MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Laboratory for Computer Science

Personal Information Architecture Note 3 4/94

A Scheme Shell

Olin Shivers
shivers@lcs.mit.edu

> *Although robust enough for general use, adventures into the esoteric periphery of the C shell may reveal unexpected quirks.*
>
> — SunOS 4.1 csh(1) man page, 10/2/89

Prologue

Shell programming terrifies me. There is something about writing a simple shell script that is just much, much more unpleasant than writing a simple C program, or a simple Common Lisp program, or a simple Mips assembler program. Is it trying to remember what the rules are for all the different quotes? Is it having to look up the multi-phased interaction between filename expansion, shell variables, quotation, backslashes and alias expansion? Maybe it's having to subsequently look up which of the twenty or thirty flags I need for my grep, sed, and awk invocations. Maybe it just gets on my nerves that I have to run two complete programs simply to count the number of files in a directory (`ls | wc -l`), which seems like several orders of magnitude more cycles than was really needed.

Whatever it is, it's an object lesson in angst. Furthermore, during late-night conversations with office mates and graduate students, I have formed the impression that I am not alone. In late February[1], I got embroiled in a multi-way email flamefest about just exactly what it was about Unix that drove me nuts. In the midst of the debate, I did a rash thing. I claimed that it would be easy and so much nicer to do shell programming from Scheme. Some functions to interface to the OS and a few straightforward macros would suffice to remove the spectre of `#!/bin/csh` from my life forever. The serious Unix-philes in the debate expressed their doubts. So I decided to go do it.

Probably only take a week or two.

---

[1] February 1992, that is.

Contents

1 Introduction **1**

2 Unix shells **2**

3 Process notation **6**
- 3.1 Extended process forms and i/o redirections 6
- 3.2 Process forms 8
- 3.3 Using extended process forms in Scheme 8
- 3.4 Procedures and special forms 11
- 3.5 Interfacing process output to Scheme 12

4 System calls **14**

5 The Tao of Scheme and Unix **14**

6 I/O **16**

7 Lexical issues **18**

8 Implementation **19**

9 Size **20**

10 Systems programming in Scheme **21**
- 10.1 Exceptions and robust error handling 22
- 10.2 Automatic storage management 22
- 10.3 Return values and procedural composition 24
- 10.4 Strings 25
- 10.5 Higher-order procedures 25
- 10.6 S-expression syntax and backquote 26

11 Other programming languages **27**
- 11.1 Functional languages 27
- 11.2 Shells 28
- 11.3 New-generation scripting languages 29

12 Future work **29**
- 12.1 Command language features 30
- 12.2 Little languages 30

13 Conclusion 31

14 Acknowledgements 31

References 33

Notes 35

1 Introduction

The central artifact of this paper is a new Unix shell called scsh. However, I have a larger purpose beyond simply giving a description of the new system. It has become fashionable recently to claim that "language doesn't matter." After twenty years of research, operating systems and systems applications are still mainly written in C and its complex successor, C++. Perhaps advanced programming languages offer too little for the price they demand in efficiency and formal rigor.

I disagree strongly with this position, and I would like to use scsh, in comparison to other Unix systems programming languages, to make the point that language *does* matter. After presenting scsh in the initial sections of the paper, I will describe its design principles, and make a series of points concerning the effect language design has upon systems programming. I will use scsh, C, and the traditional shells as linguistic exemplars, and show how their various notational and semantic tradeoffs affect the programmer's task. In particular, I wish to show that a functional language such as Scheme is an excellent tool for systems programming. Many of the linguistic points I will make are well-known to the members of the systems programming community that employ modern programming languages, such as DEC SRC's Modula-3 [Nelson]. In this respect, I will merely be serving to recast these ideas in a different perspective, and perhaps diffuse them more widely.

The rest of this paper is divided into four parts:

- In part one, I will motivate the design of scsh (section 2), and then give a brief tutorial on the system (3, 4).
- In part two, I discuss the design issues behind scsh, and cover some of the relevant implementation details (5–9).
- Part three concerns systems programming with advanced languages. I will illustrate my points by comparing scsh to other Unix programming systems (10, 11).
- Finally, we conclude, with some indication of future directions and a few final thoughts.

2 Unix shells

Unix shells, such as sh or csh, provide two things at once: an interactive command language and a programming language. Let us focus on the latter function: the writing of "shell scripts"—interpreted programs that perform small tasks or assemble a collection of Unix tools into a single application.

Unix shells are real programming languages. They have variables, if/then conditionals, and loops. But they are terrible programming languages. The data structures typically consist only of integers and vectors of strings. The facilities for procedural abstraction are non-existent to minimal. The lexical and syntactic structures are multi-phased, unprincipled, and baroque.

If most shell languages are so awful, why does anyone use them? There are a few important reasons.

- A programming language is a notation for expressing computation. Shells have a notation that is specifically tuned for running Unix programs and hooking them together. For example, suppose you want to run programs `foo` and `bar` with `foo` feeding output into `bar`. If you do this in C, you must write: two calls to `fork()`, two calls to `exec()`, one call to `pipe()`, several calls to `close()`, two calls to `dup()`, and a lot of error checks (fig. 1). This is a lot of picky bookkeeping: tedious to write, tedious to read, and easy to get wrong on the first try. In sh, on the other hand, you simply write "`foo | bar`" which is much easier to write and much clearer to read. One can look at this expression and instantly understand it; one can write it and instantly be sure that it is correct.

- They are interpreted. Debugging is easy and interactive; programs are small. On my workstation, the "hello, world" program is 16kb as a compiled C program, and 29 bytes as an interpreted sh script.

  In fact, `/bin/sh` is just about the only language interpreter that a programmer can absolutely rely upon having available on the system, so this is just about the only reliable way to get interpreted-code density and know that one's program will run on any Unix system.

- Because the shell is the programmer's command language, the programmer is usually very familiar with its commonly-used command-language subset (this familiarity tails off rapidly, however, as the

```
int fork_foobar(void)       /* foo | bar in C */
{
    int pid1 = fork();
    int pid2, fds[2];

    if( pid1 == -1 ) {
        perror("foo|bar");
        return -1;
        }

    if( !pid1 ) {
        int status;
        if( -1 == waitpid(pid1, &status, 0) ) {
            perror("foo|bar");
            return -1;
            }
        return status;
        }

    if( -1 == pipe(fds) ) {
        perror("foo|bar");
        exit(-1);
        }

    pid2 = fork();
    if( pid2 == -1 ) {
        perror("foo|bar");
        exit(-1);
        }

    if( !pid2 ) {
        close(fds[1]);
        dup2(fds[0], 1);
        execlp("foo", "foo", NULL);
        perror("foo|bar");
        exit(-1);
        }

    close(fds[0]);
    dup2(fds[1], 0);
    execlp("bar", "bar", NULL);
    perror("foo|bar");
    exit(-1);
    }
```

Figure 1: Why we program with shells.

> demands of shell programming move the programmer out into the dustier recesses of the language's definition.)

There is a tension between the shell's dual role as interactive command language and shell-script programming language. A command language should be terse and convenient to type. It doesn't have to be comprehensible. Users don't have to maintain or understand a command they typed into a shell a month ago. A command language can be "write-only," because commands are thrown away after they are used. However, it is important that most commands fit on one line, because most interaction is through tty drivers that don't let the user back up and edit a line after its terminating newline has been entered. This seems like a trivial point, but imagine how irritating it would be if typical shell commands required several lines of input. Terse notation is important for interactive tasks.

Shell syntax is also carefully designed to allow it to be parsed on-line—that is, to allow parsing and interpretation to be interleaved. This usually penalizes the syntax in other ways (for example, consider rc's clumsy if / then / else syntax [rc]).

Programming languages, on the other hand, can be a little more verbose, in return for generality and readability. The programmer enters programs into a text editor, so the language can spread out a little more.

The constraints of the shell's role as command language are one of the things that make it unpleasant as a programming language.

The really compelling advantage of shell languages over other programming languages is the first one mentioned above. Shells provide a powerful notation for connecting processes and files together. In this respect, shell languages are extremely well-adapted to the general paradigm of the Unix operating system. In Unix, the fundamental computational agents are programs, running as processes in individual address spaces. These agents cooperate and communicate among themselves to solve a problem by communicating over directed byte streams called pipes. Viewed at this level, Unix is a data-flow architecture. From this perspective, the shell serves a critical role as the language designed to assemble the individual computational agents to solve a particular task.

As a programming language, this interprocess "glue" aspect of the shell is its key desireable feature. This leads us to a fairly obvious idea: instead of adding weak programming features to a Unix process-control language, why not add process invocation features to a strong programming language?

What programming language would make a good base? We would want a language that was powerful and high-level. It should allow for implementations based on interactive interpreters, for ease of debugging and to keep programs small. Since we want to add new notation to the language, it would help if the language was syntactically extensible. High-level features such as automatic storage allocation would help keep programs small and simple. Scheme is an obvious choice. It has all of the desired features, and its weak points, such as it lack of a module system or its poor performance relative to compiled C on certain classes of program, do not apply to the writing of shell scripts.

I have designed and implemented a Unix shell called scsh that is embedded inside Scheme. I had the following design goals and non-goals:

- The general systems architecture of Unix is cooperating computational agents that are realised as processes running in separate, protected address spaces, communicating via byte streams. The point of a shell language is to act as the glue to connect up these computational agents. That is the goal of scsh. I resisted the temptation to delve into other programming models. Perhaps cooperating lightweight threads communicating through shared memory is a better way to live, but it is not Unix. The goal here was not to come up with a better systems architecture, but simply to provide a better way to drive Unix. {Note Agenda}

- I wanted a programming language, not a command language, and I was unwilling to compromise the quality of the programming language to make it a better command language. I was not trying to replace use of the shell as an interactive command language. I was trying to provide a better alternative for writing shell scripts. So I did not focus on issues that might be important for a command language, such as job control, command history, or command-line editing. There are no write-only notational conveniences. I made no effort to hide the base Scheme syntax, even though an interactive user might find all the necessary parentheses irritating. (However, see section 12.)

- I wanted the result to fit naturally within Scheme. For example, this ruled out complex non-standard control-flow paradigms, such as awk's or sed's.

The result design, scsh, has two dependent components, embedded within a very portable Scheme system:

- A high-level process-control notation.
- A complete library of Unix system calls.

The process-control notation allows the user to control Unix programs with a compact notation. The syscall library gives the programmer full low-level access to the kernel for tasks that cannot be handled by the high-level notation. In this way, scsh's functionality spans a spectrum of detail that is not available to either C or sh.

3 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*.

3.1 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* $redir_1 \; \ldots \; redir_n$)

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 *fdes* file descriptors have these defaults:

<	<<	>	>>
0	0	1	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`.” This implicit backquoting is an important feature of the process notation, as we’ll see later (sections 5 and 10.6).

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). {Note No port sync}

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.

3.2 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. (See section 7 for further details on lexical issues.)

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 3)) 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 3)` causes pf_1's file descriptor 3 to be connected to pf_2's file descriptor 3.

3.3 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)   ; Nuke the current process.
(& . epf)          ; Run epf in background; return pid.
(run . epf)        ; Run epf; wait for termination.
                   ;     Returns 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))
```

Figures 2 and 3 show a series of examples employing a mix of the process notation and the syscall library. Note that regular Scheme is used to provide the control structure, variables, and other linguistic machinery needed by the script fragments.

```
;; If the resource file exists, load it into X.
(if (file-exists? f))
    (run (xrdb -merge ,f)))

;; Decrypt my mailbox; key is "xyzzy".
(run (crypt xyzzy) (< mbox.crypt) (> mbox))

;; Dump the output from ls, fortune, and from into log.txt.
(run (begin (run (ls))
            (run (fortune))
            (run (from)))
     (> log.txt))

;; Compile FILE with FLAGS.
(run (cc ,file ,@flags))

;; Delete every file in DIR containing the string "/bin/perl":
(with-cwd dir
  (for-each (λ (file)
              (if (zero? (run (grep -s /bin/perl ,file)))
                  (delete-file file)))
            (directory-files)))
```

Figure 2: Example shell script fragments (a)

```
;; M4 preprocess each file in the current directory, then pipe
;; the input into cc. Errlog is foo.err, binary is foo.exe.
;; Run compiles in parallel.
(for-each (λ (file)
            (let ((outfile (replace-extension file ".exe"))
                  (errfile (replace-extension file ".err")))
              (& (| (m4) (cc -o ,outfile))
                 (< ,file)
                 (> 2 ,errfile))))
          (directory-files))

;; Same as above, but parallelise even the computation
;; of the filenames.
(for-each (λ (file)
            (& (begin (let ((outfile (replace-extension file ".exe"))
                            (errfile (replace-extension file ".err")))
                        (exec-epf (| (m4) (cc -o ,outfile))
                                  (< ,file)
                                  (> 2 ,errfile))))))
          (directory-files))

;; DES encrypt string PLAINTEXT with password KEY. My DES program
;; reads the input from fdes 0, and the key from fdes 3. We want to
;; collect the ciphertext into a string and return that, with error
;; messages going to our stderr. Notice we are redirecting Scheme data
;; structures (the strings PLAINTEXT and KEY) from our program into
;; the DES process, instead of redirecting from files. RUN/STRING is
;; like the RUN form, but it collects the output into a string and
;; returns it (see following section).

(run/string (/usr/shivers/bin/des -e -3)
            (<< ,plaintext) (<< 3 ,key))

;; Delete the files matching regular expression PAT.
;; Note we aren't actually using any of the process machinery here --
;; just pure Scheme.
(define (dsw pat)
  (for-each (λ (file)
              (if (y-or-n? (string-append "Delete " file))
                  (delete-file file)))
            (file-match #f pat)))
```

Figure 3: Example shell script fragments (b)

3.4 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.

`(fork` *[thunk]*`)` *procedure*

`Fork` spawns a Unix subprocess. Its exact behavior depends on whether it is called with the optional *thunk* argument.

With the *thunk* argument, `fork` spawns off a subprocess that calls *thunk*, exiting when *thunk* returns. `Fork` returns the subprocess' pid to the parent process.

Without the *thunk* argument, `fork` behaves like the C `fork()` routine. It returns in both the parent and child process. In the parent, `fork` returns the child's pid; in the child, `fork` returns `#f`.

`(fork/pipe` *[thunk]*`)` *procedure*

Like `fork`, but the parent and child communicate via a pipe connecting the parent's stdin to the child's stdout. This function side-effects 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]*`)` *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 ... pf_n`)`

process form for a description of connection lists.

3.5 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. Here are the special forms for the simple variants:

```
(run/port    . epf) ; Return port open on process's stdout.
(run/file    . epf) ; Process > temp file; return file name.
(run/string  . epf) ; Collect stdout into a string and return.
(run/strings . epf) ; Stdout->list of newline-delimited strings.
(run/sexp    . epf) ; Read one sexp from stdout with READ.
(run/sexps   . epf) ; Read list of sexps from stdout with READ.
```

`Run/port` returns immediately after forking off the process; other forms wait for either the process to die (`run/file`), or eof on the communicating pipe (`run/string`, `run/strings`, `run/sexps`). These special forms just expand into calls to the following analogous procedures:

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

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

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

These procedures can be used to manipulate the output of Unix programs with Scheme code. For example, the output of the `xhost(1)` program can be manipulated with the following code:

```
;;; Before asking host REMOTE to do X stuff,
;;; make sure it has permission.
(while (not (member remote (run/strings (xhost))))
  (display "Pausing for xhost...")
  (read-char))
```

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

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

4 System calls

We've just seen scsh's high-level process-form notation, for running programs, creating pipelines, and performing I/O redirection. This notation is at roughly the same level as traditional Unix shells. The process-form notation is convenient, but does not provide detailed, low-level access to the operating system. This is provided by the second component of scsh: its system-call library.

Scsh's system-call library is a nearly-complete set of POSIX bindings, with some extras, such as symbolic links. As of this writing, network and terminal i/o controls have still not yet been implemented; work on them is underway. Scsh also provides a convenient set of systems programming utility procedures, such as routines to perform pattern matching on file-names and general strings, manipulate Unix environment variables, and parse file pathnames. Although some of the procedures have been described in passing, a detailed description of the system-call library is beyond the scope of this note. The reference manual [refman] contains the full details.

5 The Tao of Scheme and Unix

Most attempts at embedding shells in functional programming languages [fsh, Ellis] try to hide the difference between running a program and calling a procedure. That is, if the user tries

```
(lpr "notes.txt")
```

the shell will first treat `lpr` as a procedure to be called. If `lpr` isn't found in the variable environment, the shell will then do a path search of the file system for a program. This sort of transparency is in analogy to the function-binding mechanisms of traditional shells, such as ksh.

This is a fundamental error that has hindered these previous designs. Scsh, in contrast, is explicit about the distinction between procedures and programs. In scsh, the programmer must know which are which—the mechanisms for invocation are different for the two cases (procedure call *versus* the `(run . `*epf*`)` special form), and the namespaces are different (the program's lexical environment *versus* `$PATH` search in the file system).

Linguistically separating these two mechanisms was an important design decision in the language. It was done because the two computational

Unix:	Computational agents are processes, communicate via byte streams.
Scheme:	Computational agents are procedures, communicate via procedure call / return.

Figure 4: The Tao of Scheme and Unix

models are fundamentally different; any attempt to gloss over the distinctions would have made the semantics ugly and inconsistent.

There are two computational worlds here (figure 4), where the basic computational agents are procedures or processes. These agents are composed differently. In the world of applicative-order procedures, agents execute serially, and are composed with function composition: `(g (f x))`. In the world of processes, agents execute concurrently and are composed with pipes, in a data-flow network: `f | g`. A language with both of these computational structures, such as scsh, must provide a way to interface them. {Note Normal order} In scsh, we have “adapters” for crossing between these paradigms:

	Scheme	Unix
Scheme	`(g (f x))`	`(<< ,x)`
Unix	`run/string`,...	`f	g`

The `run/string` form and its cousins (section 3.5) map process output to procedure input; the << i / o redirection maps procedure output to process input. For example:

```
(run/string (nroff -ms)
            (<< ,(texinfo->nroff doc-string)))
```

By separating the two worlds, and then providing ways for them to cross-connect, scsh can cleanly accommodate the two paradigms within one notational framework.

6 I/O

Perhaps the most difficult part of the design of scsh was the integration of Scheme ports and Unix file descriptors. Dealing with Unix file descriptors in a Scheme environment is difficult. In Unix, open files are part of the process state, 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()` and `exec()` calls.

Scheme, on the other hand, uses ports for specifying i/o sources. Ports are anonymous, garbage-collected Scheme objects, not integers. When a port is collected, it is also closed. Because file descriptors are just integers, it's impossible to garbage collect them—in order to close file descriptor 3, you must prove that the process will never again pass a 3 as a file descriptor to a system call doing I/O, and that it will never `exec()` a program that will refer to file descriptor 3.

This is difficult at best.

If a Scheme program only used Scheme ports, and never directly 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 problems. Suppose we have a Scheme port constructed on top of file descriptor 2. We intend to fork off a C program that will inherit this file descriptor. If we drop references to the port, the garbage collector may prematurely close file 2 before we exec the C program.

Another difficulty arising between the anonymity of ports and the explicit naming of file descriptors arises when the user explicitly manipulates file descriptors, as is required by Unix. For example, when a file port is opened in Scheme, the underlying run-time Scheme kernel must open a file and allocate an integer file descriptor. When the user subsequently explicitly manipulates particular file descriptors, perhaps preparatory to executing some Unix subprocess, the port's underlying file descriptor could be silently redirected to some new file.

Scsh's Unix i/o interface is intended to fix this and other problems arising from the mismatch between ports and file descriptors. The fundamental principle is that in scsh, most ports are attached to files, not to particular file descriptors. When the user does an i/o redirection (*e.g.*, with `dup2()`) that must allocate a particular file descriptor *fd*, there is a chance that *fd* has already been inadvertently allocated to a port by a prior operation (*e.g.*, an `open-input-file` call). If so, *fd*'s original port will be shifted to some new file descriptor with a `dup(`*fd*`)` operation, freeing up *fd* for use. The port machinery is allowed to do this as it does not in general reveal which file descriptors are allocated to particular Scheme ports. Not revealing the particular file descriptors allocated to Scheme ports allows the system two important freedoms:

- When the user explicitly allocates a particular file descriptor, the run-time system is free to shuffle around the port/file-descriptor associations as required to free up that descriptor.

- When all pointers to an unrevealed file port have been dropped, the run-time system is free to close the underlying file descriptor. If the user doesn't know which file descriptor was associated with the port, then there is no way he could refer to that i/o channel by its file-descriptor name. This allows scsh to close file descriptors during gc or when performing an `exec()`.

Users *can* explicitly manipulate file descriptors, if so desired. In this case, the associated ports are marked by the run time as "revealed," and are no longer subject to automatic collection. The machinery for handling this is carefully marked in the documentation, and with some simple invariants in mind, follow the user's intuitions. 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.

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.

Further details on the port mechanisms in scsh are beyond the scope of this note; for more information, see the reference manual [refman].

7 Lexical issues

Scsh's lexical syntax is not fully R4RS-compliant in two 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 important for writing shell scripts.

The lexical details of scsh are perhaps a bit contentious. Extending the symbol syntax remains backwards compatible with existing correct R4RS code. Since flags to Unix programs always begin with a dash, not extending the syntax would have required the user to explicitly quote every flag to a program, as in

```
(run (cc "-O" "-o" "-c" main.c)).
```

This is unacceptably obfuscatory, so the change was made to cover these sorts of common Unix flags.

More serious was the decision to make symbols read case-sensitively, which introduces a true backwards incompatibility with R4RS Scheme.

This was a true case of clashing world-views: Unix's tokens are case-sensitive; Scheme's, are not.

It is also 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))
```

8 Implementation

Scsh is currently implemented on top of Scheme 48, a freely-available Scheme implementation written by Kelsey and Rees [S48]. Scheme 48 uses a byte-code interpreter for portability, good code density, and medium efficiency. It is R4RS-compliant, and includes a module system designed by Rees.

The scsh 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. (I would be very interested to see scsh ported to some of the Scheme systems designed to serve as embedded command languages—*e.g.*, elk, esh, or any of the other C-friendly interpreters.)

Scsh scripts currently have a few problems owing to the current Scheme 48 implementation technology.

- Before running even the smallest shell script, the Scheme 48 vm must first load in a 1.4Mb heap image. This i/o load adds a few seconds to the startup time of even trivial shell scripts.

- Since the entire Scheme 48 and scsh runtime is in the form of byte-code data in the Scheme heap, the heap is fairly large. As the Scheme 48 vm uses a non-generational gc, all of this essentially permanent data gets copied back and forth by the collector.

- The large heap size is compounded by Unix forking. If you run a four-stage pipeline, *e.g.*,

```
(run (| (zcat paper.tex.Z)
        (detex)
        (spell)
        (enscript -2r)))
```

> then, for a brief instant, you could have up to five copies of scsh forked into existence. This would briefly quintuple the virtual memory demand placed by a single scsh heap, which is fairly large to begin with. Since all the code is actually in the data pages of the process, the OS can't trivially share pages between the processes. Even if the OS is clever enough to do copy-on-write page sharing, it may insist on reserving enough backing store on disk for worst-case swapping requirements. If disk space is limited, this may overflow the paging area, causing the `fork()` operations to fail.

Byte-coded virtual machines are intended to be a technology that provides memory savings through improved code density. It is ironic that the straightforward implementation of such a byte-code interpreter actually has high memory cost through bad interactions with Unix `fork()` and the virtual memory system.

The situation is not irretrievable, however. A recent release of Scheme 48 allows the pure portion of a heap image to be statically linked with the text pages of the vm binary. Putting static data—such as all the code for the runtime—into the text pages should drastically shorten start-up time, move a large amount of data out of the heap, improve paging, and greatly shrink the dynamic size. This should all lessen the impact of `fork()` on the virtual memory system.

Arranging for the garbage collector to communicate with the virtual memory system with the near-standard `madvise()` system call would further improve the system. Also, breaking the system run-time into separate modules (*e.g.*, bignums, list operations, i/o, string operations, scsh operations, compiler, *etc.*), each of which can be demand-loaded shared-text by the Scheme 48 vm (using `mmap()`), will allow for a full-featured system with a surprisingly small memory footprint.

9 Size

Scsh can justifiably be criticised for being a florid design. There are a lot of features—perhaps too many. The optional arguments to many procedures, the implicit backquoting, and the syntax / procedure equivalents are all easily synthesized by the user. For example, `port->strings`, `run/strings*`, `run/sexp*`, and `run/sexps*` are all trivial compositions and curries of other base procedures. The `run/strings` and `run/sexps` forms are easily written as macros, or simply written out by hand. Not only does scsh provide the basic `file-attributes` procedure (*i.e.*, the `stat()` system call), it also provides a host of derived procedures: `file-owner`, `file-mode`, `file-directory?`, and so forth. Still, my feeling is that it is easier and clearer to read

```
(filter file-directory? (directory-files))
```

than

```
(filter (λ (fname)
          (eq? 'directory
               (fileinfo:type (file-attributes fname))))
        (directory-files))
```

A full library can make for clearer user code.

One measure of scsh's design is that the source code consists of a large number of small procedures: the source code for scsh has 448 top-level definitions; the definitions have an average length of 5 lines of code. That is, scsh is constructed by connecting together a lot of small, composable parts, instead of designing one inflexible monolithic structure. These small parts can also be composed and abstracted by the programmer into his own computational structures. Thus the total functionality of scsh is greater than more traditional large systems.

10 Systems programming in Scheme

Unix systems programming in Scheme is a much more pleasant experience than Unix systems programming in C. Several features of the language remove a lot of the painful or error-prone problems C systems programmers are accustomed to suffering. The most important of these features are:

- exceptions
- automatic storage management
- real strings

- higher-order procedures
- S-expression syntax and backquote

Many of these features are available in other advanced programming languages, such as Modula-3 or ML. None are available in C.

10.1 Exceptions and robust error handling

In scsh, system calls never return the error codes that make careful systems programming in C so difficult. Errors are signaled by raising exceptions. Exceptions are usually handled by default handlers that either abort the program or invoke a run-time debugger; the programmer can override these when desired by using exception-handler expressions. Not having to return error codes frees up procedures to return useful values, which encourages procedural composition. It also keeps the programmer from cluttering up his code with (or, as is all too often the case, just forgetting to include) error checks for every system call. In scsh, the programmer can assume that if a system call returns at all, it returns successfully. This greatly simplifies the flow of the code from the programmer's point of view, as well as greatly increasing the robustness of the program.

10.2 Automatic storage management

Further, Scheme's automatic storage allocation removes the "result" parameters from the procedure argument lists. When composite data is returned, it is simply returned in a freshly-allocated data structure. Again, this helps make it possible for procedures to return useful values.

For example, the C system call `readlink()` dereferences a symbolic link in the file system. A working definition for the system call is given in figure 5b. It is complicated by many small bookkeeping details, made necessary by C's weak linguistic facilities.

In contrast, scsh's equivalent procedure, `read-symlink`, has a much simpler definition (fig. 5a). With the scsh version, there is no possibility that the result buffer will be too small. There is no possibility that the programmer will misrepresent the size of the result buffer with an incorrect `bufsiz` argument. These sorts of issues are completely eliminated by the Scheme programming model. Instead of having to worry about seven or eight trivial but potentially fatal issues, and write the necessary 10 or 15

`(read-symlink fname)`

`read-symlink` returns the filename referenced by symbolic link `fname`. An exception is raised if there is an error.

(a) Scheme definition of `readlink`

`readlink(char *path, char *buf, int bufsiz)`

`readlink` dereferences the symbolic link `path`. If the referenced filename is less than or equal to `bufsiz` characters in length, it is written into the `buf` array, which we fondly hope the programmer has arranged to be at least of size `bufsiz` characters. If the referenced filename is longer than `bufsiz` characters, the system call returns an error code; presumably the programmer should then reallocate a larger buffer and try again. If the system call succeeds, it returns the length of the result filename. When the referenced filename is written into `buf`, it is *not* nul-terminated; it is the programmer's responsibility to leave space in the buffer for the terminating nul (remembering to subtract one from the actual buffer length when passing it to the system call), and deposit the terminal nul after the system call returns.

If there is a real error, the procedure will, in most cases, return an error code. (We will gloss over the error-code mechanism for the sake of brevity.) However, if the length of `buf` does not actually match the argument `bufsiz`, the system call may either

- succeed anyway,
- dump core,
- overwrite other storage and silently proceed,
- report an error,
- or perform some fifth action.

It all depends.

(b) C definition of `readlink`

Figure 5: Two definitions of `readlink`

lines of code to correctly handle the operation, the programmer can write a single function call and get on with his task.

10.3 Return values and procedural composition

Exceptions and automatic storage allocation make it easier for procedures to return useful values. This increases the odds that the programmer can use the compact notation of function composition—`f(g(x))`—to connect producers and consumers of data, which is surprisingly difficult in C.

In C, if we wish to compose two procedure calls, we frequently must write:

```
/* C style: */
g(x,&y);
...f(y)...
```

Procedures that compute composite data structures for a result commonly return them by storing them into a data structure passed by-reference as a parameter. If `g` does this, we cannot nest calls, but must write the code as shown.

In fact, the above code is not quite what we want; we forgot to check `g` for an error return. What we really wanted was:

```
/* Worse/better: */
err=g(x,&y);
if( err ) {
    <handle error on g call>
    }
...f(y)...
```

The person who writes this code has to remember to check for the error; the person who reads it has to visually link up the data flow by connecting `y`'s def and use points. This is the data-flow equivalent of `goto`'s, with equivalent effects on program clarity.

In Scheme, none of this is necessary. We simply write

```
(f (g x)) ; Scheme
```

Easy to write; easy to read and understand. Figure 6 shows an example of this problem, where the task is determining if a given file is owned by root.

```
(if (zero? (fileinfo:owner (file-attributes fname)))
    ...)
```

Scheme

```
if( stat(fname,&statbuf) ) {
    perror(progname);
    exit(-1);
    }
if( statbuf.st_uid == 0 ) ...
```

C

Figure 6: Why we program with Scheme.

10.4 Strings

Having a true string datatype turns out to be surprisingly valuable in making systems programs simpler and more robust. The programmer never has to expend effort to make sure that a string length kept in a variable matches the actual length of the string; never has to expend effort wondering how it will affect his program if a nul byte gets stored into his string. This is a minor feature, but like garbage collection, it eliminates a whole class of common C programming bugs.

10.5 Higher-order procedures

Scheme's first-class procedures are very convenient for systems programming. Scsh uses them to parameterise the action of procedures that create Unix processes. The ability to package up an arbitrary computation as a thunk turns out to be as useful in the domain of Unix processes as it is in the domain of Scheme computation. Being able to pass computations in this way to the procedures that create Unix processes, such as `fork`, `fork/pipe` and `run/port*` is a powerful programming technique.

First-class procedures allow us to parameterise port readers over different parsers, with the

`(port->list` *parser port*`)`

procedure. This is the essential Scheme ability to capture abstraction in a procedure definition. If the user wants to read a list of objects written in some syntax from an i/o source, he need only write a parser capable of parsing a single object. The `port->list` procedure can work with the user's parser as easily as it works with `read` or `read-line`. {Note On-line streams}

First-class procedures also allow iterators such as `for-each` and `filter` to loop over lists of data. For example, to build the list of all my files in `/usr/tmp`, I write:

```
(filter (λ (f) (= (file-owner f) (user-uid)))
        (glob "/usr/tmp/*"))
```

To delete every C file in my directory, I write:

```
(for-each delete-file (glob "*.c"))
```

10.6 S-expression syntax and backquote

In general, Scheme's s-expression syntax is much, much simpler to understand and use than most shells' complex syntax, with their embedded pattern matching, variable expansion, alias substitution, and multiple rounds of parsing. This costs scsh's notation some compactness, at the gain of comprehensibility.

Recursive embeddings and balls of mud

Scsh's ability to cover a high-level/low-level spectrum of expressiveness is a function of its uniform s-expression notational framework. Since scsh's process notation is embedded within Scheme, and Scheme escapes are embedded within the process notation, the programmer can easily switch back and forth as needed, using the simple notation where possible, and escaping to system calls and general Scheme where necessary. This recursive embedding is what gives scsh its broad-spectrum coverage of systems functionality not available to either shells or traditional systems programming languages; it is essentially related to the "ball of mud" extensibility of the Lisp and Scheme family of languages.

Backquote and reliable argument lists

Scsh's use of implicit backquoting in the process notation is a particularly nice feature of the s-expression syntax. Most Unix shells provide the user with a way to take a computed string, split it into pieces, and pass them as arguments to a program. This usually requires the introduction of some sort of `$IFS` separator variable to control how the string is parsed into separate arguments. This makes things error prone in the cases where a single argument might contain a space or other parser delimiter. Worse than error prone, `$IFS` rescanning is in fact the source of a famous security hole in Unix [Reeds].

In scsh, data are used to construct argument lists using the implicit backquote feature of process forms, *e.g.*:

```
    (run (cc ,file -o ,binary ,@flags)).
```

Backquote completely avoids the parsing issue because it deals with preparsed data: it constructs expressions from lists, not character strings. When the programmer computes a list of arguments, he has complete confidence that they will be passed to the program exactly as is, without running the risk of being re-parsed by the shell.

11 Other programming languages

Having seen the design of scsh, we can now compare it to other approaches in some detail.

11.1 Functional languages

The design of scsh could be ported without much difficulty to any language that provides first-class procedures, GC, and exceptions, such as COMMON LISP or ML. However, Scheme's syntactic extensibility (macros) plays an important role in making the shell features convenient to use. In this respect, Scheme and COMMON LISP are better choices than ML. Using the `fork/pipe` procedure with a series of closures involves more low-level detail than using scsh's `(|` pf_1 ... pf_n`)` process form with the closures implied. Good notations suppress unnecessary detail.

The payoff for using a language such as ML would come not with small shell scripts, but with larger programs, where the power provided by the module system and the static type checking would come into play.

11.2 Shells

Traditional Unix shells, such as sh, have no advantage at all as scripting languages.

Escaping the least common denominator trap

One of the attractions of scsh is that it is a Unix shell that isn't constrained by the limits of Unix's uniform "least common denominator" representation of data as a text string. Since the standard medium of interchange at the shell level is ASCII byte strings, shell programmers are forced to parse and reparse data, often with tools of limited power. For example, to determine the number of files in a directory, a shell programmer typically uses an expression of the form `ls | wc -l`. This traditional idiom is in fact buggy: Unix files are allowed to contain newlines in their names, which would defeat the simple `wc` parser. Scsh, on the other hand, gives the programmer direct access to the system calls, and employs a much richer set of data structures. Scsh's `directory-files` procedure returns a *list* of strings, directly taken from the system call. There is no possibility of a parsing error.

As another example, consider the problem of determining if a file has its setuid bit set. The shell programmer must grep the text-string output of `ls -l` for the "s" character in the right position. Scsh gives the programmer direct access to the `stat()` system call, so that the question can be directly answered.

Computation granularity and impedance matching

Sh and csh provide minimal computation facilities on the assumption that all real computation will happen in C programs invoked from the shell. This is a granularity assumption. As long as the individual units of computation are large, then the cost of starting up a separate program is amortised over the actual computation. However, when the user wants to do something simple—*e.g.*, split an X `$DISPLAY` string at the colon, count the number of files in a directory, or lowercase a string—then the overhead of program invocation swamps the trivial computation being performed. One advantage of using a real programming language for the shell language is that we can get a wider-range "impedance match" of computation to process overhead. Simple computations can be done in the shell; large grain computations can still be spawned off to other programs if necessary.

11.3 New-generation scripting languages

A newer generation of scripting languages has been supplanting sh in Unix. Systems such as perl and tcl provide many of the advantages of scsh for programming shell scripts [perl, tcl]. However, they are still limited by weak linguistic features. Perl and tcl still deal with the world primarily in terms of strings, which is both inefficient and expressively limiting. Scsh makes the full range of Scheme data types available to the programmer: lists, records, floating point numbers, procedures, and so forth. Further, the abstraction mechanisms in perl and tcl are also much more limited than Scheme's lexically scoped, first-class procedures and lambda expressions. As convenient as tcl and perl are, they are in no sense full-fledged general systems-programming languages: you would not, for example, want to write an optimizing compiler in tcl. Scsh is Scheme, hence a powerful, full-featured general programming tool.

It is, however, instructive to consider the reasons for the popular success of tcl and perl. I would argue that good design is necessary but insufficient for a successful tool. Tcl and perl are successful because they are more than just competently designed; critically, they are also available on the Net in turn-key forms, with solid documentation. A potential user can just down-load and compile them. Scheme, on the other hand, has existed in multiple mutually-incompatible implementations that are not widely portable, do not portably address systems issues, and are frequently poorly documented. A contentious and standards-cautious Scheme community has not standardised on a record datatype or exception facility for the language, features critical for systems programming. Scheme solves the hard problems, but punts the necessary, simpler ones. This has made Scheme an impractical systems tool, banishing it to the realm of pedagogical programming languages. Scsh, together with Scheme 48, fills in these lacunae. Its facilities may not be the ultimate solutions, but they are useable technology: clean, consistent, portable and documented.

12 Future work

Several extensions to scsh are being considered or implemented.

12.1 Command language features

The primary design effort of scsh was for programming. We are now designing and implementing features to make scsh a better interactive command language, such as job control. A top-level parser for an sh-like notation has been designed; the parser will allow the user to switch back to Scheme notation when desired.

We are also considering a display-oriented interactive shell, to be created by merging the edwin screen editor and scsh. The user will interact with the operating system using single-keystroke commands, defining these commands using scsh, and reverting to Scheme when necessary for complex tasks. Given a reasonable set of GUI widgets, the same trick could be played directly in X.

12.2 Little languages

Many Unix tools are built around the idea of "little languages," that is, custom, limited-purpose languages that are designed to fit the area of application. The problem with the little-languages approach is that these languages are usually ugly, idiosyncratic, and limited in expressiveness. The syntactic quirks of these little languages are notorious. The well-known problem with `make`'s syntax distinguishing tab and space has been tripping up programmers for years. Because each little language is different from the next, the user is required to master a handful of languages, unnecessarily increasing the cognitive burden to use these tools.

An alternate approach is to embed the tool's primitive operations inside Scheme, and use the rest of Scheme as the procedural glue to connect the primitives into complex systems. This sort of approach doesn't require the re-invention of all the basic functionality needed by a language—Scheme provides variables, procedures, conditionals, data structures, and so forth. This means there is a greater chance of the designer "getting it right" since he is really leveraging off of the enormous design effort that was put into designing the Scheme language. It also means the user doesn't have to learn five or six different little languages—just Scheme plus the set of base primitives for each application. Finally, it means the base language is not limited because the designer didn't have the time or resources to implement all the features of a real programming language.

With the scsh Unix library, these "little language" Unix tools could easily be redesigned from a Scheme perspective and have their interface and functionality significantly improved. Some examples under consideration are:

- The awk pattern-matching language can be implemented in scsh by adding a single record-input procedure to the existing code.
- Expect is a scripting language used for automating the use of interactive programs, such as ftp. With the exception of the tty control syscalls currently under construction, all the pieces needed to design an alternate scsh-based Unix scripting tool already exist in scsh.
- A dependency-directed system for controlling recompilation such as make could easily be implemented on top of scsh. Here, instead of embedding the system inside of Scheme, we embed Scheme inside of the system. The dependency language would use s-expression notation, and the embedded compilation actions would be specified as Scheme expressions, including scsh notation for running Unix programs.

13 Conclusion

Scsh is a system with several faces. From one perspective, it is not much more than a system-call library and a few macros. Yet, there is power in this minimalist description—it points up the utility of embedding systems in languages such as Scheme. Scheme is at core what makes scsh a successful design. Which leads us to three final thoughts on the subject of scsh and systems programming in Unix:

- A Scheme shell wins because it is broad-spectrum.
- A functional language is an excellent tool for systems programming.
- Hacking Unix isn't so bad, actually, if you don't have to use C.

14 Acknowledgements

John Ellis' 1980 *SIGPLAN Notices* paper [Ellis] got me thinking about this entire area. Some of the design for the system calls was modeled after Richard Stallman's emacs [emacs], Project MAC's MIT Scheme [MIT Scheme], and COMMON LISP [CLtL2]. Tom Duff's Unix shell, rc, was also inspirational;

his is the only elegant Unix shell I've seen [rc]. Flames with Bennet Yee and Scott Draves drove me to design scsh in the first place; polite discussions with John Ellis and Scott Nettles subsequently improved it. Douglas Orr was my private Unix kernel consultant. Richard Kelsey and Jonathan Rees provided me with twenty-four hour turnaround time on requested modifications to Scheme 48, and spent a great deal of time explaining the internals of the implementation to me. Their elegant Scheme implementation was a superb platform for development. The design and the major portion of the implementation of scsh were completed while I was visiting on the faculty of the University of Hong Kong in 1992. It was very pleasant to work in such a congenial atmosphere. Doug Kwan was a cooperative sounding-board during the design phase. Hsu Suchu has patiently waited quite a while for this document to be finished. Members of the MIT LCS and AI Lab community encouraged me to polish the research prototype version of the shell into something releasable to the net. Henry Minsky and Ian Horswill did a lot of the encouraging; my students Dave Albertz and Brian Carlstrom did a lot of the polishing.

Finally, the unix-haters list helped a great deal to maintain my perspective.

References

[CLtL2] Guy L. Steele Jr.
Common Lisp: The Language.
Digital Press, Maynard, Mass., second edition 1990.

[Ellis] John R. Ellis.
A Lisp shell.
SIGPLAN Notices, 15(5):24–34, May 1980.

[emacs] Bil Lewis, Dan LaLiberte, Richard M. Stallman, *et al.*
The GNU Emacs Lisp Reference Manual, vol. 2.
Free Software Foundation, Cambridge, Mass., edition 2.1 September 1993. (Also available from many ftp sites.)

[fsh] Chris S. McDonald.
fsh—A functional Unix command interpreter.
Software—Practice and Experience, 17(10):685–700, October 1987.

[MIT Scheme] Chris Hanson.
MIT Scheme Reference Manual.
MIT Artificial Intelligence Laboratory Technical Report 1281, January 1991. (Also URL `http://martigny.ai.mit.edu/emacs-html.local/scheme_toc.html`)

[Nelson] Greg Nelson, ed.
Systems Programming with Modula-3.
Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[perl] Larry Wall and Randal Schwartz.
Programming Perl.
O'Reilly & Associates.

[rc] Tom Duff.
Rc—A shell for Plan 9 and Unix systems.
In *Proceedings of the Summer 1990 UKUUG Conference*, pages 21–33, July 1990, London. (A revised version is reprinted in “Plan 9: The early papers,” Computing Science Technical Report 158, AT&T Bell Laboratories. Also available in Postscript form as URL `ftp://research.att.com/dist/plan9doc/7`.)

[Reeds] J. Reeds.
`/bin/sh`: the biggest UNIX security loophole.
11217-840302-04TM, AT&T Bell Laboratories (1988).

[refman] Olin Shivers.
Scsh reference manual.
In preparation.

[S48] Richard A. Kelsey and Jonathan A. Rees.
A tractable Scheme implementation.
To appear, *Lisp and Symbolic Computation*, Kluwer Academic Publishers, The Netherlands. (Also URL `ftp://altdorf.ai.mit.edu/pub/jar/lsc.ps`)

[tcl] John K. Ousterhout.
Tcl: An embeddable command language.
In *The Proceedings of the 1990 Winter USENIX Conference*, pp. 133–146. (Also URL `ftp://ftp.cs.berkeley.edu/ucb/tcl/tclUsenix90.ps`)

Notes

{Note Agenda}

In fact, I have an additional hidden agenda. I do believe that computational agents should be expressed as procedures or procedure libraries, not as programs. Scsh is intended to be an incremental step in this direction, one that is integrated with Unix. Writing a program as a Scheme 48 module should allow the user to make it available as a both a subroutine library callable from other Scheme 48 programs or the interactive read-eval-print loop, and, by adding a small top-level, as a standalone Unix program. So Unix programs written this way will also be useable as linkable subroutine libraries—giving the programmer module interfaces superior to Unix's "least common denominator" of ASCII byte streams sent over pipes.

{Note No port sync}

In scsh, Unix' stdio file descriptors and Scheme's standard i/o ports (*i.e.*, the values of `(current-input-port)`, `(current-output-port)` and `(error-output-port)`) are not necessarily synchronised. 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.

{Note Normal order}

Having to explicitly shift between processes and functions in scsh is in part due to the arbitrary-size nature of a Unix stream. A better, more integrated approach might be to use a lazy, normal-order language as the glue or shell language. Then files and process output streams could be regarded as first-class values, and treated like any other sequence in the language. However, I suspect that the realities of Unix, such as side-effects, will interfere with this simple model.

{Note On-line streams}

The `(port->list` *reader port*`)` procedure is a batch processor: it reads the port all the way to eof before returning a value. As an alternative, we might write a procedure to take a port and a reader, and return a lazily-evaluated list of values, so that I/O can be interleaved with element processing. A nice example of the power of Scheme's abstraction facilities is the ease with which we can write this procedure: it can be done with five lines of code.

```
;;; A <lazy-list> is either
;;;     (delay '()) or
;;;     (delay (cons data <lazy-list>)).

(define (port->lazy-list reader port)
  (let collector ()
    (delay (let ((x (reader port)))
             (if (eof-object? x) '()
                 (cons x (collector)))))))
```

{Note Tempfile example}

For a more detailed example showing the advantages of higher-order procedures in Unix systems programming, consider the task of making random temporary objects (files, directories, fifos, *etc.*) in the file system. Most Unix's simply provide a function such as `tmpnam()` that creates a file with an unusual name, and hope for the best. Other Unix's provide functions that avoid the race condition between determining the temporary file's name and creating it, but they do not provide equivalent features for non-file objects, such as directories or symbolic links. This functionality is easily generalised with the procedure

`(temp-file-iterate` *maker [template]*`)`

This procedure can be used to perform atomic transactions on the file system involving filenames, *e.g.*:

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

The string *template* is a `format` control string used to generate a series of trial filenames; it defaults to

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

where `<pid>` is the current process' process id. Filenames are generated by calling `format` to instantiate the template's `~a` field with a varying string. (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.)

The `maker` procedure is serially called on each filename 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 "file already exists" syscall error exception, `temp-file-iterate` will loop, generating a new filename and calling `maker` again. If the first return value is true, the loop is terminated, returning whatever `maker` returned.

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

To rename a file to a temporary name, we write:

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

Note the guarantee: if `temp-file-iterate` returns successfully, then the hard link was definitely created, so we can safely delete the old link with the following `delete-file`.

To create a unique temporary directory, we write:

```
(temp-file-iterate (λ (dir) (create-directory dir) dir))
```

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).