Jscheme Web Programming in a CS0 Curriculum

Timothy J. Hickey [1]
Department of Computer Science
Brandeis University
Waltham, MA 02254 //USA

Abstract

We describe an approach to introducing non-science majors to Computer Science by teaching them to write applets, servlets, and webapps using a dialect of Scheme implemented in Java.

1 Introduction

There are two general 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. On the other end of the spectrum is the CS0 class that focusses on programming in some particular general purpose language, (e.g. Javascript [6], Scheme[4], MiniJava[5]).

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[4]) or by using an internet-based programming language (e.g. Javascript[6], MiniJava[5]). 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.

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

Jscheme is an implementation of Scheme in Java (meeting almost all of the requirements of the [3] Scheme standard). It also includes a simple syntactic extension providing full access to Java classes, methods, and fields, as well as a syntactic extension which simplifies the process of generating HTML.

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.

Jscheme has also been build 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 instructors machine, but students can easily download and install the server on their home/dorm PCs as well.

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 [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). 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

SYNTACTIC CONSTRUCT	JAVA MEMBER	EXAMPLE
"." at the end	constructor	(Font. NAME STYLE SIZE)
"." at the beginning	instance member	(.setFont COMP FONT)
"." at beginning, "$" at end	instance field	(.first$ '(1 2))
"." only in the middle	static member	(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

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. 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. This MIT approach, pioneered by Abelson and Sussman [1], is not however 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). 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.

3 Jscheme

Jscheme is an interpreter for Scheme implemented in Java. It is almost completely compliant with the R4RS standard [1] and also provides full access to Java using the Java Reflector syntax shown in Figure 1. Jscheme also provides access to thread and exception handling.

The CS2a class makes use of a small but powerful subset of Jscheme 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 operations and comparisons, a simple GUI-building library (providing declarative access to Swing components, events, and layout managers).

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

4 SXML

Jscheme also provides an optional syntactic extension to simplify the generation of HTML. The motivating problem is that when generating HTML in Scheme one must quote the double quotes, e.g. (define a "<a href=\"b\">c<\a>") and if one wants to mix static and generated HTML, a nested sequence of "string-append" expressions must be used. We simplify the first problem by introducing a new method for quoting strings: <xml> ..... </xml> which simply creates a string out of the text between the open and close xml tags (including any quotation marks that may appear there).

In addition, we allow one to escape to Scheme inside the tags using an <scheme> .... </scheme> element. The expressions inside that element are evaluated in Scheme and converted to strings which are then appended into the current string. This is similar to the quasiquote/unquote-splicing syntax used to construct expressions in Scheme.

We can illustrate these ideas using the following simple Scheme servlet which generates a webpage with the current Date.

```
<xml>
  <html>
    <head><title>Date/Time</title></head>
    <body>
      Current local time is
        <scheme>(java.util.Date.)</scheme>
    </body>
  </html>
</xml>
```

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>
```

The Jscheme webapp in the Jakarata Tomcat servlet has been designed so that any file with the extension ".sxml" is viewed as a Jscheme program.

That program is then evaluated in an environment that contains three Java servlet variables: `request, response, httpservlet`, and the resulting (HTML) string is then sent back to the client that requested that page.

This allows one to easily write servlets that process form data from webpages. For example, after a week of HTML instruction we have found that beginning students are easily able to create forms such as the following:

```
<html><head><title>SXML Demo</title></head>
 <body>
 <form method=post action="demo1.sxml">
   name <input type=text name=name><p>
   age <input type=text name=age><p>
   color <input type=text name=color><p>
   <input type=submit>
 </form>
</body></html>
```

The next step is to write a servlet that processes this data. In the following example, we get the age and color supplied by the user on the previous form and use them to set the background color of the page and compute the person's age in millions of seconds.

```
(let ((age (.getParameter request "age"))
      (color (.getParameter request "color")))
 <xml>
   <html><head><title>SXML Demo</title></head>
    <body bgcolor=
       <scheme>color</scheme>
     >
    Hello, if you are
       <scheme>age</scheme>
    years old,<p> then you are
       <scheme>
       (* (Double. age) 365.25 24 60 60 0.000001)
       </scheme>
    million seconds old.
    <a href="demo1.sxml">Try another?</a>
   </body></html>
 </xml>
)
```

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 boilerplate let expression for reading parameters. We have also added a few additional primitives for reading/writing/appending scheme terms to a file. This allows students to implement counters and log files. For motivated students, there is also a simple SQL library for accessing network databases.

4.1 SNLP

Jscheme has also been extended to allow students to learn to implement simple programs employing a Graphical User Interface. We have written a library that provides declarative access to the Swing (or just the AWT) package. An example of a 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.

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

(jlib.Swing.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)
        (writeexpr (t "secs")
          (* 365.25 24 60 60
            (readexpr (t "years"))))))))))))
(.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 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. We also introduce the idea of a "tagger" procedure which allows one to give names to components – `(t NAME OBJ)` assigns the NAME to the OBJ and `(t NAME)` looks up the OBJ with that NAME. 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 Jscheme webapp of the Tomcat 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.

5 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 keeps copying to a minimum.

6 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 HTML, servlets, applets, and applications all within an 8 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 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.
- 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.

We have also found that Scheme also 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 hoping to add peer-to-peer computing to the set of applications that are covered in the course. This will require developing a simple library for sending scheme terms between applications, and would allow students to build multi-person chats and simple internet games. We are also 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 school districts 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 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.

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.

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'00 ACM Digital Library, 2000.

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