

## Unit Testing

**Summary:** In the laboratory, you will explore the ways in which small tests can help you develop and update code. You will also familiarize yourself with our testing library.

### Contents:

- Exercises
  - Exercise 1: Testing the Tester
  - Exercise 2: Testing Triangulation
  - Exercise 3: Writing a Triangle Classifier
  - Exercise 4: A Simple Test Suite
  - Exercise 5: Testing Error Checking
  - Exercise 6: Testing Error Checking, Revisited
  - Exercise 7: Symmetric Tests
  - Exercise 8: Testing Large Sides
  - Exercise 9: Testing Small Edges
  - Exercise 10: Other Tests

## Exercises

### Exercise 1: Testing the Tester

As you may recall from the reading, `unit-test.ss` provides four primary procedures:

- `(begin-tests!)`, which prepares the environment for testing.
- `(test! expression value)`, which determines (a) whether *expression* can be successfully evaluated and, if so, (b) whether the value of *expression* is *value*.
- `(test-error! expression)`, which determines whether or not *expression* reports an error (and treats it as a problem if *expression* does not).
- `(end-tests!)`, which reports on the tests completed so far.

a. Create a new DrScheme window that will permit you to use `unit-test.ss`.

b. In the Definitions pane, add the following line and click Run.

```
(load "/home/rebelsky/Web/Courses/CS151/2006F/Examples/unit-test.ss")
```

c. In the Interactions pane, try each of the operations a few times to make sure you understand its operation. (Yes, this instruction is intentionally vague.)

d. As you should have observed from your interactions, `test!` and `test-error!` give no output. Why do you think the designers made that decision?

e. What happens if you call `end-tests!` multiple times, perhaps with some intervening calls to `test!` or `test-error!` but with no intervening calls to `begin-tests!`?

## Exercise 2: Testing Triangulation

Consider the following procedure documentation:

```
;;; Procedure:
;;;   classify-triangle
;;; Parameters:
;;;   side1, a real number [unverified]
;;;   side2, a real number [unverified]
;;;   side3, a real number [unverified]
;;; Purpose:
;;;   Determine the kind of triangle the three sides describe.
;;; Produces:
;;;   classification, a symbol
;;; Preconditions:
;;;   side1, side2, and side3 together describe a triangle [verified]
;;; Postconditions:
;;;   If all three sides are equal, classification is 'equilateral.
;;;   If exactly two sides are equal, classification is 'isoceles.
;;;   If no two sides are equal, classification is 'scalene.
```

a. Write a series of tests for this procedure. Your tests should look something like the following:

```
(load "/home/rebelsky/Web/Courses/CS151/2006F/Examples/unit-test.ss")
(load "/home/rebelsky/Web/Courses/CS151/2006F/Examples/Triangles/tri.0000.ss")
(begin-tests!)
(test! (classify-triangle 1 1 1) 'equilateral)
...
(end-tests!)
```

Remember that a goal of testing is that you should be confident that a procedure that passes all of tests is correct.

*Spend no more than five minutes writing these tests!* (Of course, given that limited time, you don't need full confidence that anything that passes is correct.)

b. I've created approximately 40 different versions of the procedure of varying degrees of correctness (yes, I wrote a program to do so). They are named as follows (and all appear in the same directory as `tri.0000.ss`): `tri.0000.ss`, `tri.0001.ss`, `tri.0010.ss`, `tri.0011.ss`, `tri.0100.ss`, `tri.0101.ss`, `tri.0110.ss`, `tri.0111.ss`, `tri.0200.ss`, `tri.0201.ss`, `tri.0210.ss`, `tri.0211.ss`, `tri.1000.ss`, `tri.1001.ss`, `tri.1010.ss`, `tri.1011.ss`, `tri.1100.ss`, `tri.1101.ss`, `tri.1110.ss`, `tri.1111.ss`, `tri.1200.ss`, `tri.1201.ss`, `tri.1210.ss`, `tri.1211.ss`, `tri.2000.ss`, `tri.2001.ss`, `tri.2010.ss`, `tri.2011.ss`, `tri.2100.ss`, `tri.2101.ss`, `tri.2110.ss`, `tri.2111.ss`, `tri.2200.ss`, `tri.2201.ss`, `tri.2210.ss`, and `tri.2211.ss`

Pick five or so of those files and run your tests on them. Which ones fail your tests?

### Exercise 3: Writing a Triangle Classifier

- Open a new DrScheme window.
- To the best of your ability, write the `classify-triangle` procedure described in the previous problem.
- Save your version in a separate file (e.g., `mytri.ss`).
- Rerun your tests using this version of the procedure. How many does it pass, how many does it fail? Do not rewrite the procedure to make it pass all these tests.

### Exercise 4: A Simple Test Suite

Here is a test that someone might write for the procedure described above.

```
(load "/home/rebelsky/Web/Courses/CS151/2006F/Examples/unit-test.ss")
(load "/home/rebelsky/Web/Courses/CS151/2006F/Examples/Triangles/tri.0000.ss")
(begin-tests!)
(test! (classify-triangle 1 1 1) 'equilateral)
(test! (classify-triangle 2 2 3) 'isocoles)
(test! (classify-triangle 3 4 5) 'scalene)
(end-tests!)
```

- Does your `classify-triangle` procedure pass these tests? If not, repair it to ensure that it passes all the tests.
- Run this test suite on five of the variants mentioned in problem 2 that you have either not tested or that have passed all tests so far. How many pass the new test suite?
- Here is an “obvious” solution to `classify-triangle` that passes all of these tests, but doesn’t meet the specifications. What other tests would help us determine that it doesn’t meet the specifications.

```
(define classify-triangle
  (lambda (side1 side2 side3)
    (cond
      ((and (eq? side1 side2) (eq? side2 side3)) 'equilateral)
      ((eq? side1 side2) 'isocoles)
      (else 'scalene))))
```

### Exercise 5: Testing Error Checking

Someone thinking carefully about the definition of `classify-triangle` might worry that the tests, as written, do not check for non-triangles. For example, something with side lengths 1, 1, and 3 is not a triangle.

- a. Add a test to the testing code above to ensure that `classify-triangle` rejects that non-triangle.
- b. Does your version of `classify-triangle` pass the revised test suite? If not, correct it so that it does.
- c. Run this test suite on five of the variants mentioned in problem 2 that you have either not tested or that have passed all tests so far. How many pass the new test suite?

## Exercise 6: Testing Error Checking, Revisited

Another set of inputs for which `classify-triangle` is supposed to fail is one in which any of the side lengths are negative.

- a. Add tests to the test suite for `classify-triangle` to ensure that `classify-triangle` rejects any set of three numbers that include a zero or negative number.
- b. Does your version of `classify-triangle` pass the revised test suite? If not, correct it so that it does.
- c. Run this test suite on five of the variants mentioned in problem 2 that you have either not tested or that have passed all tests so far. How many pass the new test suite?

## Exercise 7: Symmetric Tests

We've tested for each of the three kinds of triangles, but we've only one test for each. What if the programmer mistakenly forgets to deal with the different orderings of parameters? We should make sure that the implementation works for each.

- a. Add tests to the test suite for `classify-triangle` to ensure that `classify-triangle` correctly identifies the three kinds of triangles, not matter what the ordering. In particular, make sure that you test for the three variants of isoceles triangles.

```
(test! (classify-triangle 2 2 3) 'isocetes)
(test! (classify-triangle 2 3 2) 'isocetes)
(test! (classify-triangle 3 2 2) 'isocetes)
```

- b. Are there other tests in which the ordering might matter? (Hint: Think about some of the tests for errors.)
- c. Does your version of `classify-triangle` pass the revised test suite? If not, correct it so that it does.
- d. Run this test suite on five of the variants mentioned in problem 2 that you have either not tested or that have passed all tests so far. How many pass the new test suite?

## Exercise 8: Testing Large Sides

We now might consider the range of side values for which `classify-triangle` should be tested. In the past, we've seen that `eq?` doesn't consider large numbers equal, even when `=` and `equiv?` do. So, let's add some tests.

- a. Add tests that ensure that `classify-triangle` works for very large numbers (say numbers greater than 10 billion).
- b. Does your version of `classify-triangle` pass the revised test suite? If not, correct it so that it does.
- c. Run this test suite on five of the variants mentioned in problem 2 that you have either not tested or that have passed all tests so far. How many pass the new test suite?

## Exercise 9: Testing Small Edges

Just as we should check large numbers, so should we check small numbers. (In part, because I know that some of us make mistakes near boundaries.)

- a. Add some tests that use positive values less than 1. Here are some examples:

```
(test! (classify-triangle 1/2 1/2 1/2) 'equilateral)
(test! (classify-triangle 2/5 2/5 3/5) 'isocoles)
(test! (classify-triangle 3/7 4/7 5/7) 'scalene)
(test-error! (classify-triangle 1/6 1/6 1/2))
```

- b. Does your version of `classify-triangle` pass the revised test suite? If not, correct it so that it does.
- c. Run this test suite on five of the variants mentioned in problem 2 that you have either not tested or that have passed all tests so far. How many pass the new test suite?

## Exercise 10: Other Tests

- a. Add any other tests you conceive of (and be prepared to discuss those tests in class).
- b. Does your version of `classify-triangle` pass the revised test suite? If not, correct it so that it does.
- c. Run these tests on any variants mentioned in problem 2 that you have not yet tested or that have passed all the tests up through exercise 9. How many pass the tests?

---

Copyright © 2006 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.