Incorporating Scheme-based Web Programming in Computer Literacy Courses

Timothy J. Hickey *
Department of Computer Science
Brandeis University
Waltham, MA 02254, USA

Abstract

We describe an approach to introducing non-science majors to computers and computation in part by teaching them to write applets, servlets, and groupware applications using a dialect of Scheme implemented in 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. The course does not require the students to download any software as all programming can be done with Scheme applets. The instructor however must provide a Scheme server which will run the students' servlets.

1 Introduction

There are two general approaches to teaching a Computer Literacy class. The most common approach is a broad overview of Computer Science including hardware, software, history, ethics, and an exposure to industry standard office and internet software. On the other end of the spectrum is the class that focuses on programming in some particular general purpose language, (e.g. Javascript [12], Scheme[5], MiniJava[11]).

The primary disadvantage of the breadth-first approach is that it tends to offer a superficial view of computing. The depth-first programming approach on the other hand often requires a substantial effort just to learn the syntax of the language and the semantics of the underlying abstract model of computation, leaving little time to look at other aspects of computing such as internet technology or computer architecture.

Several authors have recently proposed merging these two approaches by using a simpler programming language (e.g. Scheme[5], [6], [7]) or by using an internet-based language (e.g. Javascript[12], MiniJava[11]).

In this paper we describe a five year experiment in combining these two approaches in a Computer Literacy course at Brandeis University (CS2a: Introduction to Computers). We deviate from many Computer Literacy courses in that we spend very little time discussing the standard application programs (e.g. word processors, spreadsheets, email, instant messaging, file sharing, image processing etc.) It has been our experience that students are able to learn how to use most of these programs on their own and that use of these applications does not generally require a deep understanding of computation. In a phrase, we don't teach them what they are going to learn by themselves anyway.

The CS2a:Introduction to Computers course teaches programming concepts and uses a small (but powerful) subset of Jscheme[2] – a Java-based dialect of Scheme. The tight integration of Java with Jscheme allows it to be easily embedded in Java programs and hence makes it easy for students to implement servlets, applets, and other web-deliverable applications. Jscheme is an implementation of Scheme in Java (meeting almost all of the requirements of the R4RS [4] Scheme standard). It also includes two simple syntactic extensions:

- **javadot notation**: this provides full access to Java classes, methods, and fields
- **quasi-string notation**: this simplifies the process of generating HTML.

The javadot notation provides a transparent access to Java and the quasi-string notation provides a gentle

*This work was supported by the National Science Foundation under Grant No. EIA-0082393.

Third Workshop on Scheme and Functional Programming. October 3, 2002, Pittsburgh, Pennsylvania, USA.

path from HTML to Scheme for novices. It also provides a convenient syntax for generating complex strings of other sorts (such as SQL queries). These two extensions will be discussed at length below.

Jscheme can be accessed as an interpreter applet (running on all Java-enabled browsers) or as a Java Network Launching Protocol (JNLP) application. Both of these provide one click access to the Jscheme IDE from standard browsers. It can also be downloaded as a jar file and run from the command line as a standard read-eval-print-loop program.

Jscheme has been built into a Jakarta Tomcat webserver as a webapp which allows students to write servlets and JNLP applications directly in Jscheme. This webserver typically runs on the instructor's machine, but students can easily download and install the server on their home/dorm PCs as well.

In the sequel, we explain, in detail, how Jscheme can be used to teach non-science majors in a large lecture class how to build servlets and applets in a six week section of a Computer Literacy course. The approach described here is very similar to the approach used in the Autumn 2001, "Introduction to Computers" course at Brandeis University, but it reflects changes that will be incorporated in the next year's version of the course. The course and the underlying language have been evolving steadily over the past five years and will likely continue to do so.

This approach to teaching Computer Literacy is feasible because of the declarative style of programming that is possible in Scheme, together with the extremely simple syntax and semantics of Scheme.

We posit that this web-programming based approach would work with other declarative languages (e.g. Haskell or Prolog), but would be infeasible with imperative languages such as Java or Perl. Scheme however is ideally suited to this application because of the relative simplicity of its syntax and semantics, both of which can be stumbling blocks for novice programmers.

Although the particular languages and techniques that we use may not be the best match at other institutions, we feel that the general approach could be easily replicated using other languages provided care is taken to make the syntax and semantics that must be learned as simple as possible.

2 Related Work

The need for a simple, but powerful, language for teaching introductory CS courses has been discussed recently by Roberts [11] who argues for a new language, Minijava, that provides both a simpler computing model (e.g. no inner classes, use of wrapper class for all scalar values, optional exception throwing) and a simpler runtime environment (e.g. a read-eval-print loop is provided).

Jscheme can be viewed as an even more radical simplification of Java in that it replaces the syntax of Java with the much simpler syntax of Scheme while maintaining access to all of the classes and objects of Java.

Another recent approach for introductory courses is to use Javascript to both teach programming concepts and to provide a vehicle for discussing other aspects of computing such as the internet and web technology. For example, David Reed proposes teaching a course [12] in which about 15% of class time is devoted to HTML, 50% to Javascript, and 35% to other topics in computer science. Our approach follows a similar breakdown but also allows the students to build servlets, applets, and GUI-based applications.

A third related approach is to teach Scheme directly as a first course. The MIT approach, pioneered by Abelson and Sussman [1], is not suitable for non-science majors as it requires a mathematically sophisticated audience. The approach being developed by the PLT group [5], [6],[7], on the other hand, provides a rigorous introduction to Scheme programming but is designed to be accessible to students from all disciplines.

In our aproach, we provide an introduction to only a subset of the language (for example, introducing lists only toward the end). We start by introducing some high-level declarative libraries for teaching an event-driven model of GUI construction. The Scheme section of the course requires only about 6 weeks. This leaves half of the course for standard Computer Literacy topics.

3 Goals, Syllabus, and Rationale

Our main goal in teaching a Computer Literacy course is to help the students gain a broad understanding of digital computation. It is our feeling that Computer Literacy courses are most effective if they focus on the fundamental mechanisms of computing at all levels and if they ground this theoretical material by requiring the students to build programs using these fundamental concepts.

The syllabus covers the mechanisms underlying CMOS gates and VLSI, the structure and interpretation of assembly language, the design of simple GUI-based applications, the mechanisms underlying servlets (including counters, logs, and auto-generated email), the basic design and structure of the internet, and the limits of computers (e.g. the Halting problem and the Turing test).

We test their understanding of this material using weekly quizzes, biweekly homework assignments, and a final exam in which they must write and/or trace programs at these various levels (from semiconductors to servlets). Before delving into a detailed description of the curriculum we first explain what we do not cover and provide some justification for these choices.

This course also does not delve very deep into the soft aspects of Computing. These topics are covered in a companion course (CS33b: Internet and Society), which is focused primarily on the social, ethical, legal, economic, political and aesthetic aspects of computers. It is our opinion that these issues are best taught in an interdisciplinary context. Indeed, the CS33b course is currently taught by a dozen instructors from half a dozen different departments.

The course does not teach algorithms and data structures. Although the students do learn to trace through the execution of fast-exponential procedures, gcd calculators, and the "map" function, we do not teach them to use computers for problem solving. Thus we do not ask them to write sorting procedures or programs to find average grade scores, etc.

We **do** teach "reactive" programming in this course, i.e. programs that interact with the user (through GUIs or HTML forms) and use the user-supplied information to generate responses and perform simple actions (logging, sending email, updating counters, performing simple calculations and tests). We also teach the students to understand how to trace recursive programs which is a far easier task than learning how to write recursive programs. More precisely,the students are required to be able to write applets and servlets in three languages (HTML, CSS, Scheme) and to trace programs in two additional "languages" (pcode assembly language, and CMOS circuit diagrams).

The goal in teaching them to write "reactive" programs and to trace recursive programs is to help them understand the deeper issues of computation more clearly. For instance, one of the applet programs we present is a simple "Psychiatrist" simulator which they are encouraged to modify. This provides a context for a deeper discussion of artificial intelligence, ethics, and the Turing problem. For another example, when we discuss the substitution model of Scheme the students are required to trace recursive programs with function parameters (e.g. map). This paves the way for a discussion of the Halting problem. We consider the consequences of extending the Scheme language by adding a primitive procedure `(halts? F X)` which returns true if `(F X)` eventually returns an answer and false if it throws an exception or does not return. In particular, we look at the following program:

```
(define (skeptic Q)
      (if (halts? Q Q) (skeptic Q) 'ha))
(skeptic skeptic)
```

The trace of `(skeptic skeptic)` yields the expected contradiction which then leads to a discussion of the limits of computation. It is true that the `skeptic` example only makes sense in the context of a Scheme which provides source code access to all procedures and closures, but the impossibility of adding a recursive "halts?" procedure still illustrates well the limits of computation. We usually couple this lecture with a classroom exercise in which the students must prove that the instructor can not tell the future. The proof consists of asking the instructor to predict the student's behavior using the same strategy as the "skeptic" procedure.

A rough outline of the syllabus, which shows the context of the web-programming part of the course is shown below.

- 1 week **HTTP** and the structure of the Internet: IP addresses, ports, sockets, services, routers, gateways. Use of telnet, dig, traceroute, ping, portscan to illustrate these issues.
- 2 weeks **HTML/CSS** – the thirty non-style HTML tags and 10 basic CSS properties. Copyright issues.
- 3 weeks **Scheme Servlets** – quasi-string notation, abstraction, conditional execution, lists, file I/O, email, database access. Security, privacy, cookies, ethics.
- 3 weeks **Scheme Applets/Groupware** – GUI components, layout, callbacks, animation, networking primitives, groupware components. Doctor applet, Turing Test. Halting problem. Substitution model. Software licenses.
- 1 week **Assembly Language/Pcode** - von Neumann architecture, memory-mapped peripherals, memory, speed, bandwidth, cacheing, super-scalar architectures. Operating Systems, file systems, time sharing, ...
- 1 week **CMOS/Logic Circuits** - semiconductors (P/N-type), gates, circuits, adders, latches and bits.

Observe that the course contains a signficant amount of non-Scheme material that would be found in most typical Computer Literacy courses (such as copyright issues and ethical questions dealing with servers), but with this programming-based approach these issues are more meaningful as the students are able to write servers that create logs and must deal with the resulting ethical questions.

4 Courseware

The main language used in the course is Jscheme[1] [2, 3, 8], an open source implementation of Scheme in Java.

[1] http://jscheme.sourceforge.net

```
SYNTACTIC CONSTRUCT          JAVA MEMBER        EXAMPLE
"." at the end               constructor        (Font. NAME STYLE SIZE)
"." at the beginning         instance method    (.setFont COMP FONT)
"." at beginning, "$" at end instance field     (.first$ '(1 2))
"." only in the middle       static method      (Math.round 123.456)
".class" suffix              Java class         Font.class
"$" at end, no "." at beg.   static field       Font.BOLD$
"$" in the middle            inner class        java.awt.geom.Point2D$Double.class
"$" at the beginning         packageless class  $ParseDemo.class
"#" at the end               access private data Symbol.#
```

Figure 1: Java reflectors in Jscheme

It is almost completely compliant with the R4RS standard [2] [4] and also provides full access to Java using the Java Reflector syntax shown in Figure 1. Jscheme also provides full access to Java thread and exception handling. The following example illustrates the ease with which one can access Java libraries in Jscheme. It implements a simple multi-threaded "echo service" on a specified port and catches/reports any errors that may arise in each thread:

```
(define (echoserver N)
 (let ((SS (java.net.ServerSocket. N)))
  (let loop ()
   (let ((S (.accept SS)))
    (.start
     (java.lang.Thread.
      (lambda()
       (tryCatch
        (let*
          ((in (java.io.BufferedReader.
                (java.io.InputStreamReader.
                 (.getInputStream S))))
           (out (java.io.PrintStream.
                 (.getOutputStream S))))
         (.println out (.readLine in))
         (.close S))
        (lambda(e)
          (.println java.lang.System.out$
                    (.toString e))))))))
   (loop))))
```

The course uses a small but powerful subset of Scheme and also relies on only a few selected Java reflectors and a small GUI-building library. For control flow and abstraction it uses `define, set!, lambda, if`,`cond`, `case, let*`. For primitives, it uses arithmetic operators and comparisons, a simple GUI-building library (providing declarative access to Swing components, events, and layout managers).

[2] strings are not mutable, and call/cc is only implemented for try/catch like applications

4.1 Scheme Servlets

Files which appear in the Jscheme webserver student directory with the extension ".servlet" are treated as Jscheme expressions which are evaluated to generate the html to send back to the client. 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 which 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 appended 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[3]

For example, using quasi-string notation we can write

```
(define (my-li NAME IMAGEFILE COST)
 {<div style="background:rgb(0,150,150)">
   <table  width="100%">
     <tr><td>
      <a href="[IMAGEFILE]">
       <img src="[IMAGEFILE]"
         alt="[NAME]" width="150"></a><br>
    </td><td> <h1 style="background:lightgreen;
                        color:black">[NAME]</h1>
    </td><td style="text-align:right">
        Cost: $[COST] </td></tr></table>
 </div>  <br>   <br>   <br>
 })
```

which is equivalent to the following (less elegant) standard Scheme expression. Note in particular the confusion that arises from the need to quote double quotes. In the quasi-string syntax, it is much easier to verify the syntactic correctness of the resulting code.

[3] 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.

```
(define (my-li NAME IMAGEFILE COST)
 (string-append
  "<div style=\"background:rgb(0,150,150)\">
   <table  width=\"100%\">
     <tr><td>
      <a href=\""
 IMAGEFILE
      "\">
       <img src=\""
 IMAGEFILE
       "\"
          alt=\""
 NAME
       "\" width=\"150\"></a><br>
    </td><td> <h1 style=\"background:lightgreen;
                  color:black\">"
 NAME
      "</h1>
    </td><td style=\"text-align:right\">
      Cost: $"
 COST
     " </td></tr></table>
  </div>  <br>   <br>   <br>
 "))
```

The quasi-string notation is similar to the quasiquote syntax used to construct s-expressions in Scheme.

4.1.1 Dynamic content

The first non-trivial examples of servlets that we provide are servlets that include runtime generated data (such as the current date, or information from the HTML headers, like the client operating system). For example, by enclosing their HTML in curly braces, changing the extension from html to servlet, they can add this dynamic content to their page just by including the `[(java.util.Date.)]` expression into their HTML.

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

Evaluating this expression yields

```
<html>
  <head><title>Date/Time</title></head>
  <body>
     Current local time is
     Fri Sep 07 09:33:30 EDT 2001
  </body>
</html>
```

These small syntactic changes provide a gentle introduction to servlets that, as we will show below, leads naturally to abstraction, conditional execution, and expression evaluation.

4.1.2 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 only requires an understanding of the substitution model of scheme evaluation, and yet allows students to start writing and sharing new HTML tag libraries, written in Scheme. For example, Figure 2 shows a typical and simple library that includes a generic webpage procedure and a captioned image procedure.

```
;; loadmylib.servlet
   (define (cimg C I) ;; captioned images
    {<table border=5>
        <tr><td>
          <img src="[I]" alt="[C]">
       </td></tr>
        <tr><td>[C]
      </td></tr> </table>})

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

Figure 2: An HTML abstraction library

An example of the use of this simple library is shown in Figure 3. The benefits of this sort of abstraction become even greater when the abstractions start using sophisticated inline-CSS style attributes to create a highly stylized HTML components.

```
(begin
  (generic-page    "Pets"
    "body {background:black;color:white}
     h1{border: thick solid red}"

    {<h1>Pets</h1>
      [(list
         (cimg "Snappy" "snappy.jpg")
         (cimg "Pepper" "pepper.jpg")
         (cimg "Missy"  "missy.jpg")
         (cimg "Kitty"  "kitty.jpg")
         (cimg "Tarzan" "dog17.jpg"))]
    })
```

Figure 3: Using HTML abstraction libraries

This technique for abstracting HTML is well-known is Lisp/Scheme web programming (e.g. LAML[10], BRL[4]) and is similar to Server-Side Includes in JSP[5] or the publishing model of the Zope environment[6].

4.1.3 Introducing User Interaction

The next pedagogical step is to introduce the notion of using HTML forms to send data from the user to the servlet.

To simplify the computational model for novice students, Jscheme provides 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 one 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):

```
(servlet (password bg fg words)
 (case password
   ((#null)  ; first visit to page, make form
    (generic-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>}))

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

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

Figure 4: A password protected page

For example, after a week of HTML instruction we have found that beginning students easily create HTML forms and it is then a small step to the servlet in Figure 4 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.

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

4.1.4 Expression Evaluation

The next step is to introduce numerical computation into servlets. An example, of the type of program the students are able to construct at this level is shown in Figure 5 below.

```
(servlet (inches pounds)
 (if (equal? inches #null)
   ;; first visit to page, create form
    (generic-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="weight">pounds<br>
      <input type=submit>
    </form>})
  ;; else  compute BMI, display results
   (let*(     (h-in-m (* inches 0.0254))
              (w-in-kg (/ pounds 2.2))
              (bmi  (/ w-in-kg (* h-in-m h-in-m))))
   (generic-page "Body Mass Index"
     " body {background:rgb(255,235,215)}"
     {<h1>Body Mass Index<h1>
      With a height of [inches] inches and
      a weight of [pounds] pounds, your
      Body Mass Index is [bmi] <br>
     Note: a BMI over 25 indicates you may be
     overweight, while a BMI over 30 indicates
     that your weight may cause significant health
     problems!}))))
```

Figure 5: A sample quasi-string servlet

This requires two new ideas:

- evaluation of arithmetic s-expressions[7]
- introduction of intermediate variables using `let*`

This is admittedly a big step. At this point we review the substitution model to explain how expression evaluation proceeds, and we introduce an environment model to explain the semantics of the `let*` expression.

For students to be able to write this type of servlet they need to learn to use prefix Scheme arithmetic expressions and to use the `servlet` and `case` macros.

4.1.5 System Interaction

We have also added a few additional primitives for writing or appending scheme terms to a file, and for reading

[7] The servlet macro automatically converts numerals to Java numbers, thus `pounds` and `inches` are numbers

a file either as a string or as a list of scheme terms. These allow students to easily write logs and counters as in Figure 6. This example also shows the `send-mail` procedure which allows the students to specify the "from", "to", "subject" fields and give a quasi-string for the body.

```
(servlet()
  (let* ((c (read-from-file "counter" 0))
         (d (list c (Date.)
                  (.getRemoteHost request))))
    (write-to-file "counter" (+ 1 c))
    (append-to-file "log" d)
    (send-mail
      "tjhickey@brandeis" "nobody@brandeis"
      "counter" {You got a hit: [d]!})
  {<html><body>
    This list has been visited by <xmp>
   [(read-string-from-file "log" "")</xmp>
  and you are visitor number [(+ 1 c)]
```

Figure 6: Logs and Counters in test.servlet

In order to simplify the problem of associating log and counter files to servlets, these primitives read and write from files whose prefix is the name of the servlet. Thus, for the log and counters example, the "log" file would be named "test.servlet_log" and the counter would be "test.servlet_counter". The students can also use library procedures that allow absolute addresses for files, but this is discouraged.

4.1.6 Data Structures and `map`

Students naturally want to handle list-style data (e.g. multiple checkboxes in form data). This leads naturally into a description of "map" and also to table abstractions. We find it useful to introduce map before car, cdr, cons, since it provides a powerful and intuitively clear operation and does not require an understanding of recursion. Moreover, as the examples in Figure 7 below illustrate, the `map` procedure gives the students most of what they need to handle lists of data values. There is also a `map*` procedure which uses a generalized map that converts Java collection objects into lists, and hence can be used with arrays, hashtables, etc.

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

Figure 7: Generating lists and tables

4.2 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 Jscheme applet and runs on the client's browser. Likewise, Jscheme programs that end in "snlp" are converted into Java Network Protocol format which will be automatically downloaded and run in the Java Web Start plugin.[8].

```
"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 t (maketagger))
(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:")
     (t "years" (textfield "" 20))

     (label "Seconds:")
     (t "secs" (label ""))

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

Figure 8: A sample SNLP program

Jscheme has also been extended to allow students to learn to implement simple programs with Graphical User Interfaces. We have written a library, JLIB, that provides declarative access to the AWT package (There is also a version for the Swing package). An example of a simple Scheme program using this library is shown below in Figure 8. The first five lines of the program listed above are strings that provide documentation about this program which is required by the Java Network Launching Protocol (JNLP).

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

4.2.1 JLIB

The JLIB model is based on five fundamental concepts:

- COMPONENTS – there are a small number of ways to construct basic components (buttons, windows, ...)
- LAYOUTS – there are a small number of ways to layout basic components (row, col, table, grid, ...)
- ACTIONS – there is a simple mechanisms for associating an action to a component
- PROPERTIES – there are easy ways for setting the font and color of components
- TAGS – this is a mechanism for giving names to components while they are being laid out.

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. The readexpr and writeexpr procedures are similar, but they allow reading and writing of Scheme expressions on GUI components. For example, the following snippet of code defines a button which changes state when pushed:

```
(define (flip x)
  (case x
       (("on") "off")
       (("off") "on")))
(define B
  (button "off" (action (lambda(e)
      (writestring B (flip (readstring B)))))))
```

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 first few arguments of these procedures are mandatory (e.g. window must have a string argument, textfield requires a string and a integer number of columns). The remaining arguments are optional and can appear in any order. Examples are fonts, background colors, and actions.

The JLIB package provides a "tagger" procedure which allows one to give names to components *in situ*

- `(define t (maketagger))` creates a tagger,
- `(t NAME OBJ)` assigns the NAME to the OBJ and
- `(t NAME)` looks up the OBJ with that NAME.

This makes the code more declarative because the name for a textfield appears with its constructor in the expression that creates the GUI.

4.2.2 Graphics and Animation

We also provide a simple graphics library providing access to a canvas with an offscreen buffer. The drawing primitives are the Java instance methods of 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 9 shows a simple example drawing a red ball moving across a blue background.

```
(jlib.JLIB.load)
(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
  (.setColor g blue)
  (.fillRect g 0 0 1000 1000) ;; clear background
  (.setColor g red)
  (.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)
```

Figure 9: Graphics programming

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 their own thread [9]

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

4.3 Networking Abstractions

After spending two weeks mastering the JLIB library we introduce network programming using 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 an object, `S`, that can 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 10. 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 program which allowed any number of students to join in a game of pictionary 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 which 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.

```
(jlib.JLIB.load)
(jlib.Networking.load)
(define (chatwin
          UserName ChatGroup Host Port)
    (define t (maketagger))
    (define S (make-group-client
         UserName ChatGroup Host Port))
    (define w (window "test"
      (col
        (button "quit" (action (lambda (e)
              (S 'logout) (.hide w))))
        (t "chatarea" (textarea 20 50))
        (t "chatline" (textfield "" 50
         (action (lambda(e)
          (S 'send "chat" (string-append
             UserName ": "
             (readstring (t "chatline"))))
          (writeexpr (t "chatline") "")
        )))))))
      (S 'add-listener "chat" (lambda R
       (appendlnexpr (t "chatarea") R)))
      (.pack w) (.show w)
      w)
(define (rand N)
       (Math.round (* N (Math.random))))
(chatwin
      (string-append "user-" (rand 1000))
      "chat"
      (.getHost (.getDocumentBase thisApplet))
      23456)
```

Figure 10: A multi-room chat program

5 Student Evaluation Strategies

We have used several techniques to accommodate the non-science students that are a majority in this class. The homework assignments allow students to exercise their creativity in creating a web artifact (webpage, servlet, applet, application) which must meet some general criteria. For example, in one assignment they are required to create a servlet that uses several specific form tags (in HTML) and generates a webpage in which some arithmetic computation is performed. This encourages a bricolage approach to learning programming concepts which seems to appeal to non-science majors.

The course features weekly quizzes which take an opposite approach. The students are shown a simple web artifact and asked to write the code for it during a twenty minute in-class quiz. This practice helps keep the students from falling behind in the class and also helps counterbalance the openness of the homework assignments.

The final exam is based on the weekly quizzes so the quizzes also prepare students for the exam. The course provides a high level of teaching assistant support and uses peers who have completed the course in a previous year. The students post their homework assignments on

the web and are thereby able to learn from each other, while the creativity requirement and the sheer joy of creating keeps copying to a minimum.

In the most recent class the three hour open-notes final exam required students to write a webpage, a Scheme servlet, a Scheme applet, and to trace through Scheme code, a logic circuit, and a CMOS circuit. The goal of the exam was to test their ability to synthesize solutions to problems using the tools they had learned.

5.1 Pitfalls

The course requires a substantial investment in TA resources and in class preparation time as there is no textbook for the course. Indeed the course has been heavily revised each year to include more web programming. We are currently working on a textbook which should lessen the class preparation time.

The fact that the course is taught as a large lecture course makes it difficult to keep track of the students who are doing poorly. This is partly ameliorated by weekly quizzes which help track student performance. Smaller class sizes or sectionals might make it easier to track students, but would require a greater commitment of staffing resources.

The current version of software tools used in the course (debuggers, help systems, etc.) are not as well-suited for novice programmers as are other more mature systems (e.g. DrScheme), but they are available as applets so there is a tradeoff between ease of access and ease of use. We are strongly considering porting the class to DrScheme and/or other Scheme systems.

Although the course covers a great deal of material and requires the students to demonstrate their mastery of it in timed quizzes and exams as well as substantial homework projects, the grades are always highly skewed toward the top. This suggests that the class should be taught in two or more sections as the very best students are clearly not being sufficiently challenged. For these students a modified version of the course which included more "algorithmic" computer science would be ideal. This would, again, require a greater commitment of department resources to the non-major course offerings.

6 Lessons learned

Overall the most surprising aspect of the course is that these non-science students have been able to learn how to write servlets, applets, and applications in Scheme, all within a 6 week unit of a 13 week semester. Although they have not delved deeply into "algorithmic" computer science, most of the students do thoroughly understand the mechanism by which a computer program can specify the appearance and functionality of simple applets and servlets. They also understand the notion of a formal semantics (the substitution model) for a computer language and the idea of the evolution of a process as a model of computation as in SICP [1].

The primary reasons for the success of this approach seems to be two-fold:

- **Scheme reduces cognitive overload.** By using a subset of Scheme we eliminate the problem of learning complicated syntax (as one must only match parens (of various sorts) and quotes and the Jscheme IDEs help one do this) and also minimize the problem of learning the underlying abstract machine due to the declarative nature of the language. They can understand the Scheme programs they write using a combination of the substitution model with an intuitive notion of objects (window, buttons, label, menus), events (button pushes, choice selections), and simple operations on these objects (reading/writing data from GUI components or HTML fields). If we were to use Java for this class they would be exposed to a much more complicated model with different kinds of methods (static/instance/constructor), variables (static/instance fields, local variables, parameters), types (classes, interfaces, scalars), and a dizzying array of packages. The use of Jscheme reduces all of the Java libraries to a set of primitive procedures and greatly reduces cognitive overload.
- **JScheme makes applets and servlets easily accessible to non-majors.** By using a Scheme implemented in Java we are able to maintain strong student interest by embedding Scheme in applets, servlets, and JNLP applications and thereby allowing the students to develop web artifacts that are usually only accessible to upper level Computer Science majors. Most of these types of applications could be made accessible through other Scheme implementations. Applets would require a plug-in, but students would probably be just as excited (if not more excited) about creating double-clickable GUI applications in Scheme, which would not require a plug-in.

Acknowledgment

I would like to acknowledge the support of the steadily growing Jscheme community, including my co-developers Ken Anderson and Peter Norvig. I would also like to thank the referees of Scheme2002 for their detailed comments as well as the referees from the ICFP02 conference, who provided some excellent suggestions for improving the paper, even though it was not accepted to ICFP02. 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.

References

[1] H. Abelson and J. Sussman. Structure and Interpretation of Computer Programs MIT Press.

[2] Kenneth R. Anderson, Timothy J. Hickey, Peter Norvig "Silk: A Playful Combination of Scheme and Java" Proceedings of the Workshop on Scheme and Function Programming Rice University, CS Dept. Technical Report 00-368, September 2000.

[3] Ken Anderson and Timothy J. Hickey, "Reflecting Java into Scheme" Proceedings of Reflection 99, Springer-Verlag, Lecture Notes in Computer Science, v. 1616, 1999.

[4] William Clinger and Jonathan Rees, editors. "The revised^4 report on the algorithmic language Scheme." In ACM Lisp Pointers 4(3), pp. 1-55, 1991

[5] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, Matthias Felleisen. *DrScheme: a programming environment for Scheme.* Journal of Functional Programming 12(2): 159-182 (2002)

[6] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. *DrScheme: a pedagogic programming environment for Scheme.* Proc. 1997 Symposium on Programming Languages: Implementations, Logics, and Programs, 1997.

[7] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi. *How to Design Programs.* MIT Press, 2001.

[8] Timothy J. Hickey, Peter Norvig, and Ken Anderson "LISP - a Language for Internet Scripting and Programming", (.ps.gz 130K) in LUGM'98: Proceedings of Lisp in the Mainstream, Nov. 1998, Berkeley, CA.

[9] Timothy J. Hickey, Richard Alterman, John Langton. "TA Groupware" Tech. Rep. CS-02-222, CS Dept. Brandeis University, 2002.

[10] Kurt Normark, "Programming World Wide Web pages in Scheme" Sigplan Notices, vol. 34, no. 12, 1999.

[11] Eric Roberts. *An overview of MiniJava.* in SIGCSE'00 ACM Digital Library, 2000.

[12] David Reed. *Rethinking CS0 with Javascript.* in SIGCSE'00 ACM Digital Library, 2000.