WeScheme: The Browser is Your Programming Environment

Danny Yoo
WPI
dyoo@cs.wpi.edu

Emmanuel Schanzer
Harvard University
schanzer@bootstrapworld.org

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

Kathi Fisler
WPI
kfisler@cs.wpi.edu

ABSTRACT

We present a programming environment called WeScheme that runs in the Web browser and supports interactive development. WeScheme programmers can save programs directly on the Web, making them accessible from everywhere. As a result, sharing of programs is a central focus that WeScheme supports seamlessly. The environment also leverages the existing presentation media and program run-time support found in Web browsers, thus making these easily accessible to students and leveraging their rapid engineering improvements. WeScheme is being used successfully by students, and is especially valuable in schools that have prohibitions on installing new software or lack the computational demands of more intensive programming environments.

Categories and Subject Descriptors

D.3.4 [**PROGRAMMING LANGUAGES**]: Processors; K.3.2 [**COMPUTERS AND EDUCATION**]: Computer science education

General Terms

Design, Languages

Keywords

programming environments, Web

1. INTRODUCTION

A programming environment that runs entirely inside a Web browser offers many benefits to educators and students:

- It is "zero-install", in that any user with a Web browser can use the environment without installing other tools. This is an advantage at institutions that have restrictions on what software can be installed.
- By using Web technologies like the *Document Object Model* (DOM) and *Cascading Style Sheets* (CSS), it can reuse the engineering effort of several companies who are competing to add features and performance.
- Giving the students access to the Web's display technology offers them an incremental path from Web authoring to programming, i.e., from statics to dynamics.
- It can allow for the development of *mashups*, programs composed of several Web applications working together. It offers the promise of deep interoperability with content and programs on the Web, like Flickr and Google Maps, and makes for interesting and novel programming exercises.
- It suggests storing programs in the Cloud, which enables easy global sharing. An important special case of "sharing" is with oneself: students can easily begin work at school, resume at home, continue again at school and so on, always having access to their "files".

We present WeScheme, a Web-based programming environment for the Scheme [11] and Racket [5] programming languages. It provides a syntax-highlighting program editor, an interactive tool to run programs on-the-fly, and a hub for sharing programs. Beneath the surface, WeScheme provides a sophisticated runtime that enables interactive programs to be written in a sequential, synchronous style, a model that is particularly well-suited to beginners for its simplicity.

2. EDUCATIONAL CONTEXT

WeScheme is used primarily to support Bootstrap [10], an educational program designed to help middle- and high-school students see the ties between the mathematics they are learning, and computation. At this stage students are studying algebra, coordinate geometry, and simple modeling. The Bootstrap curriculum uses just these concepts to create interactive animations and games [2]. This context helps students find the mathematical concepts appealing and thus more approachable. In turn, students eventually realize that mathematics is not just a dry textbook discipline but one that has direct application to topics they care about.

Because of this algebra-rich curricular context, we make heavy use of *functional* programming. In functional programming, programmers write functions in the mathematical sense: a function consumes inputs, produces a value, and is deterministic. It computes the value using algebraic

ITiCSE 2011, June 27–29, 2011, Darmstadt, Germany

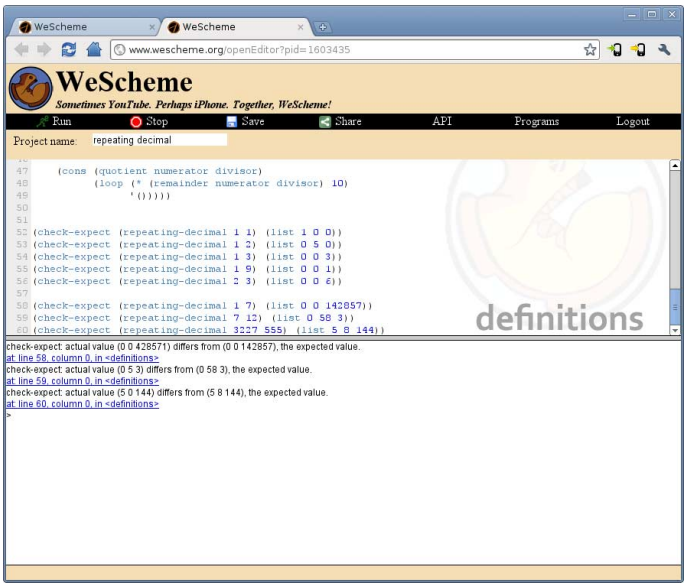

Figure 1: WeScheme

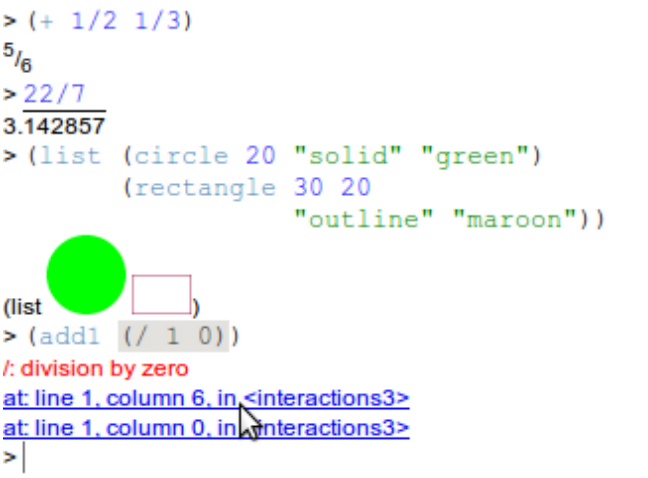

Figure 2: Examples in the REPL

substitution, as taught in schools. The kinds of values in WeScheme are much more compelling than those presented in a standard math textbook: rather than just numbers, they can be strings and booleans, but also images, sounds, and graphical animations. This combination of a simple computational model (functions and substitution) over rich values (strings, images, etc.) proves to be sufficient for writing quite sophisticated programs, including interactive games. A full discussion of the details of this curriculum is given in our textbooks [1] [3].

This educational mission has an impact on the design of WeScheme. Though the underlying virtual machine is very general and supports imperative programming, object-oriented programming, and more [5], many of the technical obstacles we have had to overcome have been to support the algebraic model. WeScheme can thus provide a simple programming interface for many Web programming tasks.

3. A QUICK TOUR OF WESCHEME

Figure 1 shows a screen-shot of WeScheme running inside the Google Chrome browser. (The environment looks essentially identical in other browsers.) The interface, which borrows heavily from DrRacket [4] (formerly DrScheme), is intentionally simple.

The toolbar at the top has the following commands:

- The Run button loads the program's definitions for use in the REPL.
- The Stop button interrupts the running program.
- The Save button saves the program onto the Cloud.
- The Share button allows the user to freeze the current program and produce a stable URL that refers to it. Accessing the URL presents an interface to look at the source code, if allowed by the owner, and to run the program outside the editing environment.

Below the toolbar is the *Definitions* pane which contains a rich-text program editor (currently a fork of CodeMirror). The editor include a syntax highlighter with color-coding, parenthesis-matching, and context-sensitive indentation, all of which help users with the editing process.

The lower pane contains the REPL. A REPL presents a calculator-like interface to a program. A REPL allows a programmer to explore a program's definitions directly without having to create external binaries, etc. Instead, when the user enters an expression at the prompt, the REPL evaluates it, prints its value, and presents a fresh prompt. The use of a REPL allows for lightweight exploration of programs, and its interface can help cement the relationship between algebra and computation.

WeScheme's REPL makes heavy use of the browser's display and interaction technology. The screenshot in figure 2 shows four examples. In the first, the user is examining a fractional value; in the second, a repeated decimal. In both cases, WeScheme presents them using representations based on those of math textbooks. In the third example, the user evaluates an expression whose value is a list of images, which are all displayed in-line. In all three cases, WeScheme exploits the browser's display technology—JavaScript, canvases, and style-sheets—to present its output. The fourth example shows an error. Error messages are presented as hyperlinks, and clicking highlights the relevant region of code.

Users can browse through their list of programs, and edit, share, and delete them. Figure 3 shows such a listing. This list, which takes the place of a typical filesystem, resides on a cloud server, so the user sees the same list no matter where they log in. In the entry for "Baduk", the sharing icon is grey because that program has not yet been shared. When a user chooses to share a program, WeScheme shows the dialog in figure 6. This lets the author choose whether or not to divulge the program source. Once shared, WeScheme generates a stable link; hovering over the sharing link (now green) in the console, as shown in figure 4, provides this link, and also the ability to upload it to various social networking sites. Users who visit the shared URL will see the window in figure 5, where they can run the program and, if allowed, read the source code.

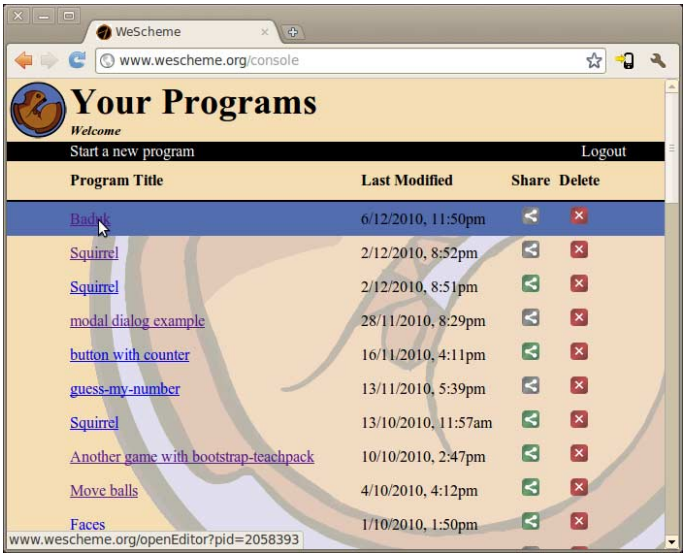

Figure 3: Program List

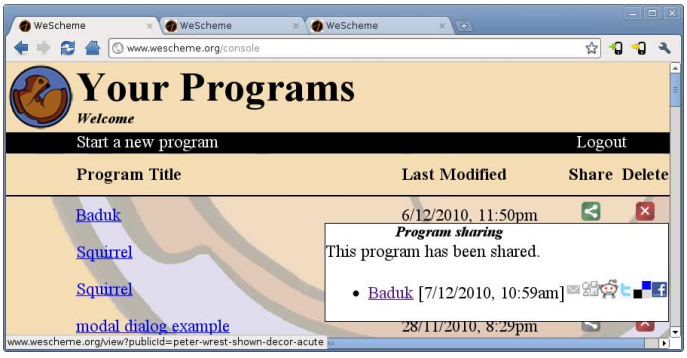

Figure 4: A Shared URL Link

Figure 5: Visiting a Shared Program URL

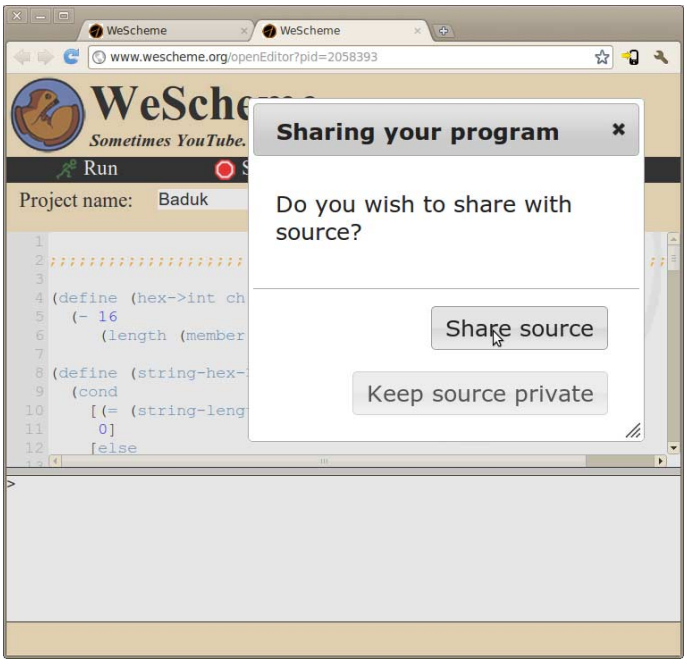

Figure 6: Sharing Dialog

4. IMPLEMENTATION TECHNOLOGY

WeScheme uses a compiler running on a cloud server. When the user clicks Run, the program is sent to the remote compiler, which converts the source into bytecodes. Likewise, every time the user types an expression at the REPL, it is sent to the server for compilation.

Of course, this architecture assumes continuous access to a server. This assumption is common in many contemporary Web-based systems such as Yahoo! Mail, Google Maps, etc. However, there is no difficulty in generating a binary that can be installed on the local host that provides access to the compiler without the need for networking support; we have simply not experienced demand for this yet. In this setup WeScheme would still run in the browser, but to initiate it the user would connect to a URL on the local machine rather than to `www.wescheme.org`. A launcher could automate this process by automatically feeding the URL to the browser, so that the details are sufficiently transparent to the user.

The bytecodes generated by the compiler are those of the Racket virtual machine [7]. The reader can think of this as analogous to the bytecodes of the Java Virtual Machine, though in fact the Racket language is richer in many ways and can thus support a host of other programming languages ranging from Java to Python (and the Racket project has even had experimental support for these other languages). We have implemented an interpreter in JavaScript for these bytecodes, relying on the threaded virtual machine technology [6] of most contemporary browser JavaScript implementations to optimize uses of the interpreter.

The programs that users write are similarly automatically backed up to and saved on the Cloud. We currently use Google's AppEngine for this purpose. When using a local compiler, we could save files to the local filesystem and periodically synchronize them on demand.

By virtue of exploiting the browser, we immediately and

automatically inherit improvements made by browser implementors. For instance, Web browsers recently added support for embedding videos inside Web pages in preparation for HTML 5. For a user to include a video in the output page (for instance, as a backdrop to a game) required no additional work from us at all. To make these videos programmatic objects—so that the user could, for instance, query the video or send it commands—required only a small amount of wrapping to make it an object in our virtual machine, on the order of about ten lines of code, most of which are boilerplate.

5. EXPERIENCE AND EVALUATION

WeScheme is in active use by students in Bootstrap, the program described in section 2. Students have created over 1000 programs in WeScheme, and over a quarter of these have been shared, most of them with the source made public. (These numbers naturally change on a daily basis.)

Many WeScheme users teach at schools with extremely limited computing infrastructure. In particular, they face two different kinds of limitations: limited computing power, and locked-down systems. When systems are locked down, it becomes impossible to install a programming environment like DrRacket. When the systems are weak, even if the instructors can install DrRacket, its resource consumption is so great that just starting and running it is virtually impossible. In contrast, these machines can still run Web browsers.

For instance, we recently had a school running Pentium III hardware and Windows 2000 software, on 256 Mb of RAM—a configuration over a decade old. On such a system today's DrRacket will barely start, but students were able to write and run modest programs in WeScheme. As browsers get leaner and more efficient, this gives WeScheme a significant engineering edge.

One of the factors driving improvement in browsers is the use of the same core browser engines in mobile platforms. Indeed, smartphone Web browsers are now sophisticated enough that one can run WeScheme directly in the phone—as we have, though of course using the editor is an exercise in masochism. However, while the phone is a poor *editing* medium, it is perfectly reasonable to *run* programs on phones. As phones are increasingly taken seriously as computing platforms, and browsers are recognized as an important component, our decision to target JavaScript, which may have seemed idiosyncratic, makes sense.[1] Indeed, the compiler underlying WeScheme has also been used in college-level courses that produce mobile phone applications.

The user interface of WeScheme is intentionally spartan, in contrast to the visual complexity of many contemporary programming environments. In this regard, and in many details, WeScheme mimics DrRacket. The differences, however, suggest ways in which DrRacket can improve. Beyond easy sharing, as mentioned earlier, we use hyperlinks to present error reports and also stack traces. DrRacket uses an icon, which users must click on, to represent stack traces. Many students have reported confusion about what that icon represents, and do not even know that it is clickable. In contrast, students have no difficulty understanding the visual metaphor of the hyperlink, and indeed it invites their exploration.

The limited space of this format makes it difficult to provide examples of program source and their output. However, because the programs are on the Web, readers can access them easily! Readers can both run, and view the source of (and thus easily modify and create their own versions of), the following programs: `tinyurl.com/2924s2s` presents a game, while `tinyurl.com/28jptyn` shows a use of the browser's display framework. We suggest trying these in Google Chrome.

6. RELATED WORK

WeScheme provides an on-line programming environment and a deployment platform that live entirely on the Web. Much of the related work in this area contain aspects of this, though often not in combination.

Lively Fabrik [8] is a programming language that runs entirely inside the browser without plugins. It is a simple, visual programming language consisting of components, pins, and connections with a dataflow semantics; these components are dragged and connected in a visual program editor. Yahoo Pipes (`pipes.yahoo.com`) is another specialized visual programming language for defining RSS feeds from data sources on the Web, using a similar set of tools to connect modules and operations together. Each component is implicitly an asynchronous event handler that listens to changes in their inputs. In contrast, WeScheme supports a general-purpose textual language with a strong tie to school mathematics. WeScheme provides synchronous interfaces to the Web's asynchronous programming style (which we have not discussed in the limited space of this paper), a feature that Lively Fabrik and Yahoo Pipes do not support.

Lively Fabrik runs atop the Lively Kernel [12], which is JavaScript augmented by an implementation of Morphic [9], a Smalltalk GUI interface. In contrast, instead of porting a different GUI library, we expose existing Web technology as the user interface platform—thus giving an incremental path from Web page authoring to programming, and also easily incorporating innovations in this rapidly growing field (such as the video example discussed in the paper).

Mozilla Skywriter (`mozillalabs.com/skywriter/`) (formerly known as Bespin) and CodeMirror (`codemirror.net`) are both text editor frameworks that work on the Web. Frameworks like these are needed because the plain `textarea` provided by HTML doesn't provide essential support for editing programs. Both these frameworks provide features such as syntax highlighting and indentation, though they use different rendering strategies: Skywriter uses a `canvas` element to render the editor, while CodeMirror uses nested DOM elements.

WeScheme's editor is based on CodeMirror, extended to support our environment's needs. We prefer CodeMirror's use of the DOM, as it allows a richer programmatic interface: individual elements can be addressed naturally, for both inspection and manipulation (including, for instance, styling with CSS). Using the DOM for a program editor also allows the intriguing possibility of allowing graphical elements to be used in program source code. WeScheme already allows REPL values to be represented as graphical DOM nodes; it should be technically possible to extend this graphical capability to program source as well, as found in DrRacket.

Web-accessible REPLs differ in how much of the work of

[1]For instance, though some phone manufacturers lock down application stores and place limits on choices of programming languages, they still allow the deployment of applications using JavaScript in browsers.

compilation and evaluation is done on the server versus the client. WeScheme takes a middle-of-the-road approach by compiling on the server-side, and evaluating the resulting bytecodes on the client. We compile on the server side so that we can reuse a well-tested, production-level compiler.

Non-interactive evaluators such as those on ideone (`ideone.com`) and REPLs such as those on Try Ruby (`tryruby.org`) or Try Haskell (`tryhaskell.org`) take the user's program, evaluate it entirely on the server side, and return the textual output back to the user's browser. This works well for textual output, but is impractical (due to bandwidth concerns) for richer data like images and videos, and obviously useless for interactive applications like games. Furthermore, these cannot provide the programmer access to the browser's own rich display facilities, such as the DOM.

A server-based REPL has additional problems. These evaluators have a choice of using session state on the server, or re-evaluating the entire sequence of interactions to reconstruct the state of bindings in the REPL. Using session state creates a resource management problem. However, re-evaluating expressions is even more problematic. First, if the definitions are computationally expensive, re-running all the expressions can become intractable. More subtly, re-running computations can produce surprising results. For instance, here is an actual interaction in Try Ruby:

```
>> x = rand(6)
>> x
3
>> x
2
```

That is, the value of `x` appears to have changed between uses even though `x` was not modified! This is because Try Ruby re-evaluates the assignment to `x`, and of course there is no guarantee that it will be bound to the same result. Needless to say, this is a rather confusing interaction. Therefore, this strategy can only be considered useful for toy programs.

In contrast to the strategy of running the program on a server, there are several virtual machine interpreters that run inside the browser, such as HotRuby (`hotruby.yukoba.jp`) and OBrowser (`www.pps.jussieu.fr/~canou/obrowser/tutorial/`). These use implementation techniques similar to those in WeScheme.

At the other extreme from running all computation on the server is the idea of performing all computation on the client. The REPL in wscheme (`wscheme.appspot.com`) (not to be confused with WeScheme) lies at the other end of the spectrum, by running entirely inside the user's browser. It accomplishes this by using the Google GWT compiler (`code.google.com/webtoolkit/`) to compile an existing Java implementation of Scheme (`jscheme.sourceforge.net/`) into JavaScript. While this strategy is attractive from the perspective of disconnected computation, wscheme's REPL has a major technical limitation relative to WeScheme: its evaluator does not implement cooperative multitasking (as required by JavaScript), so it is easy to starve the browser of cycles—thus, for instance, wscheme cannot implement a Stop button as found in WeScheme. In addition, its REPL doesn't produce stack traces with errors, and its error messages are not as informative as those of WeScheme.

Acknowledgements.

We thank Zhe Zhang, Brendan Hickey, Ethan Cecchetti, and Scott Newman for contributions to WeScheme, and Guillaume Marceau for comments on the paper. This work is partially funded by the US NSF and by Google.

7. REFERENCES

[1] M. Felleisen, R. B. Findler, K. Fisler, M. Flatt, and S. Krishnamurthi. *How to Design Worlds*. 2008. `world.cs.brown.edu`.

[2] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. A Functional I/O System or, Fun for Freshman Kids. *International Conference on Functional Programming*, 2009.

[3] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs*. second edition, 2010. `www.ccs.neu.edu/home/matthias/HtDP2e/`.

[4] R. B. Findler and PLT. DrRacket: Programming Environment. Technical Report PLT-TR-2010-2, PLT Inc., 2010. `racket-lang.org/tr2/`.

[5] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. `racket-lang.org/tr1/`.

[6] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based Just-in-Time Type Specialization for Dynamic Languages. *Programming Language Design and Implementation*, 2009.

[7] C. Klein, M. Flatt, and R. B. Findler. The Racket Virtual Machine and Randomized Testing. Technical report, Northwestern University, 2010. `plt.eecs.northwestern.edu/racket-machine/`.

[8] J. Lincke, R. Krahn, D. Ingalls, and R. Hirschfeld. Lively Fabrik: A Web-based End-user Programming Environment. In *Creating, Connecting and Collaborating through Computing*, 2009.

[9] J. H. Maloney and R. B. Smith. Directness and Liveness in the Morphic User Interface Construction Environment. *User Interface Software and Technology*, 1995.

[10] E. Schanzer. Bootstrap. `www.bootstrapworld.org`.

[11] M. Sperber, R. K. Dybvig, M. Flatt, A. van Straaten, R. Findler, and J. Matthews. *Revised^6 Report on the Algorithmic Language Scheme*. Cambridge University Press, 2010.

[12] A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz. Web Browser as an Application Platform: The Lively Kernel Experience. Technical report, Oracle, 2008. `labs.oracle.com/techrep/2008/abstract-175.html`.