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* and *redirections*.

2.1 Process forms and I/O redirections

An *process form* is a specification of a Unix process to run, possibly in a particular I/O environment. 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
(prog arg1 ... argn)             ; Exec the program.
(pf redir1 ... redirn)           ; I/O redirections
```

The (*prog* arg_1 ... arg_n) case 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.

The `begin` process form does a `stdio->stdports` synchronisation in the child process before executing the body of the form. This guarantees that the `begin` form, like all other process forms, "sees" the effects of any associated I/O redirections.

Note that R5RS does not specify whether or not `|` and `|+` are readable symbols. Scsh does.

For the final case, 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 one of the 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 Using process forms in Scheme

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

`(exec-pf` . *pf*`)` ⟶ *no return value* — syntax
`(&` . *pf*`)` ⟶ *proc* — syntax
`(run` . *pf*`)` ⟶ *status* — syntax

The `(exec-pf . ` *pf*`)` form nukes the current process: it establishes the I/O redirections and then overlays the current process with the requested computation.

The `(& . ` *pf*`)` form is similar, except that the process is forked off in background. The form returns the subprocess' process object.

The `(run . ` *pf*`)` 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-pf` 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:

```
(& . pf)    ≡  (fork (λ () (exec-pf . pf)))
(run . pf)  ≡  (wait (& . pf))
```

2.3 Process descriptors

Process form are syntactic entities. A program can convert a process form to a value, a so-called *process descriptor*. All syntactic operations for composing process forms have direct procedural counterparts for composing process descriptors at run time.

`(process-form . `*pf*`)` $\longrightarrow$ *process descriptor* syntax

> The `process-form` macro converts a process form into a process descriptor.

Similarly, redirection specs also have value counterparts, so-called *redirection descriptors*:

`(redirection `*redir*`)` $\longrightarrow$ *redirection descriptor* syntax

> The `redirection` macro converts a redirection spec into its corresponding redirection descriptor.

`(exec-process-desc `*pd*`)` $\longrightarrow$ *no return value* procedure
`(&* `$pd_1 \ldots pd_n$`)` $\longrightarrow$ *proc* procedure
`(run* `*pd*`)` $\longrightarrow$ *proc* procedure

> These procedures are analogous to the `exec-pf`, `&`, and `run` macros, respectively. The only differences are that they are procedures and that they accept process descriptors instead of process forms as arguments.

`(|* `$pd_1 \ldots pd_n$`)` $\longrightarrow$ *process descriptor* procedure
`(pipe* `$pd_1 \ldots pd_n$`)` $\longrightarrow$ *process descriptor* procedure
`(|+* `*connect-list* $pd_1 \ldots pd_n$`)` $\longrightarrow$ *process descriptor* procedure
`(pipe+* `$pd_1 \ldots pd_n$`)` $\longrightarrow$ *process descriptor* procedure
`(with-redirection `*rd pd*`)` $\longrightarrow$ *process descriptor* procedure

> The `|*` and —+* procedures correspond to the `|` and `|+` macros. The only difference is that they accept process descriptors as arguments instead of process forms. The `pipe*` and `pipe+*` procedures are synonyms for `|*` and `|+*`. The `with-redirection` macro corresponds to process forms that specify I/O redirections; it accepts a redirection descriptor returned by the `redirection` macro and a process descriptor as arguments.

2.3.1 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 . `*pf*`)` $\longrightarrow$ *port* syntax

(run/file . *pf*) ⟶ *string* syntax
(run/string . *pf*) ⟶ *string* syntax
(run/strings . *pf*) ⟶ *string list* syntax
(run/sexp . *pf*) ⟶ *object* syntax
(run/sexps . *pf*) ⟶ *list* syntax

These forms all fork off subprocesses, collecting the process' output to stdout in some form or another. The subprocess runs with file descriptor 1 and the current output port bound to a pipe.

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 . pf)` expands into

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

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

(port->string *port*) $\longrightarrow$ *string* procedure
(port->sexp-list *port*) $\longrightarrow$ *list* procedure
(port->string-list *port*) $\longrightarrow$ *string list* procedure
(port->list *reader port*) $\longrightarrow$ *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*) $\equiv$ (port->list read-line *port*)
(port->sexp-list *port*) $\equiv$ (port->list read *port*)

The following compositions also hold:

run/string*	$\equiv$	port->string	$\circ$	run/port*
run/strings*	$\equiv$	port->string-list	$\circ$	run/port*
run/sexp*	$\equiv$	read	$\circ$	run/port*
run/sexps*	$\equiv$	port->sexp-list	$\circ$	run/port*

(port-fold *port reader op . seeds*) $\longrightarrow$ *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 `port-fold` 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 (port-fold *port reader* cons '()))

An imperative way to look at `port-fold` 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.

Remark: This procedure was formerly named `reduce-port` . The old binding is still provided, but is deprecated and will probably vanish in a future release.

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+proc . `*pf*`)` $\longrightarrow$ *[port proc]* — syntax
`(run/port+proc* `*thunk*`)` $\longrightarrow$ *[port proc]* — 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 current output port), and the subprocess's process object. A process object encapsulates the subprocess' process id and exit code; it is the value passed to the `wait` system call.

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 child) (run/port+proc (zcat tr91-145.tex.Z))
  (let* ((paper (port->string port))
         (status (wait child)))
    ...use paper, status, and child 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 . pf*`)` $\longrightarrow$ *[status port...]* — syntax
`(run/collecting* `*fds thunk*`)` $\longrightarrow$ *[status 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.

`(||` $pf_1 \ldots pf_n$`)` $\longrightarrow$ *boolean* syntax

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

(&& $pf_1 \ldots pf_n$) $\longrightarrow$ *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.