-delimited!` *char-set buf [port start end]*) $\longrightarrow$ *nchars or eof or #f* procedure

A side-effecting variant of `read-delimited`.

The data is written into the string *buf* at the indices in the half-open interval $[start, end)$; the default interval is the whole string: $start = 0$ and $end =$ `(string-length` *buf*`)`. The values of *start* and *end* must specify a well-defined interval in *str*, *i.e.*, $0 \leq start \leq end \leq$ `(string-length` *buf*`)`.

It returns *nbytes*, the number of bytes read. If the buffer filled up without a delimiter character being found, `#f` is returned. If the port is at eof when the read starts, the eof object is returned.

If an integer is returned, then `(eof-object (peek-char port))` tells if the string was terminated by a delimiter or eof.

6.1.2 Reading records

(`record-reader` *[delims elide-delims? handle-delim]*) $\longrightarrow$ *procedure* procedure

Returns a procedure that reads records from a port. The procedure is invoked as follows:

(*reader [port]*) $\longrightarrow$ *string* `or eof`

A record is a sequence of characters terminated by one of the characters in *delims* or eof. If *elide-delims?* is true, then a contiguous sequence of delimiter chars are taken as a single record delimiter. If *elide-delims?* is false, then a delimiter char coming immediately after a delimiter char produces an empty string record. The reader consumes the delimiting char(s) before returning from a read.

The *delims* set defaults to the set $\{\text{newline}\}$. It may be a charset, string, character, or character predicate, and is coerced to a charset. The *elide-delims?* flag defaults to `#f`.

The *handle-delim* controls what is done with the record's terminating delimiter.

`'trim`	Delimiters are trimmed. (The default)
`'split`	Reader returns delimiter string as a second argument. If record is terminated by EOF, then the eof object is returned as this second argument.
`'concat`	The record and its delimiter are returned as a single string.

The reader procedure returned takes one optional argument, the port from which to read, which defaults to the current input port. It returns a string or eof.

`(read-paragraph` *[port delimiter?]*`)` $\longrightarrow$ *string or eof* procedure

This procedure skips blank lines, then reads text from a port until a blank line or eof is found. A "blank line" is a (possibly empty) line composed only of white space. If *delimiter?* is true, the terminating blank line is included in the return string; it defaults to `#f`. When the delimiter is included, `(match-string "\n[ \t]*\n$" paragraph)` can be used to determine if the paragraph was terminated by a blank line or by eof.

6.1.3 Parsing fields

`(field-splitter` *[regexp num-fields]*`)` $\longrightarrow$ *procedure* procedure
`(infix-splitter` *[delim num-fields handle-delim]*`)` $\longrightarrow$ *procedure* procedure
`(suffix-splitter` *[delim num-fields handle-delim]*`)` $\longrightarrow$ *procedure* procedure
`(sloppy-suffix-splitter` *[delim num-fields handle-delim]*`)` $\longrightarrow$ *procedure* procedure

These functions return a parser function that can be used as follows:

(*parser string [start]*) $\longrightarrow$ *string-list*

The returned parsers split strings into fields defined by regular expressions. You can parse by specifying a pattern that *separates* fields, a pattern that *terminates* fields, or a pattern that *matches* fields:

Procedure	Pattern
`field-splitter`	matches fields
`infix-splitter`	separates fields
`suffix-splitter`	terminates fields
`sloppy-suffix-splitter`	terminates fields

These parser generators are controlled by a range of options, so that you can precisely specify what kind of parsing you want. However, these options default to reasonable values for general use.

Defaults:

delim	=	`"[ \t\n]+|$"`	(suffix delimiter: white space or eos)
		`"[ \t\n]+"`	(infix delimiter: white space)
re	=	`"[^ \t\n]+"`	(non-white-space)
num-fields	=	`#f`	(as many fields as possible)
handle-delim	=	`'trim`	(discard delimiter chars)

. . . which means: break the string at white space, discarding the white space, and parse as many fields as possible.

The regular expression *delim* is used to match field delimiters. It can be either a string or a compiled regexp structure (see the `make-regexp` procedure). In the separator case, it defaults to a regular expression matching white space; in the terminator case, it defaults to white space or end-of-string.

The regular expression *re* is a regular expression used to match fields. It defaults to non-white-space.

The boolean *handle-delim* determines what to do with delimiters.

`'trim`	Delimiters are thrown away after parsing. (default)
`'concat`	Delimiters are appended to the field preceding them.
`'split`	Delimiters are returned as separate elements in the field vector.

The *num-fields* argument used to create the parser specifies how many fields to parse. If `#f` (the default), the procedure parses them all. If a positive integer n, exactly that many fields are parsed; it is an error if there are more or fewer than n fields in the record. If *num-fields* is a negative integer or zero, then $|n|$ fields are parsed, and the remainder of the string is returned in the last element of the field vector; it is an error if fewer than $|n|$ fields can be parsed.

The field parser produced is a procedure that can be employed as follows:

(*parse string [start]*) $\Longrightarrow$ *string-list*

The optional *start* argument (default 0) specifies where in the string to begin the parse. It is an error if *start* $>$ (`string-length` *string*).

The parsers returned by the four parser generators implement different kinds of field parsing:

`field-splitter` The regular expression specifies the actual field.

`suffix-splitter` Delimiters are interpreted as element *terminators*. If vertical-bar is the the delimiter, then the string `""` is the empty record `()`, `"foo|"` produces a one-field record `("foo")`, and `"foo"` is an error.

The syntax of suffix-delimited records is:

<record>	::=	`""`	(Empty record)
	|	*<element>* *<delim>* *<record>*	

It is an error if a non-empty record does not end with a delimiter. To make the last delimiter optional, make sure the delimiter regexp matches the end-of-string (`regexp "$"`).

`infix-splitter` Delimiters are interpreted as element *separators*. If comma is the delimiter, then the string `"foo,"` produces a two-field record `("foo" "")`.

The syntax of infix-delimited records is:

<record>	::=	`""`	(Forced to be empty record)
	|	*<real-infix-record>*	
<real-infix-record>	::=	*<element>* *<delim>* *<real-infix-record>*	
	|	*<element>*	

Note that separator semantics doesn't really allow for empty records – the straightforward grammar (*i.e.*, *<real-infix-record>*) parses an empty string as a singleton list whose one field is the empty string, `("")`, not as the empty record `()`. This is unfortunate, since it means that infix string parsing doesn't make `string-append` and `vector-append` isomorphic. For example,

```
((infix-splitter ":") (string-append x ":" y))
```

doesn't always equal

```
(vector-append ((infix-splitter ":") x)
               ((infix-splitter ":") y))
```

It fails when x or y are the empty string. Terminator semantics *does* preserve a similar isomorphism.

However, separator semantics is frequently what other Unix software uses, so to parse their strings, we need to use it. For example, Unix `$PATH` lists have separator semantics. The path list `"/bin:"` is broken up into `("/bin" "")`, not `("/bin")`. Comma-separated lists should also be parsed this way.

`sloppy-suffix` The same as the `suffix` case, except that the parser will skip an initial delimiter string if the string begins with one instead of parsing an initial empty field. This can be used, for example, to field-split a sequence of English text at white-space boundaries, where the string may begin or end with white space, by using regex `"[ \t]+|$"`. (But you would be better off using `field-splitter` in this case.)

Record	: suffix	: | $ suffix	: infix	non-: field
`""`	`()`	`()`	`()`	`()`
`":"`	`("")`	`("")`	`("" "")`	`()`
`"foo:"`	`("foo")`	`("foo")`	`("foo" "")`	`("foo")`
`":foo"`	*error*	`("" "foo")`	`("" "foo")`	`("foo")`
`"foo:bar"`	*error*	`("foo" "bar")`	`("foo" "bar")`	`("foo" "bar")`

Figure 6.1: Using different grammars to split records into fields.

Figure 6.1 shows how the different parser grammars split apart the same strings. Having to choose between the different grammars requires you to decide what you want, but at least you can be precise about what you are parsing. Take fifteen seconds and think it out. Say what you mean; mean what you say.

`(join-strings` *string-list [delimiter grammar]*`)` $\longrightarrow$ *string* procedure

This procedure is a simple unparser—it pastes strings together using the delimiter string.

The *grammar* argument is one of the symbols `infix` (the default) or `suffix`; it determines whether the delimiter string is used as a separator or as a terminator.

The delimiter is the string used to delimit elements; it defaults to a single space `" "`.

Example:

```
(join-strings '("foo" "bar" "baz") ":")
    ⟹  "foo:bar:baz"
```

6.1.4 Field readers

`(field-reader` *[field-parser rec-reader]*`)` $\longrightarrow$ *procedure* procedure

This utility returns a procedure that reads records with field structure from a port. The reader's interface is designed to make it useful in the `awk` loop macro (section 6.2). The reader is used as follows:

`(`*reader [port]*`)` $\Longrightarrow$ *[raw-record parsed-record]* `or` *[eof ()]*

When the reader is applied to an input port (default: the current input port), it reads a record using *rec-reader*. If this record isn't the eof object, it is parsed with *field-parser*. These two values—the record, and its parsed representation—are returned as multiple values from the reader.

When called at eof, the reader returns [eof-object `()`].

Although the record reader typically returns a string, and the field-parser typically takes a string argument, this is not required. The record reader can produce, and the field-parser consume, values of any type. However, the empty list returned as the parsed value on eof is hardwired into the field reader.

For example, if port `p` is open on `/etc/passwd`, then

```
((field-reader (infix-splitter ":" 7)) p)
```

returns two values:

```
"dalbertz:mx3Uaqq0:107:22:David Albertz:/users/dalbertz:/bin/csh"
("dalbertz" "mx3Uaqq0" "107" "22" "David Albertz" "/users/dalbertz"
          "/bin/csh")
```

The *field-parser* defaults to the value of `(field-splitter)`, a parser that picks out sequences of non-white-space strings.

The *rec-reader* defaults to `read-line`.

Figure 6.2 shows `field-reader` being used to read different kinds of Unix records.

6.1.5 Forward-progress guarantees and empty string matches

A loop that pulls text off a string by repeatedly matching a regexp against that string can conceivably get stuck in an infinite loop if the regexp matches the empty string. For example, the regexps `^`, `$`, `.*`, and `foo|[^f]*` can all match the empty string.

The routines in this package that iterate through strings with regular expressions are careful to handle this empty-string case. If a regexp matches the empty string, the next search starts, not from the end of the match (which in the empty string case is also the beginning—that's the problem), but from the next character over. This is the correct behaviour. Regexps match the longest possible string at a given location, so if the regexp matched the empty string at location i, then it is guaranteed it could not have matched a longer pattern starting with character i. So we can safely begin our search for the next match at char $i + 1$.

With this provision, every iteration through the loop makes some forward progress, and the loop is guaranteed to terminate.

This has the effect you want with field parsing. For example, if you split a string with the empty pattern, you will explode the string into its individual characters:

```
((suffix-splitter "") "foo") ⟹  ("" "f" "o" "o")
```

However, even though this boundary case is handled correctly, we don't recommend using it. Say what you mean—just use a field splitter:

```
;;; /etc/passwd reader
(field-reader (infix-splitter ":" 7))
    ; wandy:3xuncWdpKhR.:73:22:Wandy Saetan:/usr/wandy:/bin/csh

;;; Two ls -l output readers
(field-reader (infix-splitter "[ \t]+" 8))
(field-reader (infix-splitter "[ \t]+" -7))
    ; -rw-r--r--  1 shivers    22880 Sep 24 12:45 scsh.scm

;;; Internet hostname reader
(field-reader (field-splitter "[^.]+"))
    ; stat.sinica.edu.tw

;;; Internet IP address reader
(field-reader (field-splitter "[^.]+" 4))
    ; 18.24.0.241

;;; Line of integers
(let ((parser (field-splitter "[+-]?[0-9]+")))
  (field-reader (λ (s) (map string->number (parser s))))
    ; 18 24 0 241

;;; Same as above.
(let ((reader (field-reader (field-splitter "[+-]?[0-9]+"))))
  (λ maybe-port (map string->number (apply reader maybe-port))))
    ; Yale beat harvard 26 to 7.
```

Figure 6.2: Some examples of `field-reader`

```
((field-splitter ".") "foo") ⟹  ("f" "o" "o")
```

Or, more efficiently,

```
((λ (s) (map string (string->list s))) "foo")
```

6.1.6 Reader limitations

Since all of the readers in this package require the ability to peek ahead one char in the input stream, they cannot be applied to raw integer file descriptors, only Scheme input ports. This is because Unix doesn't support peeking ahead into input streams.

6.2 Awk

Scsh provides a loop macro and a set of field parsers that can be used to perform text processing very similar to the Awk programming language. These basic functionality of Awk is factored in scsh into its component parts. The control structure is provided by the `awk` loop macro; the text I/O and parsers are provided by the field-reader subroutine library (section 6.1). This factoring allows the programmer to compose the basic loop structure with any parser or input mechanism at all. If the parsers provided by the field-reader package are insufficient, the programmer can write a custom parser in Scheme and use it with equal ease in the awk framework.

Awk-in-scheme is given by a loop macro called `awk`. It looks like this:

```
(awk <next-record> <record&field-vars>
     [<counter>] <state-var-decls>
  <clause1> ...)
```

The body of the loop is a series of clauses, each one representing a kind of condition/action pair. The loop repeatedly reads a record, and then executes each clause whose condition is satisfied by the record.

Here's an example that reads lines from port `p` and prints the line number and line of every line containing the string "`Church-Rosser`":

```
(awk (read-line) (ln) lineno ()
  ("Church-Rosser" (format #t "~d: ~s~%" lineno ln)))
```

This example has just one clause in the loop body, the one that tests for matches against the regular expression "`Church-Rosser`".

The *<next-record>* form is an expression that is evaluated each time through the loop to produce a record to process. This expression can return multiple values; these values are bound to the variables given in the <*record&field-vars*> list of variables. The first value returned is assumed to be the record; when it is the end-of-file object, the loop terminates.

For example, let's suppose we want to read items from *etc.*/etc/password, and we use the `field-reader` procedure to define a record parser for `/etc/passwd` entries:

```
(define read-passwd (field-reader (infix-splitter ":" 7)))
```

binds `read-passwd` to a procedure that reads in a line of text when it is called, and splits the text at colons. It returns two values: the entire line read, and a seven-element list of the split-out fields. (See section 6.1 for more on `field-reader` and `infix-splitter`.)

So if the <*next-record*> form in an `awk` expression is `(read-passwd)`, then <*record&field-vars*> must be a list of two variables, *e.g.*,

```
(record field-vec)
```

since `read-passwd` returns two values.

Note that `awk` allows us to use *any* record reader we want in the loop, returning whatever number of values we like. These values don't have to be strings or string lists. The only requirement is that the record reader return the eof object as its first value when the loop should terminate.

The `awk` loop allows the programmer to have loop variables. These are declared and initialised by the <*state-var-decls*> form, a

`((`*var init-exp*`) (`*var init-exp*`) ...)`

list rather like the `let` form. Whenever a clause in the loop body executes, it evaluates to as many values as there are state variables, updating them.

The optional <*counter*> variable is an iteration counter. It is bound to 0 when the loop starts. The counter is incremented each time a non-eof record is read.

There are several kinds of loop clause. When evaluating the body of the loop, `awk` evaluates *all* the clauses sequentially. Unlike `cond`, it does not stop after the first clause is satisfied; it checks them all.

- `(`*test* $body_1$ $body_2$ `...)`
  If *test* is true, execute the body forms. The last body form is the value of the clause. The test and body forms are evaluated in the scope of the record and state variables.

  The *test* form can be one of:

integer:	The test is true for that iteration of the loop. The first iteration is #1.
string:	The string is a regular expression. The test is true if the regexp matches the record.
expression	If not an integer or a string, the test form is a Scheme expression that is evaluated.

- `(range` *start-test stop-test* $body_1$ `...)`
  `(:range` *start-test stop-test* $body_1$ `...)`
  `(range:` *start-test stop-test* $body_1$ `...)`
  `(:range:` *start-test stop-test* $body_1$ `...)`
  These clauses become activated when *start-test* is true; they stay active on all further iterations until *stop-test* is true.

  So, to print out the first ten lines of a file, we use the clause:

  ```
  (:range: 1 10 (display record))
  ```

  The colons control whether or not the start and stop lines are processed by the clause. For example:

`(range 1 5 ...)`	Lines 2 3 4
`(:range 1 5 ...)`	Lines 1 2 3 4
`(range: 1 5 ...)`	Lines 2 3 4 5
`(:range: 1 5 ...)`	Lines 1 2 3 4 5

  A line can trigger both tests, either simultaneously starting and stopping an active region, or simultaneously stopping one and starting a new one, so ranges can abut seamlessly.

- `(else` $body_1$ $body_2$ `...)`
  If no other clause has executed since the top of the loop, or since the last `else` clause, this clause executes.

- `(`*test* `=>` *exp*`)`
  If evaluating `test` produces a true value, apply *exp* to that value. If *test* is a regular-expression string, then *exp* is applied to the match data structure returned by the regexp match routine.

- `(after` $body_1$ `...)`
  This clause executes when the loop encounters EOF. The body forms execute in the scope of the state vars and the record-count var, if there are any. The value of the last body form is the value of the entire awk form.

  If there is no `after` clause, `awk` returns the loop's state variables as multiple values.

6.2.1 Examples

Here are some examples of awk being used to process various types of input stream.

```
(define $ vector-ref)        ; Saves typing.

;;; Print out the name and home-directory of everyone in /etc/passwd:
(let ((read-passwd (field-reader (infix-splitter ":" 7))))
  (call-with-input-file "/etc/passwd"
    (lambda (port)
      (awk (read-passwd port) (record fields) ()
        (#t (format #t "~a's home directory is ~a~%"
                    ($ fields 0)
                    ($ fields 5)))))))


;;; Print out the user-name and home-directory of everyone whose
;;; name begins with "S"
(let ((read-passwd (field-reader (infix-splitter ":" 7))))
  (call-with-input-file "/etc/passwd"
    (lambda (port)
      (awk (read-passwd port) (record fields) ()
        ("^S" (format #t "~a's home directory is ~a~%"
                      ($ fields 0)
                      ($ fields 5)))))))


;;; Read a series of integers from stdin. This expression evaluates
;;; to the number of positive numbers were read. Note our "record-reader"
;;; is the standard Scheme READ procedure.
(awk (read) (i)   ((npos 0))
  ((> i 0) (+ npos 1)))


;;; Filter -- pass only lines containing my name.
(awk (read-line) (line) ()
  ("Olin" (display line) (newline)))


;;; Count the number of non-comment lines of code in my Scheme source.
(awk (read-line) (line) ((nlines 0))
  ("^[ \t]*;" nlines)              ; A comment line.
  (else       (+ nlines 1)))       ; Not a comment line.
```

```
;;; Read numbers, counting the evens and odds.
(awk (read) (val) ((evens 0) (odds 0))
  ((> val 0) (display "pos ")  (values evens odds)) ; Tell me about
  ((< val 0) (display "neg ")  (values evens odds)) ; sign, too.
  (else      (display "zero ") (values evens odds))

  ((even? val) (values (+ evens 1) odds))
  (else        (values evens       (+ odds 1))))


;;; Determine the max length of all the lines in the file.
(awk (read-line) (line) ((max-len 0))
  (#t (max max-len (string-length line))))


;;; (This could also be done with REDUCE-PORT:)
(reduce-port (current-input-port) read-line
             (lambda (line maxlen) (max (string-length line) maxlen))
             0)


;;; Print every line longer than 80 chars.
;;; Prefix each line with its line #.
(awk (read-line) (line) lineno ()
  ((> (string-length line) 80)
   (format #t "~d: ~s~%" lineno line)))


;;; Strip blank lines from input.
(awk (read-line) (line) ()
  ("." (display line) (newline)))


;;; Sort the entries in /etc/passwd by login name.
(for-each (lambda (entry) (display (cdr entry)) (newline))          ; Out
          (sort (lambda (x y) (string<? (car x) (car y)))           ; Sort
                (let ((read (field-reader (infix-splitter ":" 7)))) ; In
                  (awk (read) (line fields) ((ans '()))
                    (#t (cons (cons ($ fields 0) line) ans))))))


;;; Prefix line numbers to the input stream.
(awk (read-line) (line) lineno ()
  (#t (format #t "~d:\t~a~%" lineno line)))
```

Chapter 7

Miscellaneous routines

7.1 Integer bitwise ops

(`arithmetic-shift` *i j*) $\longrightarrow$ *integer* procedure
(`bitwise-and` *i j*) $\longrightarrow$ *integer* procedure
(`bitwise-ior` *i j*) $\longrightarrow$ *integer* procedure
(`bitwise-not` *i*) $\longrightarrow$ *integer* procedure
(`bitwise-xor` *i j*) $\longrightarrow$ *integer* procedure

These operations operate on integers representing semi-infinite bit strings, using a 2's-complement encoding.

`arithmetic-shift` shifts *i* by *j* bits. A left shift is $j > 0$; a right shift is $j < 0$.

7.2 List procedures

(`nth` *list i*) $\longrightarrow$ *object* procedure

Returns the i^{th} element of *list*. The first element (the car) is `(nth` *list* `0)`, the second element is `(nth` *list* `1)`, and so on.

This procedure is provided as it is useful for accessing elements from the lists returned by the field-readers (chapter 6).

7.3 Top level

`(repl)` $\longrightarrow$ *undefined* procedure

This runs a Scheme 48 read-eval-print loop, reading forms from the current input port, and writing their values to the current output port.

If you wish to try something dangerous, and want to be able to recover your shell state, you can fork off a subshell with the following form:

```
(run (begin (repl)))
```

Chapter 8

Running scsh

Scsh is currently implemented on top of Scheme 48, a freely-available Scheme implementation written by Jonathan Rees and Richard Kelsey. Scheme 48 uses a byte-code interpreter for good code density, portability and medium efficiency. It is R4RS. The version on top of which scsh is currently built (0.36) lacks floating point. It also has a module system designed by Jonathan Rees.

Scsh's design is not Scheme 48 specific, although the current implementation is necessarily so. Scsh is intended to be implementable in other Scheme implementations—although such a port may require some work. The Scheme 48 vm that scsh uses is a specially modified version; standard Scheme 48 virtual machines cannot be used with the scsh heap image.

To run the Scheme 48 implementation of scsh, you run a specially modified copy of the Scheme 48 virtual machine with a scsh heap image. This command starts the vm up with a 1Mword heap (split into two semispaces):

```
scshvm -o scshvm -h 1000000 -i scsh.image arg1 arg2 ...
```

The vm peels off initial vm arguments up to the `-i` heap image argument, which terminates vm argument parsing. The rest of the arguments are passed off to the scsh top-level. Scsh's top-level removes scsh arguments; the rest show up as the value of `command-line-arguments`.

Alternatively, you can run the scsh top-level binary. This is nothing but a small cover program that invokes the scsh vm on the scsh heap image for you. This allows you to simply start up an interactive scsh from a command line, as well as write shell scripts that begin with the simple trigger

```
#!/usr/local/bin/scsh -s
```

8.1 VM arguments

Scsh uses a special version of the Scheme 48 virtual machine. It takes arguments in the following form:

`scshvm` [*meta-arg*] [*vm-options*$^+$] [*end-option scheme-args*]

where

meta-arg:	`\` *script*
vm-option:	`-h` *heap-size-in-words* `-s` *stack-size-in-words* `-o` *object-file-name*
end-option:	`-i` *image-file-name* `--`

8.1.1 The meta argument

The Scheme 48 vm takes a special command-line switch, a single backslash called the "meta-switch," which is useful for shell scripts. While parsing the command-line arguments, if the vm sees a "\" argument, followed by a file-name argument *fname*, it will open the file *fname*, and read more arguments from the second line of this file. This list of arguments will then replace the "\" argument—*i.e.*, the new arguments are inserted in front of *fname*, and the argument parser resumes argument scanning. This is used to overcome a limitation of the `#!` feature: the `#!` line can only specify a single argument after the interpreter. For example, we might hope the following scsh script, `ekko`, would implement a simple-minded version of `echo(1)`:

```
#!/bin/scshvm -o /bin/scshvm -i /lib/scsh.image -s
!#
(map (λ (arg) (display arg) (display " "))
     command-line-arguments)
(newline)
```

The idea would be that the command

```
ekko Hi there.
```

would by expanded by `exec(2)` into

```
/bin/scshvm -o /bin/scshvm -i /lib/scsh.image -s ekko Hi there.
```

In theory, this would cause scsh to start up, set `command-line-arguments` to `("Hi" "there.")`, load the source file `ekko`, and exit.

However, the Unix `exec(2)` call will not handle multiple arguments on the `#!` line, so this script won't work. We must instead invoke the Scheme 48 vm with the single \ argument, and put the rest of the arguments on line two of the script. Here's the correct script:[1]

```
#!/bin/scshvm \
-o /bin/scshvm -i /lib/scsh.image -s
!#
(map (λ (arg) (display arg) (display " "))
     command-line-arguments)
(newline)
```

Now, the invocation starts as

```
ekko Hi there.
```

and is expanded by exec(2) into

```
/bin/scshvm \ ekko Hi there.
```

When scshvm starts up, it expands the "\" argument into the arguments read from line two of `ekko`, producing this argument list:

```
-o /bin/scshvm -i /lib/scsh.image -s ekko Hi there.
                     ↑
```

Expanded from `\ ekko`

With this argument list, processing proceeds as we intended.

8.1.2 VM options

The `-o` *object-file-name* switch tells the vm where to find relocation information for its foreign-function calls. Scsh will use a pre-compiled default if it is not specified. Scsh must have this information to run, since scsh's syscall interfaces are done with foreign-function calls.

The `-h` and `-s` options tell the vm how much space to allocate for the heap and stack.

[1] In fact, I'm playing fast and loose with the actual pathnames used in this example: `scshvm` is probably not going to be found in `/bin`. I've abbreviated things so the long argument lists will fit into one line of text. See the following sections for the full details.

8.1.3 End options

End options terminate argument parsing. The `-i` switch is followed by the name of a heap image for the vm to execute, and terminates vm argument parsing; following arguments are passed off to the heap image's top-level program. The `--` switch terminates argument parsing without giving a specific heap image; the vm will start up with using a default heap (whose location is compiled into the vm).

Notice that you are not allowed to pass arguments to the heap image's top-level program (*e.g.*, scsh) without delimiting them with `-i` or `--` flags.

8.2 Scsh arguments

Scsh's top-level argument parser takes arguments in a simple format:

`scsh [`*end-option* arg_1 ... arg_n`]`

where

end-option: `-s` *script*
`--`

The `-s` argument causes scsh to load a script file and exit. It also terminates argument parsing; following arguments are passed to the scsh program as the value of `command-line-arguments`.

If the `-s` argument is not given, scsh runs in interactive mode, with a standard Scheme 48 prompt-read-eval-print loop.

The `--` switch terminates argument parsing without specifying a script to load; it allows the user to pass arguments to an interactive scsh.

Shell scripts can be written and invoked with a `#!` initial line. Scsh defines the sequence `#!` to be a read-macro similar to the comment character `;`. The read-macro causes scsh to skip characters until it reads a newline, `!`, `#`, newline sequence. So an initial `#!` line is ignored by scsh.

8.3 Compiling shell scripts

The Scheme implementation of scsh allows you to create a heap image with your own top-level procedure. Adding the pair of lines

```
#!/usr/local/bin/scshvm \\
-o /usr/local/bin/scshvm -i
```

to the top of the heap image will turn it into an executable Unix file.

(`dump-scsh-program` *main fname*) $\longrightarrow$ *undefined* procedure

This procedure writes out a scsh heap image. When the heap image is executed by the Scheme vm, it will call the *main* procedure on no arguments and then exit. The Scheme vm will parse command-line arguments as described in section 8.1, and bind remaining arguments to the `command-line-arguments` variable before calling `main`. Further argument parsing (as described for scsh in section 8.2 is not performed.

The heap image created by `dump-scsh-program` has unused code and data pruned out, so small programs compile to much smaller heap images.

8.4 Standard file locations

Because the scshvm binary is intended to be used for writing shell scripts, it is important that the binary be installed in a standard place, so that shell scripts can dependably refer to it. The standard directory for the scsh tree should be `/usr/local/lib/scsh/`. Whenever possible, the vm should be located in

```
/usr/local/lib/scsh/scshvm
```

and a scsh heap image should be located in

```
/usr/local/lib/scsh/scsh.image
```

The top-level scsh program should be located in

```
/usr/local/lib/scsh/scsh
```

with a symbolic link to it from

```
/usr/local/bin/scsh
```

The Scheme 48 image format allows heap images to have `#!` triggers, so `scsh.image` should have a `#!` trigger of the following form:

```
#!/usr/local/bin/scshvm \
-o /usr/local/bin/scshvm -i
... heap image goes here ...
```

Chapter 9

Todo

The LaTeX hackery needs yet another serious pass. Most importantly, long procedure "declarations" need to be broken across two lines.

Fix up 0-or-more and 1-or-more parameter typesetting, with subscripts.

Parameter subscripts need to be made real subscripts.

Job control, after `jcontrol.scm`

Static heaps; fast startup.

Gnu readline lib.

Interrupt system.

Make it all coexist with S48 threads as well as can be done for Unix. The DEC SRC tech report gives a good discussion of the issues.

Support for file locking: `(lock-file fd op)`, `with-file-locked`, ...

Testing broken symlinks – new value for *chase?* flag?

Interactive flag machinery

Rename and release `ensure-file-name-is-{non,}directory`.

More informative errno exception packets & documentation for them.

Other things should be available: hash tables, sort, list utils, pattern matchers. But things start to overload. The module system is the appropriate way to use these.

Support for writing scripts that use the module language.

Need calls to control port `i/o` buffering.

Need to do file-control (*i.e.*, `fcntl()`). `fcntl` is ugly. Better to have a procedure for each different operation.

Tty stuff and control tty.

More documentation for the `wait()` machinery.

We need a general time/date parser, that can convert strings like "Thursday after Christmas" into date records.

Index