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.

1 Introduction

There are two general approaches to teaching a Computer Literacy (CS0) 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 CS0 class that focuses on programming in some particular general purpose language, (e.g. Javascript [9], Scheme[5], MiniJava[8]).

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]) or by using an internet-based programming language (e.g. Javascript[9], MiniJava[8]).

In this paper we describe a five year experiment in combining these two approaches using 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 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 fairly sophisticated 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

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

have been evolving steadily over the past five years and will likely continue to do so.

This approach to teaching CS0 is feasible only 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 (CS0 or CS1) has been discussed recently by Roberts [8] 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 syntax of Scheme (matching parentheses and quotes is the only syntactic restriction) while maintaining access to all of the classes and objects of Java.

Another recent approach to CS0 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 CS0 course [9] in which about 15% of class time is devoted to HTML, 50% to Javascript, and 35% to other topics in computer science. The Jscheme approach allows for a similar breakdown but in addition allows the students to also 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.

A gentler introduction to Scheme[5] has recently been proposed as a CS0 course which is appropriate (and in fact important) for students in all disciplines. This course has as its goal teaching almost all of the Scheme language in an introductory course. This leaves little time is left for other topics (e.g. computer architecture, chip design, internet programming, ethical and legal issues in computing), but that can be a worthwhile trade-off.

In the Jscheme approach we provide an introduction to only a subset of the language (introducing lists only toward the end) and we introduce some high-level declarative libraries for teaching an event-driven model of GUI construction. Since the Scheme section of the course requires only about 6 weeks, it leaves the other half of the course for standard Computer Literacy topics.

3 Role of Programming in CS0

We take a language-based approach to Computer Literacy where each section of the course focuses on one particular "layer" of computation and explores that layer through the predominant language. The students learn how to use each language to control this level of computation (although we do not currently have them program in real assembly language or build real CMOS devices). A rough outline of the curriculum, 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, 6] an open source implementation of Scheme in Java. 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 course makes use of a small but powerful subset of Scheme and also relies on 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).

One fundamental problem with teaching a course in which students will be asked to write servlets is providing access to a machine that has that capability. We have developed a simple Jscheme webapp which plugs into the open source Jakarta Tomcat web server[3] and allows students to edit and upload their files to this common server directly from their browser.

After registering for the course, the students visit the page on the server they want to edit or create and add a U to the end of the name. The server then generates a form that allows them to either upload the file or edit it in a textarea input form. If they supply the correct password, the file will be uploaded to the server. The advantage of this approach is that it gives them full access to their directory (including making subdirectories), and it requires no tools other than a web browser.

Another useful tool developed for this course is the TATool. This is a groupware application, written in Jscheme and delivered as a Java Web Start application or as a jar file. It provides a simple IDE for scheme development as well as providing FTP-like services to the course web server. In addition it allows students to connect with Teaching Assistants using a IM-like buddylist mechanism. The TAs can then view the students program (but not edit it) and both students and TAs can comment on the program line by line in adjacent textareas. A shared whiteboard and a standard chatroom are also provided. The TATool allows each TA to connect to many students (using a tabbed pane with a separate tab for each student). Each student can also connect to many TAs (again using the tabbed pane model). Students can ask each other for help if no TAs are available. This tool has only been available as an early prototype in the recent CS0 course, but will be fully deployed starting in Summer 2002.

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

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

[3] http://jakarta.tomcat.org

5 Scheme Servlets

Files which appear in the Jscheme webserver student directory with the extension sssp are interpreted 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[4]

For example, using quasi-string notation we can write

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

which is equivalent to the following (less elegant) standard Scheme expression:

```
(define (captioned-image C I)
 (string-append
 "<table border=5>
    <tr><td>
      <img src=\"" I "\" alt=\"" C "\">
    </td></tr>
    <tr><td>" C "
    </td></tr> </table>"))
```

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

[4] 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.

```
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
```

Figure 1: Java reflectors in Jscheme

5.1 Dynamic content

The first simple, 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.

Scheme servlets interact with the Tomcat environment through three Java servlet variables: `request, response, httpservlet`, (which are allowed to appear free in the quasi-string code). The `request` variable is used to get the HTTP headers For example, to get the client's operating system type (from the header) we can use

```
[(.getHeader request "UA-OS")]
```

The `response` variable is used to set the MIME type (e.g. to return `text/plain` instead of HTML and to redirect the page, We can set MIME-type to an image type and redirect to an image as follows:

```
(.setContentType response "image/gif")
(.sendRedirect response "images/picture.gif")
```

The `httpservlet` variable can be used to get information about the scheme servlet (this is not often used by students, but is useful for experts).

5.2 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.

```
;; mylib.scm
   (define (captioned-image C I)
    {<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
  (load "webapps/user/tim/mylib.scm")
  (generic-page

    "Demo page for ICFP talk"

    "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")]
    })
```

Figure 3: Using HTML abstraction libraries

This technique for abstracting HTML is well-known is Lisp/Scheme web programming (e.g. LAML[7], BRL[5]) and is similar to Server-Side Includes in JSP[6] or the publishing model of the Zope environment[7].

5.3 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):

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 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.

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

```
(servlet (password bg fg words)
 (load "webapps/user/tim/mylib.scm")
 (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

5.4 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.

This requires three new ideas:

- data types (converting strings into numbers using `Double.`)
- evaluation of arithmetic s-expressions
- 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.

5.5 System Interaction

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

```
(servlet (inches pounds)
 (load "webapps/user/tim/mylib.scm")
 (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 (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))))
   (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

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.

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.

5.6 Data Structures and smap

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 `smap` procedure (which applies string-append to the result of map) gives the students most of what they need to handle lists of data values. The `smap` procedure uses a generalized map that also converts Java collection objects into lists, and hence can be used with arrays, hashtables, etc.

```
(servlet()
  (load "webapps/scheme/lib/cs2a.scm")
  (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

```
(define smap (lambda R
     (apply string-append (apply map* R))))
(define (li x) {<li>[x]</li>})
(define (lis L) (smap 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) (smap tableitem Ts))
(define (tr Ts) {<tr> [(tds Ts)] </tr>})
(define (trs Rs) (smap tr Rs))
(define (table Rs) {<table> [(trs Rs)] </table>})
```

Figure 7: Generating lists and tables

6 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].

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

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

```
"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

program which is required by the Java Network Launching Protocol (JNLP).

6.1 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.

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.

We have found that this makes the code more declarative in that the name for a textfield appears with the constructor for the textfield in the expression that creates the GUI.

6.2 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 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 9 shows a simple example drawing a red ball moving across a blue background.

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.

```
(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

7 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 .  restargs)
```

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.

```
(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

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.

8 Experience

We have used Jscheme and its predecessors to teach a large Introduction to Computers course for the past five years. The classes have ranged in size from 150-250 students whose majors are evenly distributed across the liberal arts departments.

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 exam. 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 they also serve a role in preparing 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.

9 Conclusions/Future Work

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

- 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.
- by using a Scheme implemented in Java we are able to easily embed Scheme in applets, servlets, and JNLP applications and thereby allow the students to develop web artifacts that are usually only accessible to upper level Computer Science majors. (Much of this could be done with non-Java schemes, but would require custom plug-ins, web-servers, etc.)

We have also found that Scheme provides an ideal vehicle for introducing key CS concepts such as formal syntax and semantics (e.g. students are introduced to the substitution model of Scheme and given quizzes in which they must trace the evolution of a Scheme process).

Another advantage of Jscheme is that it is quite easy to implement declarative libraries providing access to Java packages (e.g. the Swing library is only a few pages of code, as is the code for implementing applets and servlets, and for accessing databases, email, and file I/O).

We are experimenting with using this curriculum to teach computer science concepts in a transitional year program whose aim is to prepare high-potential students from under-resourced high schools for admission to Ivy League universities.

The Jscheme approach could still be improved. Many of the non-science students find the process of writing a program by themselves to be an isolating and frustrating experience. We are looking into introducing online Pair programming as a required part of the course and we have been developing some peer-to-peer tools which will allow student to get online support for Teaching Assistants. These groupware tools are written, of course, in Jscheme as is all of the course management software.

Acknowledgment

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.

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, 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.

[6] 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.

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

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

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