

A SCHEME UNIT-TESTING FRAMEWORK

Sara von Mosch
Division of Science and Mathematics
University of Minnesota, Morris
`vonmoss@mrs.umn.edu`

Scott Lewandowski
Division of Science and Mathematics
University of Minnesota, Morris
`swl@mrs.umn.edu`

Janet Kinney
Division of Science and Mathematics
University of Minnesota, Morris
`kinneyj@mrs.umn.edu`

ABSTRACT

Scheme is a programming language used worldwide by many colleges and universities to introduce students to programming and software development. Unit testing is a key component of software development. Frameworks and other testing tools exist and are widely used for languages such as Java, C and C++, but this is not the case with Scheme. Because of this deficiency, vital testing skills are often overlooked until later in a programmer's education. Noticing the need for effective testing tools for Scheme, we created a Scheme Unit Testing Framework. The framework allows users to write tests, execute them as a group, and receive information about the success or failure of each test. Initial feedback has been positive, and there is interest in incorporating unit testing using our framework into the Scheme-based introductory computer science class at the University of Minnesota, Morris.

Background Information

Introduction to Testing

Software testing is an important part of the software development process. In large projects, testing can easily consume 40 to 60% of the total development resources [4]. Thus, it is important to make testing as efficient as possible. One tool that minimizes the amount of time spent on testing, without sacrificing quality or completeness, is a testing framework. Testing frameworks provide a structured environment that facilitates writing and executing tests. Such frameworks are widely used and include C++Builder for C++ and JUnit for Java, which was co-developed by Kent Beck [8].

Unit testing was introduced by Beck in his paper *Simple Smalltalk Testing: With Patterns* [1]. Like other forms of testing, the goal of unit testing is to ensure that each piece of code functions as expected, given any possible situation. This is accomplished through the use of test cases. Each test case focuses on a single function or piece of code. This minimizes the amount of time spent tracking down the source of a failed test. Good testers try to find as many errors as possible. Only through finding and correcting errors can software quality be improved. Test cases typically use boundary conditions, and unexpected input values to verify the overall robustness of the code.

Individual test cases may be bundled together into a test suite. A test suite is simply a collection of tests. Executing a test suite will run each included test case separately. Upon completion, the results of the test cases are reported, alerting the user of any unexpected errors. Errors revealed during testing are corrected in the original code, and the code is then tested again.

Why Scheme?

Scheme is a functional programming language, which is used by nearly 275 colleges and universities worldwide. Almost half of these institutions use Scheme in their introductory courses [6]. Projects like *TeachScheme!* at Rice University advocate introducing students to programming concepts with Scheme, as opposed to more complex object-oriented programming languages [5]. Unlike many other languages, Scheme's syntax is very simple. After only a few hours of instruction, students can begin writing involved programs. This is not typically so with languages such as Java and C++ which have much more complex syntax and semantics [7]. Scheme allows for the gradual introduction of programming principles and language constructs, thus avoiding a steep learning curve [2].

Project Overview

The main goal of this project is the creation of a Scheme Unit Testing Framework that can be used by students, educators, and professionals who work with Scheme. Such a

framework would facilitate the development of better, more correct code, and would allow educators to introduce the importance of careful testing at the very beginning of a student's education.

We feel it is important to model our framework after those already in existence. Doing so not only provides a guide for our research, but should also ease the transition for users to testing frameworks in other languages, such as JUnit for Java. However, Scheme is a very different type of language from object-oriented languages. Though the basic idea is modeled after an object-oriented design, we chose to write our framework entirely in Scheme, using functional programming constructs and data structures. Because of this, the framework is small, simple, and easy to use for programmers already familiar with Scheme. Beck stresses that all of these things are important to keep in mind when developing a testing tool. The tool should not be an obstruction to writing good tests. It should aid the programmer by making testing easier and providing test results in a useful format, thereby encouraging the programmer to continue writing tests [1].

The framework consists in part of a collection of assertions similar to those found in object-oriented programming language testing frameworks. These assertions are available to the user writing test cases. The test cases, along with their respective input values, are bundled together in a test suite that executes each test in turn. When a test case fails, a message containing information about the test that failed is displayed. In addition, the tester can implement specific error messages with details as to the cause of each error. All of these components will be described further in the next section.

To make the framework accessible to anyone already familiar with Scheme, regardless of their knowledge of software testing, we have created a user manual to guide those users who are unfamiliar with testing. The manual contains an introduction to unit testing and instructions that guide the user through sample exercises. It is set up in a tutorial style format, with each instruction explained in detail, sample code provided, and exercises for the user to try on his/her own. An appendix contains the sample code used throughout the tutorial, which is available to the user for their own testing.

The remainder of this paper will follow a format similar to that used in the manual. Information on unit testing, asserts, test cases, and test suites is interspersed with Scheme code examples.

Using the Framework

One of the examples we commonly use in our Scheme-based introductory class is rational numbers. This example has a number of pedagogical advantages. First, students are already familiar with the concepts behind constructing and performing basic arithmetic functions using rational numbers, which means they can focus their efforts on understanding the programming concepts being introduced. Second, the problem is easily decomposed into smaller pieces, allowing students to see how small, focused functions can be composed to solve more complex problems. Throughout the tutorial we

use rational numbers as the vehicle for demonstrating the various features of the framework.

The following is a list of basic functions used in working with rational numbers [7]:

- (numr *r*) – returns the numerator of *r* (a rational number)
- (denr *r*) – returns the denominator of *r*
- (make-ratl *n d*) – constructs a rational n/d in lowest terms with negative sign (if any) in numerator
- (gcd *x y*) – returns the greatest common denominator of the numbers *x* and *y*
- (rprint *r*) – prints rational number *r* to standard output (in n/d form)
- basic arithmetic operations – *r*+, *r*-, *r**, *r*/

The following variables will be used in some of the examples that follow:

- (define *r* (make-ratl 8 -10))
- (define *s* (make-ratl 1 2))

The initial exercises explore the various `assert` functions. An `assert` function takes in either one or two parameters, and returns a Boolean value. For example, `assert-equal?` takes two parameters, and returns the Boolean result of calling Scheme's `equal?` function with the given parameters. The expression `(assert-equal? (numr s) 1)` returns `true`, as expected. The other binary asserts (`assert-eq?` and `assert-eqv?`) work in a similar fashion. The unary asserts (`assert-true?`, `assert-false?`, `assert-null?`, and `assert-not-null?`) each take one parameter and return a Boolean value. These assertions are very similar to Scheme's predicates, so the user should be familiar with these concepts.

Assert functions can be nested and combined with other functions to produce longer, more complicated expressions. Below is an example illustrating this concept.

```
(assert-true?
  (and (assert-equal? (gcd (numr r) (denr r)) 1)
    (assert-true? (> (denr r) 0))))
```

Both nested asserts inside the `and` evaluate to `true`. `(and #t #t)` returns `true`, so the outer `assert-true?` evaluates to `true`.

Test cases are written using the various `assert` functions. Each test case should focus on a single piece of code or feature to test. Test cases commonly check for boundary conditions and unexpected input. The following example uses the expression shown above to form a test case.

```
(define test-make-rational
  (lambda (r)
    (assert-true?
      (and (assert-equal? (gcd (numr r) (denr r)) 1)
        (assert-true? (> (denr r) 0))))))
```

One of the features of the constructor is that it reduces all rational numbers to lowest terms, and moves any negative signs to the numerator. A rational number such as $8/-10$ is not simplified. Passing `(make-ratl 8 -10)` as an input to this test case allows us to verify whether the constructor correctly handles a non-reduced rational with a negative denominator. If the constructor correctly handles this case, the resulting rational will be $-4/5$. The test will then evaluate to true.

Tests are written individually. Each test case is independent not only from the other tests, but also from the suite. This has several benefits. First, tests can be executed in combination, or individually, without concern for how one test may influence the outcome of another. This is a crucial point. Writing tests that interact with each other defeats the purpose of unit testing, and makes locating the cause of an error message very difficult. A second benefit of writing separate tests is that tests can be re-used easily.

Testing continues throughout the lifetime of a piece of software. Even after software has been integrated, and is being used for its intended purpose, the code must still be maintained. During this maintenance period, regression testing is frequently performed to ensure that recent changes have not caused unexpected conflicts. Ideally, the test cases used during development are saved for regression testing. Therefore, it is important to have individual self-contained tests that can be combined with any number of other tests at any point in the life of the software.

In order to run test cases, input values must be provided to each test. These are defined in a test suite, and can be used by the various tests contained within the suite. Once these inputs are created, the user creates a list of test cases, and a list containing the input values needed for each test case (the input values for a particular test case are bundled together in a sub-list). When the test suite is invoked, the tests will be executed from the list sequentially.

Below is another test used in the test suite.

```
(define test-r+  
  (lambda (r1 r2 expected-result)  
    (assert-equal? (r+ r1 r2) expected-result)))
```

The following test suite can be assembled from these sample components:

```
(define rational-test-suite1  
  (lambda ()  
    (let ((a (make-ratl 8 10))  
          (b (make-ratl 3 -4))  
          (c (make-ratl 1 20)))  
      (let ((tests (list test-make-rational test-r+))  
            (args (list (list a) (list a b c))))  
        (let ((list-of-results (run-tests tests args)))  
          (if (all-tests-passed? list-of-results)  
              (display "All tests passed.")  
              (report-failed-tests list-of-results)))))))
```

- The first `let` statement sets up the input values needed to run the tests.
- The second `let` statement creates the list of test cases and the list of the respective input values.
- `run-tests` takes a list of functions and a list containing lists of arguments. It applies the first function to the first list of arguments, the second function to the second list of arguments, etc.
- `all-tests-passed?` is a function that returns true if all elements in the given list of Boolean values are true.
- `report-failed-tests` takes the list of output values generated by the test cases and identifies which tests (if any) failed (i.e. evaluated to false) and prints an error message for those that did.

By default, tests that execute correctly produce no output. Instead, each test that fails produces a single error message on the standard output that clearly states which test failed. This is the result of a conscience choice to minimize the amount of output a user needs to process after running a suite of tests. Because the tests are individually identified, the user can quickly and easily locate the test case that failed.

```
(define rational-test-suite2
  (lambda ()
    (let ((a (make-rat1 8 10))
          (b (make-rat1 3 -4))
          (c (make-rat1 0 3)))
      (let ((tests (list test-make-rational test-r+))
            (args (list (list a) (list a b c))))
        (let ((list-of-results (run-tests tests args)))
          (if (all-tests-passed? list-of-results)
              (display "All tests passed.")
              (report-failed-tests list-of-results)))))))
```

In this example, `test-r+` will try to verify that $8/10 + (3/-4) = 0/3$. This is false, so this test fails. The output alerts the user with the following message: "Test 2 failed." Looking back at the suite, the user can quickly locate the second test `(test-r+ a b c)`. Once located, the error (in this instance with the test case itself) can be corrected.

The user also has the option of inserting customized error messages into their test cases. These may be used to further narrow down where an error is being generated, or to notify the user that a specific set of events has occurred.

Redefining our `test-r+` function, we can insert specific error message to display the two rational numbers formed.

```
(define test-r+
  (lambda (r1 r2 expected-result)
    (if (not (assert-equal? (r+ r1 r2) expected-result))
        (begin
          (display "Expected ")
          (rprint expected-result)
          (display ", returned "))
        #t)))
```

```
(rprint (r+ r1 r2))  
#f)  
#t)))
```

With this new definition of `test-r+`, executing the test suite above would produce the output "Expected 0/1, returned 1/20 ". This is actually the result of an incorrect argument handed to the test (i.e. 1/20 is the correct expected value of `c`, not 0/1), however this same concept applies to finding other types of errors.

Future Work

There is interest in incorporating unit testing using our framework into the Scheme-based introductory computer science course at the University of Minnesota – Morris. This will be beneficial on many fronts. It will allow us the opportunity to observe how introductory level students use the framework. We can then modify it in such a way that students find it convenient and beneficial. Also, the students will be introduced to unit testing nearly three semesters earlier than previous years.

The URL for this project is <http://www.umn.edu/~vonmoss/research.html>. We welcome and encourage any feedback.

References

1. Beck, Kent. *Simple Smalltalk Testing: With Patterns*. Retrieved August 28, 2001 from <http://www.xprogramming.com/testfram.htm>.
2. Bloch, Stephen (2000). Scheme and Java in the First Year [Electronic version]. *Proceedings of the Fifth Annual CCSC Northeastern Conference on the Journal of Computing in Small Colleges*.
3. Gamma, Erich & Beck, Kent. *JUnit A Cook's Tour*. Retrieved August 20, 2001 from <http://junit.sourceforge.net/doc/cookstour/cookstour.htm>
4. Hilburn, Thomas B. & Towhidnejad, Massood (2000). Software Quality: A Curriculum Postscript? [Electronic version]. *Proceedings of the Thirty-First SIGSCE Technical Symposium on Computer Science Education*.
5. PLT Scheme. *The TeachScheme! Project*. Retrieved February 26, 2002, from <http://www.teach-scheme.org>
6. Schemers, Inc. (2001) *Schools Using Scheme*. Retrieved February 23, 2002, from <http://www.schemers.com/schools.html>
7. Springer, George & Friedman, Daniel P. (1989). *Scheme and the Art of Programming*. Cambridge, Massachusetts: The MIT Press.
8. XProgramming.com (2002). *Testing Frameworks*. Retrieved February 26, 2002, from <http://www.xprogramming.com/software.htm>

Acknowledgments

This project was funded by *The Computing Research Association (CRA) Committee on the Status of Women in Computing Research* as part of the *Collaborative Research Experience for Women in Undergraduate Computer Science and Engineering (CREW)* Program.