Scheme-based Web Programming as a basis for a CS0 Curriculum*

Timothy J. Hickey†
Department of Computer Science
Brandeis University
Waltham MA, 02254
USA
tim@cs.brandeis.edu

ABSTRACT

The thesis of this paper is that Scheme-based web programming is a worthy organizing topic for CS0 computer literacy courses. We describe an approach to introducing non-science majors to Computer Science by teaching them to write webpages using HTML and CSS and to also write applets and servlets using Scheme. The programming component of our approach is completed in about nine weeks of a thirteen week course, leaving time for a treatment of more traditional CS0 topics such as intellectual property, privacy, artificial intelligence, the limits of computability, PC architecture, Operating Systems, CMOS and logic circuits. We argue that the use of a high level scripting language (like Scheme) is essential to the success of this approach. We also argue that wide scale success in teaching web programming to non-majors could enhance the students productivity when they enter the job market, and hence this approach deserves further study.

Categories and Subject Descriptors

K.3 [**Computers and Education**]: Computer and Information Science Education—*Computer Science Education*; D.3.2 [**Language Classifications**]: Applicative (functional) languages

General Terms

Non-majors, programming languages/paradigms, web-based techniques,CS1/2, course pedagogy, curriculum issues

Keywords

Scheme, HTML, CSS, servlets, applets

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

SIGCSE '04 USA

1. INTRODUCTION

There are many approaches to teaching a 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. A more traditional approach is the CS0 class that focuses on programming in some particular general purpose language, (e.g. Javascript [6], Scheme[4], MiniJava[5]). The primary challenge in teaching a breadth-first course is to provide students with a critical understanding of how computers and the internet works. The challenge in a programming-based course on the other hand is to make the course interesting and relevant to non-majors, who may very well never take another Computer Science course.

Several authors have proposed merging these two approaches by using a simpler programming language (e.g. Scheme[4]) or by using an internet-based programming language (e.g. Javascript[6], MiniJava[5]). In this paper we describe a seven year experiment (1997-2003) in combining these two approaches by organizing the course around a study of web programming using a scripting language. We currently use Jscheme[2], a dialect of Scheme implemented in Java, but the course could conceivably be taught in other lightweight languages such as Javascript, Python, or Ruby.

2. RELATED WORK

The need for a simple, but powerful, language for teaching introductory CS courses (CS0 or CS1) has been discussed by Roberts [5] 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). 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 [6] in which about 15% of class time is devoted to HTML, 50% to Javascript, and 35% to other topics in computer science. A third related approach is to teach Scheme directly as a first course. This 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[4] has recently been proposed as a CS0 course which is appropriate (and in fact important) for students in all disciplines. Unfortunately, by

attempting to teach the full Scheme language in an introductory course, little time is left for other topics (e.g. computer architecture, chip design, internet programming, ethical and legal issues in computing). The approach we are proposing here provides an introduction to only a subset of the Scheme language (introducing lists only toward the end) and introduces some high-level declarative libraries for teaching an event-driven model of GUI construction.

In the next section of the paper we give a brief introduction to the material we teach in the servlet and applet sections of the course. We then discuss the merits and problems with this approach based on our experience teaching this materal to about a thousand non-majors over the past seven years.

3. SCHEME SERVLETS

We begin our study of Scheme servlets after first spending three weeks introducing the students to the architecture of the Internet (IP addresses, ports, routers, domain name servers, protocols) and teaching them the fundamentals of HTML and CSS as simple programming languages. At this point they will have been exposed to the ideas of formal syntax and semantics, nesting of expressions, and some abstraction in CSS (where you can give names to complex style specifications).

Scheme is introduced initially as a method for adding dynamic content to HTML. We provide students web-based access to a server which is dedicated to the student accounts. They upload their scheme programs (using a web-based form) to that server. The modified webserver treats any programs ending in ".servlet" as Scheme servlets – when a browser tries to view such a file, the webserver treats the file as a program which is evaluated by the Scheme interpreter to produce a string that is then returned to the client.

To simplify the transition from HTML/CSS to Scheme, we use a dialect of Scheme (called Jscheme) that provides a syntactic extension to simplify the construction of strings containing generated content. These new strings are called quasi-strings and are formed using curly braces instead of double quotes.

```
{ ..... }
```

In addition, we allow one to escape to Scheme inside the quasi-string by using square brackets:

```
[ ... ]
```

The expression inside the square brackets is evaluated in Scheme and converted to a string which is then appended into the current quasi-string. This is similar to the quasi-quote syntax used to construct expressions in Scheme.

We can illustrate these ideas using the following simple Scheme servlet which generates a webpage with the browsers IP address:

```
{
  <html>
    <head><title>Your IP address</title></head>
    <body>
      Your IP address is
        [(.getRemoteAddress request)]
    </body>
  </html>
}
```

Evaluating this expression yields the following string

```
<html>
  <head><title>Date/Time</title></head>
  <body>
     Your IP address is
       129.64.2.155
  </body>
</html>
```

The students find the quasi-string syntax easy to learn and are excited about adding dynamic content to their pages (e.g. current date/time, host's operating system, and other HTML header items). At this point, Scheme simply consists of a way of accessing the HTML headers using a "[(...)]" syntax and is easy for them to understand.

The next step is to introduce I/O. We do this by providing a macro `servlet` that allows the student access to the HTML parameters which are specified in the argument list. For example, they could create a webpage with a variable text and background color using:

```
(servlet (textcolor bgcolor)
 {<html>
   <head><title>Demo</title></head>
   <body>
   <div style=
        "background:[bgcolor];
         color:[textcolor]">
    <h1>Color demo</h1>
      The text is [textcolor]
      and the page is [bgcolor].
   </div></body></html>})
```

The servlet can then be "run" by adding parameters to the URL of the servlet in the browser:

```
...colordemo.servlet?textcolor=red&bgcolor=black
```

This introduction to dynamic webpages and the servlet macro is usually covered in one 50 minute lecture and the students can immediately start adding dynamic content to the web pages.

The use of parameters in servlets naturally motivates user-defined procedures which use the same substitution model. These procedure are introduces as a kind of named "super-tag" for generating HTML and are compared to CSS classes which are named sets of style specifications.

For example, we show them how to define a custom webpage procedure which is an abstraction of a class of standard webpages:

```
(define (mypage css title body)
   {<html><head>
         <title>[title]</title></head>
     <body>
       <h1>Tim's Page<br />[title]</h1>
        [body]
       &copy; Tim Hickey 2001
     </body></html>})
```

which can then be used used to simplify webpages:

```
(servlet (textcolor bgcolor)
  (mypage {Demo}
    {<div style=
        "background:[bgcolor];
```

```
        color:[textcolor]">
      The text is [textcolor]
      and the page is [bgcolor]
     </div>}))
```

The semantics here is the substition model[1] and in this domain the advantages of using abstraction are easy to explain as it allows them to create new webpages more quickly and to modify the look of all of their webpages more easily.

We then continue on to introduce new primitives and means of combining expressions to extend their expressiveness. We introduce simple conditionals (`if` and `case`) and arithmetic expressions, which allows them to write simple I/O programs (convert dollars to yen ...). This leads to a renewed discussion of the evaluation of Scheme expressions by successive rewriting and substitution which provides both a nice conceptual model of computation and helps them understand how their servlets work.

Below is a typical example of the kind of servlets these students are writing after the few week of instruction on Scheme servlets.

```
;; project2a.servlet
(my-page (servlet (a s1)
 (case a
  ((#null)
   {<form method="post"
          action="project2a.servlet">
      Please enter an amount in U.S. dollars:
       <input type="text" name="a">
      <br/> <br/>
      calculate<input type="submit" name="s1"
           value="How many shekels?"
    </form>
   })
  (else
    {[a] dollars is equal to
      [(* 4.56 (Double. a))] shekels
    }))))
```

The example was written by a non-science major and has editted slightly to remove her name and other identifying data. The first clause of the case is invoked when the servlet is first visited. The second clause is invoked on the second visit, after the form has been completed. The `case` pattern can also be used to create password protected pages or to combine several webpages into a single page (e.g. for a presentation.)

To make the servlets more fun (and useful) we provided access to the `send-mail` procedure, which allows them to write servlets that automatically construct and send emails. We found we needed to restrict the email to destinations inside the University, as students would sometime write buggy programs that mistakenly sent emails to unsuspecting persons outside of the university.

For persistence, we currently provide the `dbquery` procedure which allows them to make simple SQL queries to a student database and returns the result as a list of lists. This motivates the study of structured data and provides an opportunity to discuss the "map" procedure of Scheme and use it to generate HTML tables of a SQL query results. We don't teach much SQL, but rather show them a few simple SQL patterns (creating a simple table, inserting into a database, selecting records from a database, counting records in a query) and provide a procedure to convert the results into an HTML table. With these simple tools they can begin to create interesting and useful web programs and in the process learn about fundamental programming concepts like formal syntax and semantics, procedural abstraction, side effects, evaluation of expressions, and the substitution model of Scheme.

4. APPLETS

The course then moves on to the study of applets, which are programs than run in the browser window. The course web server has been modified to allow students to write applet programs in Scheme and store them in files ending with ".applet". Any such file is automatically converted into an HTML page containing a Jscheme interpreter applet in which the students code is passed as a parameter. From the student's point of view, they upload a program to server, visit the page they've just created, and their program starts running on their browser.

To allow the students to focus on the fundamental concepts of this type of programming, we provide them with a library (jlib) that provides declarative access to the javax.swing (or just the java.awt) package. An example of one student's simple Scheme program using this library is shown below. Due to the declarative nature of the library, this should be fairly easy to understand without any explanation. This program was written after one week of instruction on Scheme applets. It is not algorithmically complex, but it does illustrate the range of "reactive" programs they are able to create.

```
"Homework 3b"
"STUDENT X"
"http://tat.cs.brandeis.edu:8090/aut02/cs2a"
"money calculator"

(define (init thisApplet)
 (define lib (jlib.JLIB.load))
 (define pennies (textfield "" 10))
 (define nickels (textfield "" 10))
 (define dimes (textfield "" 10))
 (define quarters (textfield "" 10))
 (define totalDollars (label "---"))

 (define w
  (window "money calculator"
          (HelveticaBold 24)
   (border
    (north (label "Money Calculator"
                  (HelveticaBold 40)))
    (center
     (table 5 2
      (label "pennies") pennies
      (label "nickels") nickels
      (label "dimes") dimes
      (label "quarters") quarters
      (button "Total"
        (action (lambda(e)
         (let* (
           (p (readexpr pennies))
           (n (readexpr nickels))
           (d (readexpr dimes))
           (q (readexpr quarters))
           (total
```

```
            (* .01
              (+ (* 1 p)(* 5 n)(* 10 d)(* 25 q))))
              )
        (writeexpr totalMoney total)
        ))))
      totalMoney)
  ))))

(.pack w)
(.show w)
```

The key points about this windowing library are that it 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 arguments are optional and can appear in any order. The library determines how too act on the arguments based on their type, freeing the user from having to remember whether color goes before or after the font, etc. Examples of arguments are fonts, background colors, and actions. Also, several of the GUI widgets (textfield, textarea, label, choice, ...) are viewed as I/O objects and the "readexpr" and "writeexpr" procedures can be used to read/change their displayed values.

The first five lines of the program listed above are strings that provide documentation about this program. If this code is placed in a file with the extension ".snlp" in the course webserver, then it is converted into an XML file using the Java Network Launching Protocol (JNLP), and this causes the program to be downloaded to the client's computer and run in a sandbox. If it is placed in a file ending in ".applet" then the server generates a webpage containing an applet that runs the code in the file.

We also introduce some 2D graphics and a little networking in the section of the class so that motivated students can write their own network games. As an example of how much the students can learn, we had one student with no prior background in Computer Science who implemented a networked game applet for playing pictionary. She implemented the chatting windows, a whiteboard, and demonstrated that it could be played by many people at different computers. Most students however, chose simpler final projects such as quiz games or chat simulators, and some simple created GUI landscapes that the user could traverse by clicking on buttons and menus that would open new windows, change background colors, and write text into textareas.

5. EXPERIENCE

We have used the Scheme-based web programming approach to teach a large Introduction to Computers course for the past seven years (1997-2003) and we would judge the course to be successful in a number of ways.

First of all, the classes have ranged in size from 80-250 students, tracking the rise and fall of the Internet bubble, but the course has remained one of the most popular courses on campus in terms of the number of students enrolling (very few courses at our institution have more than 100 students). Second, the class has continued to attract a wide distribution of students across the liberal arts departments and class years (freshman through senior), so the inclusion of programming has not "scared off" non-science students. Finally, the students have continued to score highly on the final exam, which is a three hour in class programming assignment in which they must write fairly sophisticated servlets and applets without using a computer. Fully half of the students scored 90% or higher on the rigorously graded final exam last year.

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 keeps copying to a minimum.

There are still some remaining rough spots in the course. One problem is the lack of a really good debugger/validator for the Scheme servlets and applets. This is especially tricky for the Scheme servlets where the students are mixing three or more languages – HTML, CSS, Scheme, and possibly SQL. The relatively mediocre quality of our debuggers creates some unnecessary frustration for some of our students, but most of the problems are with paren matching and eventually do get solved. We are working on building better debuggers. Another problem is that we currently teach the course entirely on a dedicated course server, so the students never learn to install and run a server on their home computers. The downside of this approach is that they are not able to apply what they've learned outside of the course. Similarly, if the database component of the course were a little better designed the students ability to write useful programs would be greatly increased.

From a social point of view, some students are still uncomfortable with the solitary nature of programming. We have tried to answer these concerns by scheduling abundant Lab Assistant hours staffed by undergraduates who have recently completed the course. We are currently looking into allowing group homework projects in the hope of enticing more students to continue programing after the class has completed.

There are two requirements for offering this type of course. First, one must find a way to provide the students with easy access to a webserver that can handle Scheme servlets. Second, the instructor needs to carefully select a subset of the language that will be taught to the students and to provide highlevel libraries where appropriate to make the language more declarative. Links to the software and course materials for the class we teach is free, open-source, and available online at

`http://jscheme.sf.net`

Links to some earlier versions of the class are also available.

6. CONCLUSIONS/FUTURE WORK

Overall the most rewarding aspect of the course is that these non-science students have been able to learn how to write fairly sophisticated HTML, CSS, servlets, and applets all within an 8 week unit of a 13 week semester, and they still have time to learn about more traditional Computer Literacy topics such as intellectual property, the internet, operating systems, PC architecture, and logic circuits. This is especially gratifying considering the diversity of the student body in terms of class year and major, as shown in the tables below

	Major
27%	Science
28%	Social Science
15%	Humanities
2%	Arts
27%	Undecided

Year	Fr	So	Jr	Se
	50%	24%	13%	13%

The overall course evaluations were generally positive although some students felt the course moved too quickly and others that it moved to slowly. Most were happy to have learned to create webpages and many were surprised by how much they had learned.

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 syntax (as one must only match parens 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. Rather than learning about memory locations and program counters and control flow, they are learning about the substitution model and the evolution of Scheme programs by rewriting text.

- by organizing the course around web programmming we capture the interest of the students who are then willing to devote long hours of programmming to get their servlets or applets working.

We have tried unsuccessfully to organize an introductory Java course around web programming, but this has had much less satisfactory results in that the students learned less than when we followed a more traditional console-based approach, and they weren't able to master the same level of web programming as in the Scheme-based course. These results might be due to the difficulty that novices have in learning a "professional API" like java.awt or javax.swing, but it could also be related to the fact that Java is cognitively more complex than Scheme. It requires students to learn about objects, classes, fields and methods (static and instance) just to write the simplest applet.

Scheme, on the other hand, has proved to be an excellent vehicle for introducing key CS concepts such as formal syntax and semantics, and also allows the students to build interesting and fairly complex web artifacts. Other languages could probably be used equally well in this approach to CS0, but our experience would indicate that high-level scripting languages (like Scheme, Python and Ruby) would have the best chance of success in this domain.

Although we are not quite there yet, it seems likely that in a few more years, we should understand enough about this approach to allow all students, no matter what their major, to learn to program and to have their programming skills enrich their personal and professional lives in a way that the Basic and Pascal programming Computer Literacy classes of the 70's and 80's were never able to accomplish. It is at least conceivable that a large scale effort to teach web programming to our youth will result in a more skilled and productive workforce and citizenry.

Acknowledgment

I would like to acknowledge the support of my Jscheme codevelopers over the years, including Ken Anderson and Peter Norvig, and my students Hao Xu, Lei Wang who helped develop the very first version in 1997. I would also like to thank the 1000+ students that have taken the course over the past seven years and who have helped refine the course by their suggestions and encouragement.

7. REFERENCES

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

[2] Ken Anderson, Timothy J. Hickey, Peter Norvig. Silk: A playful combination of Scheme and Java Workshop on Scheme and Functional Programming Rice University, CS Dept. Tech. Rep. 00-368, Sept 2000.

[3] 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

[4] 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.

[5] Eric Roberts. *An overview of MiniJava.* in SIGCSE'01 ACM Digital Library, 2000.

[6] David Reed. *Rethinking CS0 with Javascript.* in SIGCSE'01 ACM Digital Library, 2000.