JFP **14** (5): 475–488, September 2004. © 2004 Cambridge University Press
DOI: 10.1017/S0956796804005052 Printed in the United Kingdom

EDUCATIONAL PEARL

Computer literacy via Scheme and web programming

TIMOTHY J. HICKEY

Department of Computer Science, Brandeis University, Waltham, MA 02454, USA
(*e-mail:* tim@cs.brandeis.edu)

Abstract

We describe an approach to introducing non-science majors to programming and computation in part by teaching them applets, servlets, and groupware applications. The course uses a dialect of didactic Scheme that is implemented in, and tightly integrated with, Java. The declarative nature of our approach allows non-science majors with no programming background to develop surprisingly complex web applications in about half a semester. This level of programming provides a context for a deeper understanding of computation than is usually feasible in a Computer Literacy course.

1 Introduction

Computer literacy courses offer a shining opportunity for functional programming and Scheme in particular to move into the mainstream of undergraduate education. The mandate for Computer Literacy classes requires that they teach useful skills and that they be accessible to all students, no matter what their background or future field of study. The challenge is to avoid falling into the trap of teaching students what they are going to learn on their own anyway, that is, how to use commercial applications such as word processors, email, spreadsheets, webpage builders, photo and movie toolkits, etc. Our thesis, which we propose and support in this article, is that Web programming is a more appropriate subject for Computer Literacy courses and that Scheme provides an ideal environment for teaching Web programming to this population.

Over the past six years we have taught the "Introduction to Computers" course at Brandeis University using such a Scheme-based Web programming approach. This article distills the content of that course into a few simple examples that show the elegance of functional programming for teaching this group of students. We conclude with a discussion of what is required to adapt such a class in other contexts.

1.1 Curricular overview

The course is based around teaching little languages. The first two languages are HTML and CSS. The HTML language is a page layout language that has a simple

formal syntax (tags and elements) and a small set of primitives (about 30 common tags). HTML provides an opportunity for students to learn about nested expressions. The CSS language is a little more complex that HTML as it offers a means of abstraction (giving names to styles) and also introduces notions of inheritance.

Once the students are familiar with creating web pages using HTML and CSS, we gradually introduce Scheme programmming in the guise of dynamic/interactive web pages. To make this transition as comfortable as possible, we have made a simple syntactic extension to Scheme (quasi-strings), which allow complex strings to be constructed in a natural hierarchical fashion. Using this new syntax, we first show how to embed simple expressions in HTML to add dynamic content to a page (e.g. the date or the client's IP address), then we introduce a macro to easily access the HTML parameters and move on to simple calculator servlets. It is then a small step to writing servlets that check passwords and IP-address prefixes before showing their contents. Side-effects are introduced using a `send-mail` procedure that sends mail within the University. Finally, we introduce procedural abstraction as a kind of "super-tag" and this gives an opportunity to discuss the substitution model. At this point, roughly one month into the course, the students have learned some useful skills and have also developed an understanding of the substitution model.

Lists and map are introduced as a means for easily constructing HTML lists and tables. A deeper appreciation for Scheme lists is provided by introducing the `dbquery` procedure, which submits an SQL request to a student database and returns the result as a list of lists (ready to be turned into an HTML table using the methods they have already learned for creating static tables using lists and map).

In the third part of the course we introduce a GUI library (JLIB) and teach the students how to create applets. The key examples we present here are a simple I/O program, a graphics program, a "robot chat" program that simulates an IM conversation with the user, and an IM chat program that exposes some networking primitives and lets students write their own chat applets.

At this point in the semester, there are still 4–5 weeks left, in which we can cover the traditional topics of PC architecture (bits, bytes, bandwidth, clock speed, von Neuman architecture, peripherals, etc.) and digital logic (CMOS gates and circuits, boolean algebra, adders, bits).

1.2 Why Scheme?

It is obvious to anyone who has taught this level of students that it would be difficult, if not impossible, to cover this same material in a one semester general audience course using Java, C++, or any of the other common imperative languages. What is it about Scheme that enables non-science majors to learn how to write sophisticated web applications in a large lecture style intro course? After presenting the details of several servlets and applets in the next section, we argue that this is due to the simplicity of Scheme's syntax and semantics and in particular to the declarative nature of the language – the substitution model is much easier to learn than the underlying machine model of Java or C.

In the rest of this Pearl, we first present the programming pearls that make up the main content of the course, then we take up the question of the level of support that is needed to successfully teach such a course.

2 Scheme servlets and quasi-strings

We provide students with access to a Scheme server so that they can write and deploy servlets. When they put files into the server directory with the extension `.servlet`, those files are interpreted as Scheme programs that return a webpage after being evaluated by the server. Thus, if the file `time.servlet` contains the expression

```
(string-append
     "<html>\n<head><title>Date/Time</title></head>"
     "<body bgcolor='green'>"
     "Current local time is "
     (java.util.Date.)
     "</html>")
```

people get a webpage with the current local time, when people visit `time.servlet`.

After working with this model for a while, we found that the need to combine Scheme and text resulted in programs containing large numbers of string-append's and quoted strings (with many quoted quotes). In response to this somewhat confusing syntax, we introduced a slight syntactic extension to Scheme.[1] It allows curly braces {} to be used in place of double quotes for strings. Moreover, inside a {} string, any Scheme expressions appearing within square brackets [], are evaluated and spliced into the string. These two devices make use of the unassigned outfix operators [] and {}, and allow for a more concise method for constructing strings in Scheme. We call this quasi-string notation.[2]

For example, using quasi-string notation we can write

```
{<html>
  <head><title>Date/Time</title></head>
  <body bgcolor=green>
     Current local time is
     [(java.util.Date.)]
  </body>
</html>}
```

instead of the above Scheme expression. This quasi-string notation is much more accssible for novice students than plain Scheme.

The `java.util.Date.` procedure is a library procedure that returns the date and time. In the first lecture on servlets, we show how to include this and other forms of dynamic data into a webpage, e.g., the IP address or domain name of the client,

[1] `http://jscheme.sf.net`

[2] The quasi-string notation is a syntactic variant on Bruce R Lewis' Beautiful Report Language (BRL) Syntax. Our approach is based on the quasiquote/unquote approach for constructing lists in Scheme.

https://doi.org/10.1017/S0956796804005052 Published online by Cambridge University Press

```
;; init-mylib.servlet
(begin
  (define (<captioned-image> C I)
   {<table border=5>
       <tr><td>
         <img src="[I]" alt="[C]">
      </td></tr>
       <tr><td>[C]
     </td></tr> </table>})

  (define (<tims-page> Title CSS Body)
    {<html>
        <head><title> [Title]</title>
         <style type="text/css" media="screen">
           <!-- [CSS] --></style></head>
      <body> [Body]</body>
    </html>})
)
```

Fig. 1. An HTML abstraction library.

the type of operating system the client is using, and other HTML header data. The students find this exciting and easy. The small syntactic changes to their HTML code provides a gentle introduction to servlets that, as we show next, leads naturally to abstraction, conditional execution, and expression evaluation.

2.1 Introducing abstraction

Once the idea of dynamic content is clearly established, we move on to abstraction and show how to use the "define" form to create "Scheme tags." This simple and powerful idea leads naturally to a discussion of the substitution model of Scheme evaluation, and allows students to start writing and sharing new HTML tag libraries, written in Scheme.

For example, Figure 1 shows a library that includes a personal webpage procedure and a captioned image procedure. This "super-tag" library is implemented as a servlet that adds the specified bindings to the environment when it is visited by the user. In practice, the HTML/CSS code for the students' personal webpage is much larger, as it typically contains HTML to generate a fancy header, footer, and a sidebar. Likewise, their code for a captioned image would often contain much more HTML.

A use of this simple library is shown in Figure 2. The benefits of this sort of abstraction become even greater when the abstractions start using sophisticated inline-CSS style attributes to create highly stylized HTML components.

This technique for abstracting HTML is well-known in Lisp/Scheme Web programming (e.g. LAML (Normark, 1999), BRL[3]) and is similar to the approach used

[3] http://brl.sourceforge.net

https://doi.org/10.1017/S0956796804005052 Published online by Cambridge University Press

```
(<tims-page>

  "Demo page"

  "body {background:black;color:white}
   h1{border: thick solid red}"

  {<h1>My Pets</h1>
    [(<captioned-image> "Snappy and Pepper" "cats.jpg")]
   <br>
    [(<captioned-image> "Missy" "missy.jpg")]  })
```

Fig. 2. Using HTML abstraction libraries.

by Curl, the Server-Side Includes in JSP[4] and the publishing model of the Zope environment.[5]

2.2 Introducing user interaction

The next pedagogical step is to introduce I/O with HTML forms. Forms are used to send data from the user to the servlet. The data is then used to generating HTML pages that are returned to the user.

To simplify the computational model for novice students, we provide easy access to form parameters using the `(servlet (p1 p2 ...) ....)` macro, which binds the variables `p1,...` to the strings associated with the form parameters of the same names. This allows students to easily write servlets that process form data from webpages. This also proves to be a good time to introduce the notion of conditional execution (using if, cond, and case):

For example, after a week of HTML instruction we have found that beginning students are easily able to create HTML forms and it is then a small step to the servlet in Figure 3, which either generates a form or generates a response to the form, depending on whether the form parameter has been given a value by the browser.

2.3 Expression evaluation

The next step is to introduce numerical computation into servlets. This requires three new ideas:

- data types (converting strings into numbers using `Double.`)
- evaluation of arithmetic s-expressions
- introduction of intermediate variables using `let*`

An example, of the type of program the students are able to construct at this level is shown in Figure 4.

Admittedly, this is a big step. To help students we introduce and review the substitution model to explain how expression evaluation proceeds.

[4] http://java.sun.com/products/jsp
[5] http://www.zope.org

https://doi.org/10.1017/S0956796804005052 Published online by Cambridge University Press

```
(servlet (password bg fg words)
 (case password
   ; first visit to page, make form
   ((#null)  (<tims-page> {color viewer form}
                          ""
                          {<h1>pw-protected color viewer</h1>
                           <form method=post action="demo1.servlet">
                            pw <input type=text name="pw"><p>
                            bg <input type=text name="bg"><p>
                            fg <input type=text name="fg"><p>
                            text<textarea name="words">
                            Enter text to view here</textarea>
                            <input type=submit>
                          </form>}))

  ;; correct pw, process data
  (("cool!") (<tims-page> "color viewer"
                          {body \{background:[bg];color:[fg]\}}
                          words))

  ;; incorrect password, complain!
 (else  (<tims-page> "ERROR"
                     " body {color:red;background:black}"
                     {<h1>WRONG PASSWORD<h1>
                       Go back and try again!}))))
```

Fig. 3. A password protected page.

2.4 *Data structures and* `map`

Students naturally want to handle list-style data (e.g., multiple checkboxes in form data). This leads into a description of list functions and also to table abstractions. We find it useful to introduce map before car, cdr, cons, because it provides a powerful and intuitively clear operation and does not require an understanding of recursion. Moreover, as the examples in Figure 5 below illustrate, the `map` procedure gives the students most of what they need to handle lists of data values. Note that the quasi-string syntax splices lists and hence prints out a list of HTML elements without the parentheses.

2.5 *Email and database interaction*

Responding to student requests, we have also added a few additional primitives for interacting with databases. This provides an opportunity to provide a brief introduction to another small language (SQL) and introduces a simple mechanism for persistence into the Web programing paradigm. Currently, we provide a brief introduction to SQL and provide them with a few stock expressions (create a table, select rows, insert into a table, update an entry, delete an entry). With this brief introduction, students easily add logs and counters to their webpages as shown in Figure 6. This example also shows the `send-mail` procedure, which allows the

https://doi.org/10.1017/S0956796804005052 Published online by Cambridge University Press

```
(servlet (inches pounds)

 (define (generate-form)
    (<tims-page> {color viewer form}
                 ""
                 {<h1>BMI Calculator</h1>
                  <form method=post action="bmi.servlet">
                   height:
                   <input type=text name="inches"> inches<br>
                   weight:
                   <input type=text name="pounds">pounds<br>
                   <input type=submit>
                 </form>}))

 (define (create-BMI-page inches pounds bmi)
   (<tims-page>   "Body Mass Index"
                  " body {background:rgb(255,235,215)}"
                  {<h1>Body Mass Index<h1>
                    With a height of [inches] inches<br/> and
                    a weight of [pounds] pounds,<br/> your
                    Body Mass Index is [bmi] <br>
                    Note: a BMI over 25 indicates you may be
                    overweight,<br/>while a BMI over 30 indicates<br/>
                    that your weight may cause significant health
                    problems!}))


 (if (equal? inches #null)
     (generate-form)
     (let*( (h (Double. inches))
            (w (Double. pounds))
            (h-in-m (* h 0.0254))
            (w-in-kg (/ w 2.2))
            (bmi  (/ w-in-kg (* h-in-m h-in-m))))
       (create-BMI-page inches pounds bmi)))
)
```

Fig. 4. A sample quasi-string servlet.

students to specify the “from”, “to”, “subject” fields of an email message and to give a quasi-string for the body.

3 Scheme applets

After spending about three weeks studying servlets, we turn to client-side computing. The tomcat server has been configured so that any Scheme program that ends with “.applet” is transformed into a Scheme applet and runs in the client’s browser. Likewise, Scheme programs that end in “snlp” are converted into Java Network

https://doi.org/10.1017/S0956796804005052 Published online by Cambridge University Press

```
(define (li x) {<li>[x]</li>})
(define (ul L) {<ul>[(map li L)]</ul>})
(define (ol L) {<ol>[(map li L)]</ol>})
(define (td X) {<td>[X]</td>})
(define (tr Ts) {<tr> [(map td Ts)] </tr>})
(define (table Rs) {<table> [(map tr Rs)] </table>})
```

Fig. 5. Generating lists and tables.

```
(servlet()
   (dbquery "insert into classlog values('tim',[(.getRemoteHost request)])")
   (dbquery "update classcounter set c = c+1 where tag='tim'")
   (send-mail request
      "tjhickey@brandeis" "nobody@brandeis"
      "counter" {You got a hit! [(Date.)]})

  (<tims-page>
       "log/counter demo"
       ""
       { This list has been visited by
         [(table ""
            (dbquery {select * from classlog where name="tim" limit 20}))]
         and you are visitor number
         [(first
           (second
             (dbquery {select c from classcounter where tag="tim"})))]
       }))
```

Fig. 6. Logs and Counters in test.servlet.

Protocol format which is automatically downloaded and run in the Java Web Start plugin.[6]

To help students deal with Graphical User Interfaces, we have written a library, JLIB, that provides declarative access to the Java Swing toolkit. An example of a simple Scheme program using this library is shown in Figure 7. The first five lines of the program are strings that provide documentation about this program, which is required by the Java Network Launching Protocol (JNLP).

3.1 JLIB

The JLIB model is based on four fundamental concepts:

- COMPONENTS – there are a small number of ways to construct basic GUI components (buttons, windows, . . .).
- LAYOUTS – there are a small number of ways to layout components (row, col, table, grid, . . .).
- ACTIONS – there is one simple mechanisms for associating an action to a component.

[6] http://java.sun.com/products/javawebstart

https://doi.org/10.1017/S0956796804005052 Published online by Cambridge University Press

```
"John Doe"
"http://www.johndoe.com"
"years->secs calculator"
"Convert age in years to age in seconds"
"http://www.johndoe.com/jd.gif"

(jlib.JLIB.load)
(define years-tf (textfield "" 20))
(define seconds-tf (textfield "" 20))
(define w
 (window "years->secs"
  (menubar
   (menu "File"
    (menuitem "quit"
     (action (lambda(e) (.hide w))))))
  (border
   (north (label "Years->Seconds Calculator"
                 (HelveticaBold 60)))
   (center
    (table 3 2
     (label "Years:")
     years-tf

     (label "Seconds:")
     seconds-tf

     (button "Compute" (action(lambda(e)
       (let*
          ((y (readexpr years-tf))
           (s  (* 365.25 24 60 60 y)))
         (writeexpr seconds-tf s)))))))))
(.pack w)
(.show w)
```

Fig. 7. A sample SNLP program.

- PROPERTIES – there are easy ways for setting the font and color of components.

Another key idea is that operations on all components should be as uniform as possible. For example, there are procedures "readstring" and "writestring" which allow one to read a string from a component, and write a string onto a component. Thus "writestring" can change the string on a label, a button, a textfield, a textarea. It can also change the title of a window or add an item to a choice component. Likewise, "readstring" returns the label of a button, the text in a textarea or textfield, the text of the currently selected item in a choice, the title of a window, and the text of a label.

JLIB provides procedures for each of the main GUI widgets (window, button, menubar, label) and it also provides procedures for specifying layouts (e.g. border, center, row, col, table). The order of the arguments is not important, the JLIB

https://doi.org/10.1017/S0956796804005052 Published online by Cambridge University Press

```
(jlib.JLIB.load)
(import "java.awt.Graphics")

(define c (canvas 400 400))
(define w (window "graphics1"
 (border
    (center c)
   (south
      (button "draw"
        (action (lambda(e)
         (run-it drawballs))))))))

(define (run-it F) (.start (Thread. F)))


(define (drawballs) (drawball 200))

(define (drawball N)
  (define g (.bufferg$ c)) ;get graphics object
  (.Graphics.setColor g blue)
  (.Graphics.fillRect g 0 0 1000 1000) ;; clear background
  (.Graphics.setColor g red)
  (.Graphics.fillOval g N N 100 100)    ;draw red disk
  (.repaint c) ; copy buffer to screen
  (Thread.sleep 100L) ;; pause 0.1 sec
  (if (> N 0) (drawball (- N 1))) ;; loop
  )

(.resize w 400 400)
(.show w)
```

Fig. 8. Graphics programming.

procedure examines the type of the argument to determine how it should be handled. For example, string arguments are "written" onto the widget, integers specify the "size" of the textfield components, fonts, background colors, and actions likewise modify the component in natural ways. A closure `F` appearing in the argument list of a widget `W` is applied to the widget with the side effect `(F W)` usually modifying the widget in some way.

3.2 Graphics and animation

We provide a simple graphics library providing access to a canvas with an offscreen buffer. The drawing primitives are the Java primitives from the java.awt.Graphics class. The "canvas" procedure is a JLIB procedure that creates a canvas with an offscreen buffer accessed by `(.bufferg$ c)` and which can be drawn to the screen using `(.repaint c)`. The program in Figure 8 shows a simple example drawing a red ball moving across a blue background.

https://doi.org/10.1017/S0956796804005052 Published online by Cambridge University Press

The `run-it` procedure is used when the students write animations. They seem to understand the notion of multi-threaded programming in the context of having several animations each running in its own thread.[7]

4 Networking abstractions

After spending two weeks mastering the JLIB library we introduce network programming. The library uses a simple model where applets communicate by sending Scheme terms to each other through a `group-server`. Since applets are only able to open sockets on their host server, we must run the `group-server` on the same machine that manages the students' applets. The students connect to this `group-server` using the `make-group-client` procedure:

```
(define S
  (make-group-client Name Group Host Port))
```

This creates a closure, `S`, that can be used to communicate with the group-server. To send the Scheme terms `key b c ...` to the server, one evaluates the expression

```
(S 'send key b c ...)
```

The first term, `key`, is used as a filter. Indeed, the group-server bounces back every message it receives to all the members of the group. A member can specify how to handle a message using the `add-listener` method

```
(S 'add-listener key
    (lambda (key . restargs ) ...))
```

This method indicates that the indicated procedure should be called on each message that arrives from the server with the specified key.

This model builds on the student's experience with callbacks in GUI's and with reading/writing on GUI components. The analogy is that "send" is like writing to a component and "add-listener" is like adding an action.

An example of the kind of applet that is explained in class is the chat applet shown in Figure 9. In the most recent semester we did not require students to write an applet using networked communication, but several students chose to write such applets for their final project. The best example was a pictionary applet. Any number of students could join the game, and they communicated using a shared whiteboard as well as private and group chats. This program was written by a student with no previous programming experience and made use of almost all of the examples we had given previously in the course.

In the coming year we plan on introducing networked communication using the notion of groupware components. These are textareas and canvases that are shared among several users on the network. This approach may provide an even simpler model of network programming that builds more directly on their understanding of GUI programs.

[7] We also have a version of run-it that looks for errors and reports them in a debugging window.

https://doi.org/10.1017/S0956796804005052 Published online by Cambridge University Press

```
(jlib.JLIB.load)
(jlib.Networking.load)
(define (chatwin User Group Host Port)
   (define S (make-group-client User Group Host Port))
   (define chataction
     (action (lambda(e)
      (S 'send "chat"
         {[User]:[(readstring chatline-TF)]\n})
      (writestring chatline-TF "")
     )))

   (define chatline-TF (textfield "" 50 chataction))

   (define chatarea-TA (textarea 20 50))

   (define w
     (window "test"
      (col
        (button "quit" (action (lambda (e)
              (S 'logout) (.hide w))))
        chatarea-TA
        chatline-TF)))
   (.pack w) (.show w)

   (S 'add-listener "chat"
     (lambda R (appendlnexpr chatarea-TA R)))
   w)
(define (rand N)
       (Math.round (* N (Math.random))))
(chatwin
      {user-[(rand 1000)]}
      "chat"
      (.getHost (.getDocumentBase thisApplet))
      23456)
```

Fig. 9. A multi-room chat program.

5 Experience

This curriculum has been developed and taught over the past six years to an audience of primarily non-science undergraduates. The most recent course of 120 students had about 50% freshman, 25% sophmores, 12% juniors, and 12% seniors. They were about equally distributed between four groups: science majors, social science majors, arts and humanities majors, and undecided students. About two thirds of the students took the course out of interest while the remainder took it to satisfy a requirement.

Over a thousand students have completed this course, which has regularly been among the most popular courses each year. The curriculum has also been used in the core computer science course in the Transitional Year Program at Brandeis. This program recruits bright high school graduates from under-resourced schools and provides them with a year of college-level classes in preparation for applying to elite

https://doi.org/10.1017/S0956796804005052 Published online by Cambridge University Press

colleges. In the TYP the course is taught as a "computer lab" with 10–20 students. The curriculum works well in both large classes and small classes, but the former requires that lab assistants be available to help students with their programming assignments.

In course evaluation surveys the students are generally positive about the course and are often surprised by how much they have learned. There are ocassional students that complain about Scheme, but it is usually because they would rather be using Python or Perl or some other scripting language.

My colleagues are uniformly pleased with the course, and are amazed that non-majors are able to learn so much computer science. It is generally felt that the course is a "real" Computer Science course and not a vocational service course.

The course has not yet been replicated at other schools. The curriculum has been used by another instructor teaching a small format version of the course in the Transitional Year Program. He covered somewhat fewer topics, but went into greater depth.

Overall the most surprising aspect of the course is that non-science majors are able to learn how to write HTML, CSS, servlets, applets, and applications in Scheme all within an eight week unit of a 13 week semester. The primary reasons for the success of this approach seem to be:

- we eliminate the problem of learning complicated syntax (as one must only match parentheses and quotes with support from the Scheme IDEs)
- the substitution model for Scheme semantics is much simpler to learn than the abstract machine models of Java, C, or other imperative languages
- the use of quasi-strings eases the transition from HTML to Scheme
- by carefully crafting a simple web server environment, the students are able to deploy servlets and applets, simply by uploading them (with the appropriate file suffix) to the server.

Although we use JScheme (Anderson *et al.*, 2000; Anderson & Hickey, 1999; Hickey *et al.*, 1998) in this course,[8] it is clear that Web programming ideas like those taught in our course could be taught using other dialects of Scheme and other functional languages. Dr. Scheme would be a natural environment (Findler *et al.*, 1997), especially since it already provides access to a webserver and graphical user interface libraries. In higher level courses, the libraries for Web and applet programming can easily be incorporated into the text of a homework assignment (e.g., in an SICP-based course (Abelson and Sussman, 1984)).

The key idea behind the Web programming approach to Computer Literacy pedagogy is to use the students interest and excitement about building web artifacts to provide an avenue for teaching them about programming concepts.

[8] The courseware and lecture notes we have developed for this course are available from the Jscheme website.[9]

https://doi.org/10.1017/S0956796804005052 Published online by Cambridge University Press

Acknowledgements

I would like to acknowledge the support of the steadily growing JScheme community, including my co-developers Ken Anderson and Peter Norvig, and my students Hao Xu, Lei Wang who helped develop the very first version in 1997. Finally, I'd like to thank the 1000+ students who have explored the possibilities of Scheme applets and servlets with me in various introductory classes over the past five years. This work was supported by the National Science Foundation under Grant No. EIA-0082393.

References

Abelson, H. and Sussman, J. (1984) *Structure and Interpretation of Computer Programs*. MIT Press.

Anderson, K. A., Hickey, T. J. and Norvig, P. (2000) Silk: A Playful Combination of Scheme and Java. *Proceedings of the Workshop on Scheme and Functional Programming*, Rice University, Computer Science Department, Technical Report 00-368.

Anderson, K. and Hickey, T. J. (1999) Reflecting Java into Scheme. *Proceedings of Reflection 99: Lecture Notes in Computer Science 1616*. Springer-Verlag.

Clinger, W. and Rees, J. (eds.) (1991) The revised[4] report on the algorithmic language Scheme. *ACM Lisp Pointers*, **4**(3), 1–55.

Findler, R. B., Flanagan, C., Flatt, M., Krishnamurthi, S. and Felleisen, M. (1997) DrScheme: a pedagogic programming environment for Scheme. *Proceedings Symposium on Programming Languages: Implementations, Logics, and Programs*.

Hickey, T. J., Norvig, P. and Anderson, K. (1998) LISP – a Language for Internet Scripting and Programming, *LUGM'98: Proceedings of Lisp in the Mainstream*, Berkeley, CA.

Normark, K. (1999) Programming World Wide Web pages in Scheme. *Sigplan Notices*, **34**(12).