A Universal Scripting Framework
or
Lambda: the ultimate "little language"

Olin Shivers

MIT AI Lab, Cambridge, Mass. 02139, USA

Abstract. The "little languages" approach to systems programming is flawed: inefficient, fragile, error-prone, inexpressive, and difficult to compose. A better solution is to embed task-specific sublanguages within a powerful, syntactically extensible, universal language, such as Scheme. I demonstrate two such embeddings that have been implemented in scsh, a Scheme programming environment for Unix systems programming. The first embedded language is a high-level process-control notation; the second provides for Awk-like processing. Embedding systems in this way is a powerful technique: for example, although the embedded Awk system was implemented with 7% of the code required for the standard C-based Awk, it is significantly more expressive than its C counterpart.

1 Introduction

Many programming tools are built around the idea of "little languages"—small interpreters implementing a programming language that has been tuned to the specifics of some specialised task domain. This approach to systems-building was popularised by Unix, which provides a host of little-language processors. For example, the following Unix language interpreters all support notations tuned for specialised task domains:

Task	Interpreter
regular-expression based string transforms	sed
pattern-matching	awk
type-setting	nroff/tbl/eqn
dependency-directed recompilation	make
file-system tree-walking	find
program invocation and composition	sh

Little languages complement the Unix "toolkit" philosophy—the operating system provides mechanisms for composing little-language based components into larger systems (and, in fact, the principal interface for doing so is itself a little language, the shell).

In *Concurrency and Parallelism, Programming, Networking, and Security,* Lecture Notes in Computer Science #1179, pages 254–265, Editors Joxan Jaffar and Roland H. C. Yap, 1996, Springer.

2 The Case for Little Languages

A programming language is a notation for expressing computation. By restricting ourselves to a specific task domain, we can tune our notation to the needs of the domain's computations. The language understood by the `make` utility, for example, is not a general programming language, but rather one which is specially adapted for expressing dependencies among the components of a system [3]. Sacrificing generality is rewarded by notational compactness and clarity.

Unix's little-languages philosophy is to present the programmer with a suite of specialised languages—a linguistic toolkit for systems implementation. The programmer can then write each component of his system in the language which is best suited to the requirements of the component. Once constructed, the components are then composed together using pipelines.

In principle, this approach provides a flexible and powerful method for constructing systems. In practice, however, little languages have a number of problems.

3 The Problems of Little Languages

One problem with the little-languages approach is that these languages are usually ugly, idiosyncratic, and limited in expressiveness. Although many of these little languages are similar to C, they are all slightly different from one another. Because each little language is different from the next, the user is required to master a handful of languages, remembering all the trivial distinctions between them, thus unnecessarily increasing the cognitive burden to use these tools.

The bizarre syntactic quirks of little languages are notorious. For example, the well-known problem with `make`'s syntax distinguishing tab and space has been tripping up programmers for years.

When a programmer decides to implement a little language, he must start from scratch, and implement an entire programming language. The designer and implementor is unable to concentrate solely on the task-specific aspects of his design. Basic linguistic elements such as loops, conditionals, variables, and subroutines must be reinvented and re-implemented. Not only does this add to confusing and unnecessary linguistic proliferation, it is not an approach that is likely to produce a high-quality language design. The temptation is to hurry through these “details” to get to the “interesting parts” of the design—the task-specific elements. So the basic programming substrate is likely to be hastily thought-out and implemented, and the task-specific elements of the design are denied the implementor's full consideration. What is produced, all too often, is a little language with a half-baked variable scoping discipline, weak procedural facilities, and a limited set of data types.

Finally, by implementing a little language as a standalone interpreter that executes in a separate address space, the system component implemented in that little language must typically be implemented as a standard-input/standard-output text transducer that interacts with other system components “at arms length” through pipes. Components do not have an opportunity to interact in ways that require sophisticated data structures or patterns of control transfer. Forcing data to be communicated from component to component as a linear byte-stream implies a cost of repeated parsing and unparsing operations at each component interface. In practice, it often leads to fragile programs that rely on heuristic, error-prone parsers (often based on limited regular-expression matchers).

4 An Alternate Approach

An alternate approach is to choose a powerful, syntactically extensible programming language, such as Scheme [6], and embed our little language within it. This has the benefit of focus: it allows the tool designer and implementor to concentrate on the task-specific elements of his language. Standard programming constructs, such as loops, variables, procedures, conditionals, data structures, and so forth, can be taken wholesale from the underlying "glue" language when needed. This means:

- There is a greater chance of the designer getting the basics right since he is really leveraging off of the enormous design effort that was put into designing the Scheme language.
- Not only is the designer able to exploit the design effort represented by the Scheme language, the implementor can also capitalise on the efforts of others. By embedding within Scheme, he gets interpreters and native-code compilers for free.
- The designer's task is much, much easier because he doesn't have to start from scratch; he can devote all of his time and thought to the task-specific elements of his little language.
- 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.
- The user doesn't have to learn five or six different little languages—just Scheme plus the set of base primitives for each application.
- Because different little languages can be embedded within Scheme, components written in different little languages can now interact using the sophisticated data structures and patterns of control-flow available in Scheme.

As exemplars of this approach, I have embedded two "little languages" within Scheme. The first is a high-level process-control notation equivalent to the notation provided by Unix shells for constructing pipelines of processes, performing I/O redirection, and so forth. The second, more detailed example, will be an embedding of an Awk-like little language within Scheme.

These systems were implemented in scsh, a portable Unix programming environment built in Scheme [7, 8]. Scsh has been used for a wide variety of systems programming tasks, such as cell-phone interfaces, mobile Web browsers, CAD tools, log analysers, http servers and other network tools. Its chief relevance is that it provides an interface from Scheme to the underlying operating system, and so makes a suitable platform for experimenting with embedding systems-oriented little languages within Scheme.

5 The Scsh Process Notation

Scsh has a notation for controlling 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*.

5.1 Extended Process Forms and I/O Redirections

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

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

where *pf* is a process form and the *redir*$_i$ are redirection specs. There are seven types of redirection spec; the most common four are:

```
(< [fdes] file-name)      ; Open file for read.
(> [fdes] file-name)      ; Open file for write.
(<< [fdes] object)        ; Use object's printed representation.
(>> [fdes] file-name)     ; Open file for append.
```

The *fdes* file descriptors default to 0 and 1 for input and output redirections, respectively.

The subforms of a redirection are implicitly backquoted. So the output redirection `(> ,x)` means "output to the file named by Scheme variable `x`," and the input redirection `(< /usr/shivers/.login)` means "read from `/usr/shivers/.login`." This implicit backquoting is an important feature of the process notation, as it provides escapes to general Scheme computation from within the little language.

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 standard input, and the file `/tmp/buf` to be opened for append writes on standard error.

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 standard input to produce the characters of the above string. Note that *object* is also implicitly backquoted, so we can connect a computed Scheme string up to a process' standard input, *e.g.*:

```
(<< ,(append s1 (f x 7) s2))
```

5.2 Process Forms

A *process form* specifies a computation to perform as an independent Unix process. There are six types of process form; the most common three are:

```
(begin . scheme-code)      ; Run scheme-code in a fork.
(| pf1 ... pfn)            ; Simple pipeline
(prog arg1 ... argn)       ; Default: exec the program.
```

The default case `(`*prog* *arg*$_1$ `...` *arg*$_n$`)` is also implicitly backquoted. That is, variables can be substituted into the command line with `,`*exp* or `,@`*exp* forms.

5.3 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)         ; Fork epf in background and return pid.
(run . epf)       ; Run epf and return 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))
```

Figure 1 shows a series of examples employing a mix of scsh's process notation and Scheme. Note that regular Scheme is used to provide the control structure, variables, and other linguistic machinery needed by the script fragments; this was made possible by our strategy of embedding our specialised notation within Scheme.

6 Awk in Scheme

Our second case study is the Awk language for pattern-directed processing [1]. The first part of the design task was to factor the language into its two basic components: a record- and field-based I/O system, and a rule-based, pattern-directed control structure. Each of these components was implemented separately in the Scheme design. This allows us to mix and match the components—a given task may only require one of the two components. Also, this factoring allows the user a degree of extensibility—if the provided record I/O library doesn't serve his needs, he can use the full power of Scheme to implement his own I/O primitives, and plug them into the pattern-matching control structure in a modular fashion.

6.1 Field and Record Readers

Awk programs iterate over a sequence of records, processing each in turn. The C-based Awk has limited support for specifying the record and field structure of the input stream. If the programmer's data has structure that is too complex to be parsed with these simple mechanisms, he is out of luck. The Scheme-based design, however, provides an extensible toolkit for constructing procedures to read records and parse them into their component fields. This toolkit makes it easy to build record readers for the typical cases, *but in no way limits the programmer from constructing and using his own, arbitrarily complex, input parsers.*

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

(run (crypt ,key) (< mbox.crypt) (> mbox)) ; Decrypt my mailbox.

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

;; 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))

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

Fig. 1. Example process-notation fragments

Reading Records The basic function for constructing a record reader is

`(record-reader` *[delims elide?]*`)` $\rightarrow$ *reader*

The `record-reader` function constructs a reader procedure which reads delimited records from an input stream. The optional *delims* parameter is a string of delimiter characters—any of these characters will terminate a record. If the optional *elide?* parameter is true, then a contiguous sequence of delimiter characters is taken to mean a single record boundary. The default values of *delims* and *elide?* are newline and false, respectively, so the expression `(record-reader)` constructs a simple line-reading procedure:

```
(define read-line (record-reader))
```

Notice that the `record-reader` procedure doesn't read records itself—it constructs record readers for later use.

Note also that there is nothing special about `record-reader`. The Awk programmer is not limited by the span of record readers it provides. As we will see, he is free to use general Scheme to define his own arbitrary record readers—a facility unavailable to the client of the C-based Awk.

Parsing Records into Fields Once an Awk program has obtained a record to process, it then parses the record into its component fields. Our Awk system provides three functions for constructing field parsers. These procedures allow the programmer to define a record's field structure in terms of regular expressions:

(`field-splitter` *[regexp num-fields]*) $\rightarrow$ *parser*
(`infix-splitter` *[regexp num-fields]*) $\rightarrow$ *parser*
(`suffix-splitter` *[regexp num-fields]*) $\rightarrow$ *parser*

The constructed parsers map a record string to a list of field strings. The regular expression passed to the `field-splitter` function defines the fields to be parsed from the record. For example, the parser

```
(field-splitter "[0-9]+")
```

produces a function that extracts digit sequences from a string:

```
(define s "Yale beat harvard 27 to 3.")
((field-splitter "[0-9]+") s) → ("27" "3")
```

The three functions differ in their interpretation of the *regexp* parameter:

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

The regular expression passed to the `infix-splitter` function is used to match field separators. So the parser `(infix-splitter ":")` will split a line of text into colon-separated components. The optional *num-fields* parameter is used to specify tight or lower bounds on how many fields must be parsed from an input record; its default is just to parse as many as possible. So we can simply define a parser for the Unix `/etc/passwd` file with the expression `(infix-splitter ":" 7)`.

Composing Record Readers and Field Parsers The function `field-reader` is used to compose a record reader with a field parser:

(`field-reader` *[field-parser record-reader]*) $\rightarrow$ *reader*

When the returned reader is applied to an input stream, it uses *record-reader* to read a record from the stream, then applies *field-parser* to the record to break it into fields. The reader then returns two values: the raw, unparsed record, and the list of parsed fields. Both parameters are optional; the default record reader is `read-line`, and the default field parser splits strings at white-space boundaries. Therefore, if `p` is bound to an input stream open on the `/etc/passwd` file, the expression

```
((field-reader (infix-splitter ":" 7)) p)
```

might return the two values

```
"ctkwan:mx3Uaqq0:107:22:Doug Kwan:/usr/ctkwan:/bin/sh"
("ctkwan" "mx3Uaqq0" "107" "22" "Doug Kwan"
          "/usr/ctkwan" "/bin/sh")
```

Figure 2 gives some examples of field readers that can be simply constructed with Awk's constructors. Notice how we have exploited Scheme's higher-order procedures to succinctly construct a wide array of input procedures. Again, as we shall see, the Awk programmer is not limited to this particular set of readers and parsers—he has the full power of Scheme to define his own arbitrary readers and parsers.

6.2 The Awk Loop

Besides field parsing, Awk also provides a rule-based, pattern-directed control structure for determining what to do with records as they are read into the program. Awk's basic control structure is to read in a record, parse it into its component fields, and then match the record against a set of pattern/action rules. If a rule's pattern matches the record, it fires its action part, updating iteration variables and performing output. After all rules have had a chance to fire, the program loops, reading another record, and so forth.

In our Scheme-embedded Awk, this control-structure is realised as a macro. The general syntax of our new control structure is

(awk *next-record record&field-vars state-var-decls*
  clause$_1$ ... *clause*$_n$)

For example, here is a very simple Awk loop:

```
(awk (read-line) (line) ()              ; Strip blank lines
  ("[^ \t]" (display line) (newline))) ; from input.
```

The first form in the awk expression, (read-line), is an expression which is evaluated on each iteration to produce the next record to be processed. The next form is a

```
;;; ls -l output reader, e.g.
;;;   -rw-r--r--  1 shivers    22880 Sep 24 12:45 scsh.scm
(field-reader (infix-splitter "[ \t]+" 8))

;;; Internet IP address reader, e.g.
;;;   18.24.0.241
(field-reader (field-splitter "[^.]+" 4))

;;; Line of decimal integers, e.g.
;;;   73 72 84 70 80
(let ((parser (field-splitter "[+-]?[0-9]+")))
  (field-reader (λ (s) (map string->number (parser s)))))
```

Fig. 2. Some examples of field readers

list of variables—one for each value returned by the *next-record* expression. Since the `read-line` procedure in our example only returns a single value, we only have one variable, `line`. This loop has only one pattern/action clause, whose test is the regular expression `"[^ \t]"`. When the loop variable `line` contains at least one non-blank character, it matches the regular expression. This fires the clause body, which prints out the line.

A clause's test expression doesn't have to be a string denoting a regular expression. It can be a general Scheme expression, which is evaluated to produce a boolean value:

```
;;; Print every line longer than 80 chars.
(awk (read-line) (line) ()
  ((> (string-length line) 80)
   (display line)
   (newline)))
```

Note that we can use the full power of Scheme to encode arbitrarily complex tests, something we cannot do with the limited “little language” provided by the C-based Awk.

`Awk` loops can have iteration variables, which are declared and initialised in the `let`-style *state-var-decls* part of the form. When iteration variables are used, each clause must return the new values for the iteration variables when it fires. So if a loop has three iteration variables, each clause must return three values—one for each iteration variable. The `awk` expression's value is the final value of the iteration variable (if the loop has multiple iteration variables, it produces multiple values).

Here are two examples of `awk` expressions that use loop state:

```
;;; Find the length of the longest line.
(awk (read-line) (line) ((max-len 0))
  (#t (max max-len (string-length line))))

;;; Count the number of non-comment lines
;;; of code in my Scheme source.
(awk (read-line) (line) ((nlines 0))
  ("^[ \t]*;" nlines)           ; Comment
  (else       (+ nlines 1)))    ; Not a comment
```

We haven't bothered yet to parse our records into fields, as none of our simple examples have required it. It is a simple matter, however, to employ a field parser in an `awk` loop when necessary. As an example, we can process the Unix `/etc/passwd` file. For our purposes, we only need to know that this file contains a sequence of newline-terminated records; each record contains seven colon-separated fields; and the first field is the user's login id. In this example, the `read-passwd` procedure returns two values each time it is called: a line of text (bound to `line`), and its field-parse as a seven-element list of strings (bound to `fields`). The `awk` expression produces a list of elements, where each entry in `/etc/passwd` is tagged by the user's login id. This list is sorted by the login-id key, and then the sorted entries are printed out.

```
(define $ nth)  ; A convenient abbreviation.

(define read-passwd (field-reader (infix-splitter ":" 7)))

;;; Sort the entries in /etc/passwd by login name.
(for-each (λ (entry) (display (cdr entry)) (newline))
          (sort (λ (x y) (string<? (car x) (car y)))
                (awk (read-passwd) (line fields) ((ans '()))
                  (#t (cons (cons ($ fields 0) line)
                            ans)))))
```

Note that `awk` is not restricted to performing output to express its result. It is a general Scheme expression, and can be embedded within a larger Scheme context. The three main computations in this example—parsing, sorting, and output—were composed with function composition in Scheme, instead of via pipes as the classical "Unix tools" approach would employ with the C-based Awk utility. Functional composition is a much more powerful technique for composition, as it allows for sophisticated data structures, such as linked lists, to be passed from component to component efficiently and reliably.

7 The Methodology of Embedding

Both the Scheme-embedded process notation and Awk sublanguages have extra features we haven't covered—for example, the process notation provides ways to capture and parse process output into Scheme data, and the `awk` macro supports rule patterns that are active over designated ranges of records. However, we've seen enough of these two little languages to get a feel for the general technique of embedding a sublanguage within Scheme.

In both cases, we followed the same methodology. The first step for our embedding was to functionally decompose the system into its primitive computational elements. These can be realised as base procedures and their associated data structures. In the case of Awk, this was captured by the library of record readers and field parsers. In the case of scsh, this was captured by binding the Unix system calls for I/O and process control into Scheme.

The second step of our embedding was to capture unusual patterns of control and environment structure as macros, providing gains in notational compactness and clarity. This step is what distinguishes Scheme from languages such as Tcl or Perl—the ability to introduce new forms of notation. In Scheme, we can make these extensions all within a common s-expression framework. Although Tcl and Perl are useful scripting languages, one cannot extend their syntax to embed a new language within them.

The `awk` macro and the scsh process notation are examples of this "notational engineering." In both cases, we were careful to allow escapes within the notation to allow general Scheme code to extend the set of allowed computations. For example, scsh's implicit backquoting of its I/O redirections allow the programmer to use general Scheme expressions for I/O redirection; the `begin` process form allows general Scheme computations to serve as the process specification; and the `<<` redirection cross-connects

Scheme and general process computations. In the `awk` design, we allow any Scheme form at all to be employed as the input record reader, and both the test and body elements of the clauses can be arbitrary Scheme code. These escapes allow us to have the best of both worlds: the compactness and simple clarity of task-specific notation, and the generality of a general-purpose programming language. The Scheme escapes keep us from being trapped by our specialised notations. For example, if we wanted to write an Awk loop that scanned over files of C source code, operating on a single top-level function definition at a time, we could write a general C parser in Scheme and use it as the loop's record reader. This is not a possibility with the traditional Awk; we are restricted to the limited class of built-in record readers it provides.

8 The Benefits of Embedding

By now, the benefits of embedding a sublanguage within Scheme should be clear. Embedding a little language within Scheme gets its power from three sources:

– It is, of course, much *easier* to embed a sublanguage within Scheme than to invent and implement a new language out of whole cloth.
  For example, the Gnu project's C implementation of Awk is about 14,000 lines of code. The Scheme-embedded implementation is 933 lines of code, of which roughly one-third are the field-reader library, one-third are the `awk` macro, and one-third are comments. It took about three days to implement the system. Things were this easy mostly because of the things that *didn't* have to be designed or implemented: variables, conditionals, arithmetic operators, procedures, and the rest of the machinery that typical programming languages need.
  Lowering the barrier to invent and embed new little languages within Scheme means this linguistic tool is more generally available to the programmer for application within specialised niches.
– The *quality* of the little language produced is greater.
  Even though the embedded Awk macro required only 7% as many lines of code to implement as the standalone C version, it is significantly more powerful.
  • Whenever necessary, the user can break out of the special-purpose notation and express complex computations in a general-purpose programming language. The Scheme embedding makes simple things easy, and complex things possible. The standalone little language only provides the former.
  • The general-purpose computational facilities provided by Scheme are far more powerful than those provided by the C-based interpreters that implement traditional Awk or the Unix shell.

  In short, Scheme is always available, which is a tremendous source of power.
– System components *compose* better.
  Embedding subsystems within a single broad-spectrum language, such as Scheme, makes it possible for them to be more intimately intertwined. Instead of interacting via flat text streams sent through pipes, components written in different little languages can pass complex data structures around. For example, uses of the process notation can be embedded within Awk scripts, or *vice versa*. In fact, current scsh programmers do exploit the ability to compose system components in this way.

9 Final Thoughts

The technique of embedding little languages within Scheme is a fairly general one. Having embedded two little languages within Scheme, it's not too hard to see how to embed twenty more. Two obvious candidates are the `expect` programmed-dialogue scripting tool [4], and the `make` utility for dependency-directed program recompilation [3]. Since the power of a glue language is the power of the interaction that it facilitates, this is a case of "the more the better"—the possibilities for interaction between system components written in embedded sublanguages go up quadratically with the number of systems that are embedded within Scheme.

We aren't restricted to using Scheme to embed our little languages. We could have used almost any member of the Lisp family of programming languages, including Common Lisp [9] or the original s-expression Dylan [2]. While Scheme has the most sophisticated macro system of any language in this family, the macro systems of the other members can be exploited in a similar manner.

A final note on the power of syntactic extension: It seems curious in the mid-nineties to be using Scheme as a serious tool for systems programming. The current standard lacks a module system, static type-checking, exceptions, and record types—it would appear that time has passed the language by. Scheme's sole remaining virtue distinguishing it from more modern languages, such as ML [5], is its unusually sophisticated macro system. But it is the syntactic extensibility provided by this macro system that provided us the crucial mechanism we needed to embed our little languages within Scheme. Designers of new programming languages would do well to note the power and utility of this mechanism.

References

1. Alfred V. Aho, Brian W. Kernighan, Peter J. Weinberger. Awk—a pattern scanning and processing language. *Software—Practice and Experience*, IX(4):267–279, April, 1979.
2. *Dylan: An Object-Oriented Dynamic Language.* Apple Computer, 1992.
3. Stuart I. Feldman. Make—A program for maintaining computer programs. *Software—Practice and Experience*, IX(4):255-265, April, 1979.
4. Don Libes. expect: Curing those uncontrollable fits of interaction. In *Proceedings of the Summer 1990 USENIX conference*, Anaheim, Ca., June 1990.
5. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Mass., 1990.
6. J. Rees and W. Clinger (editors). The revised^4 report on the algorithmic language Scheme. *Lisp Pointers* IV(3):1-55, July–September 1991.
7. Olin Shivers. A Scheme shell. To appear in the *Journal of Lisp and Symbolic Computation*.
8. Olin Shivers. *The scsh manual.* November 1995, scsh release 0.4. MIT Laboratory for Computer Science. (Also available at URL `http://swissnet.ai.mit.edu/scsh/`.)
9. Guy L. Steele Jr. *Common Lisp: The Language.* Digital Press, Maynard, Mass., second edition 1990.