

Unit Testing

Summary: As you develop procedures (and collections of procedures), you have a responsibility to make sure that they work correctly. One mechanism for checking your procedure is a comprehensive suite of tests. In this reading, we consider the design and use of tests.

Contents:

- Introduction
- What is a Test?
- Testing Environments
- When To Write Tests
- An Example: Testing `tally`
- Appendix: A Historical Tale

Introduction

Most computer programmers strive to write clear, efficient, and correct code. It is (usually) easy to determine whether or not code is clear. With some practice and knowledge of the correct tools, one can determine how efficient code is. However, it is often difficult to determine whether or not code is correct.

The gold standard of correctness is, of course, a formal proof that the procedure or program is correct. However, in order to prove a program or procedure correct, one must develop a rich mathematical toolkit and devote significant effort to writing the proof. Such effort is worth it for life-critical applications, but for many programs, it is often more than can be reasonably expected.

There is also a disadvantage of formal proof: Code often changes and the proof must therefore also change. Why does code change? At times, the requirements of the code change (e.g., a procedure that was to do three related things is now expected to do four related things). At other times, with experience, programmers realize that they can improve the code by making a few changes. If we require that all code be proven correct, and if changing code means rewriting the proof, then we discourage programmers from changing their code.

Hence, we need other ways to have some confidence that our code is correct. A typical mechanism is a *test suite*, a collection of tests that are unlikely to all succeed if the code being tested is erroneous. One nice aspect of a test suite is that when you make changes, you can simply re-run the test suite and see if all the tests succeed. To many programmers, test suites encourage programmers to experiment with improving their code, since good suites will tell them immediately whether or not the changes they have made are successful.

But when and how do you develop tests? These questions are the subject of this reading.

What is a Test?

So, you should write tests when you write code. But what is a test? Put simply, a test is a bit of code that reveals something about the correctness of a procedure (or a set of procedures). Most typically, we express tests in terms of expressions and their expected values. For example, if we've written a procedure, `rev`, that reverses lists, we would expect that

- `(rev null)` is `null`
- `(rev (list 3))` is `(list 3)`
- `(rev (list 3 7 11))` is `(list 11 7 3)`

We could express those expectations in a variety of ways. The simplest strategy is to execute each expression, in turn, and see if the result is what we expected. You should have used this form of testing regularly in your coding.

```
> (rev null)
null
> (rev (list 3))
(3)
> (rev (list 3 7 11))
(11 7 3)
```

Of course, one disadvantage of this kind of testing is that you have to manually look at the results to make sure that they are correct. There's some evidence that you don't always catch errors when you have to do this comparison, particularly when you have a lot of tests. I know that I've certainly missed a number of errors this way. An appendix to this document presents an interesting historical anecdote about the dangers of writing a test suite in which you must manually read all of the results.

Since reading the results is tedious and dangerous, it is often useful to have the computer do the comparison for you. For example, we might write a procedure, `check`, that checks to make sure that two expressions are equal.

```
(define check
  (lambda (exp1 exp2)
    (if (equal? exp1 exp2)
        "OK"
        "FAILED")))
```

We can then use this procedure for the tests above, as follows.

```
> (check (rev null) null)
"OK"
> (check (rev (list 3)) (list 3))
"OK"
> (check (rev (list 3 7 11)) (list 11 7 3))
"OK"
> (check (rev (list 3 7 11)) (list 11 'd 3))
"FAILED"
```

(Note that in this case, the last test was incorrect.)

Now, confirming that our code is correct is now simply a matter of scanning through the results and seeing if any say "FAILED".

Of course, there are still some disadvantages with this strategy. For example, if we put the tests in a file to execute one by one, it may be difficult to tell which ones failed. Also, for a large set of tests, it seems a bit excessive to print OK every time. Finally, we get neither OK nor FAILED when there's an error in the original expression.

```
> (check (rev 5) 5)
Error: Cannot take the car of a non-pair.
```

Testing Environments

To handle all of these additional issues, many program development environments now include some form of testing environment, in which you specify a sequence of tests and receive quick summaries of the results of the tests. For example, a testing environment, given the four tests for the previous section, might report something like:

```
4 tests conducted.
One test failed.
No errors encountered.
One other test failed to give the expected result:
  For [(rev (list 3 7 11))], Expected [(11 7 3)], got [(11 3 7)].
Sorry. You'll need to fix your code.
```

Different testing environments are designed differently. For this class, we have developed a simple testing environment that emphasizes testing for expected outcomes. That environment is loaded automatically into DrFu.

To use that environment, you begin a series of tests with the `begin-tests!` procedure. That procedure takes no parameters. When you are done with a sequence of tests, you invoke the `end-tests!` procedure. That procedure reports on the results of the tests.

The tests themselves might seem a bit strange at first glance. For each test, you call the `test!` procedure with two parameters, (1) the expression you want to evaluate and (2) an expression that gives the value you expect. For example, we would express the first three (correct) tests of `rev` as follows.

```
(test! (rev null) null)
(test! (rev (list 3)) (list 3))
(test! (rev (list 3 7 11)) (list 11 7 3))
```

We might also want to test whether an expression causes an error. For example, we would expect `(my-reverse 5)` to give an error because 5 is a number and not a list. The `test-error!` procedure lets us check whether an expression we expect to give an error actually gives that error.

```
(test-error! (my-reverse 5))
```

When To Write Tests

To many programmers, testing is much like documentation. That is, it's something you add after you've written the majority of the code. However testing, like documentation, can make it much easier to write the code in the first place.

As we suggested in the reading on documentation, by writing documentation first, you develop a clearer sense of what you want your procedures to accomplish. Taking the time to write documentation can also help you think through the special cases. For some programmers, writing the formal postconditions can give them an idea of how to solve the problem.

If you design your tests first, you can accomplish similar goals. For example, if you think carefully about what tests might fail, you make sure the special cases are handled. Also, a good set of tests of the form "this expression should have this value", can serve as a form of documentation for the reader, explaining through example what the procedure is to do. There is even a style of software engineering, called test-driven development, in which you write tests first.

An Example: Testing `tally`

Let's consider this test-first strategy as we attempt to write a common procedure, `(tally val lst)`, which counts the number of times that `val` appears in `lst`. What are some good tests?

- We should make sure that `tally` returns 0 for the empty list.
- We should make sure that `tally` returns 1 for a singleton list in which the singleton element is `val`.
- We should make sure that `tally` returns 0 for a singleton list in which the singleton element is not `val`.
- We should make sure that `tally` returns 1 for length-three lists in which `val` is just the first, just the last, or just the middle element.
- We should make sure that `tally` returns the appropriate count for lists that include multiple copies of `val`.
- We should make sure that `tally` returns 0 for longer lists that do not include `val`.
- We might try different types of values.
- You can probably fill in some more of your own.

So, we create two files, `tally.sct`, which will contain the code and documentation for `tally`, and `tally-test.sct`, which will contain our testing code. We create `tally-test.sct` first.

```
(load "tally.sct")
(begin-tests!)
(test! (tally 5 null) 0)
(test! (tally 5 (list 5)) 1)
(test! (tally 5 (list 17)) 0)
(test! (tally 5 (list 5 17 6)) 1)
(test! (tally 5 (list 17 5 6)) 1)
(test! (tally 5 (list 6 17 5)) 1)
(test! (tally 5 (list 5 5 5 5 5)) 5)
```

```
(test! (tally 5 (list 5 17 5 17 5)) 3)
(test! (tally 5 (list 17 5 5 17 17)) 2)
(test! (tally 5 (list 17 17 17 17 17)) 0)
(end-tests!)
```

Of course, we haven't defined `tally` yet, so we don't expect the tests to work, but let's see what happens.

```
10 tests conducted.
10 tests failed.
10 errors encountered
The expression (tally 5 null) failed to evaluate because [reference to undefined identifier: tally]
The expression (tally 5 (list 5)) failed to evaluate because [reference to undefined identifier: tally]
The expression (tally 5 (list 17)) failed to evaluate because [reference to undefined identifier: tally]
The expression (tally 5 (list 5 17 6)) failed to evaluate because [reference to undefined identifier: tally]
The expression (tally 5 (list 17 5 6)) failed to evaluate because [reference to undefined identifier: tally]
The expression (tally 5 (list 6 17 5)) failed to evaluate because [reference to undefined identifier: tally]
The expression (tally 5 (list 5 5 5 5 5)) failed to evaluate because [reference to undefined identifier: tally]
The expression (tally 5 (list 5 17 5 17 5)) failed to evaluate because [reference to undefined identifier: tally]
The expression (tally 5 (list 17 5 5 17 17)) failed to evaluate because [reference to undefined identifier: tally]
The expression (tally 5 (list 17 17 17 17 17)) failed to evaluate because [reference to undefined identifier: tally]
No other tests failed to give the expected result.
Sorry. You'll need to fix your code.
```

Yeah, we'd expect some errors. So, let's start to define `tally`. Here's an incorrect definition:

```
(define tally
  (lambda (val lst)
    (if (= val (car lst))
        1
        0)))
```

Amazingly, this passes several of our tests. When we run the tests, here is the output.

```
10 tests conducted.
6 tests failed.
One error encountered.
The expression (tally 5 null) failed to evaluate because [car: expects argument of type <pair>; given ()]
5 other tests failed to give the expected result.
For (tally 5 (list 17 5 6)) expected [1] got [0]
For (tally 5 (list 6 17 5)) expected [1] got [0]
For (tally 5 (list 5 5 5 5 5)) expected [5] got [1]
For (tally 5 (list 5 17 5 17 5)) expected [3] got [1]
For (tally 5 (list 17 5 5 17 17)) expected [2] got [0]
Sorry. You'll need to fix your code.
```

That first error is important. It indicates that we're taking the `car` of an empty list. The others are probably a case of us forgetting to recurse. Let's try again

```
(define tally
  (lambda (val lst)
    (if (null? lst)
        0
        (+ (if (= (car lst) val) 1 0)
           (tally val (cdr lst))))))
```

What does our test suite say?

```
10 tests conducted.
No errors encountered.
No tests failed.
CONGRATULATIONS! All tests passed.
```

Wow. That's comforting. We seem to have written it correctly (or at least correctly enough to pass our tests). However, some people might find the formulation above confusing or inelegant, so we might try to rewrite it. Instead of putting the addition within the if, we'll have a three-way cond.

```
(define tally
  (lambda (val lst)
    (cond
      ((null? lst) 0)
      ((= val (car lst)) 1)
      (else (tally val (cdr lst))))))
```

What do the tests say?

```
10 tests conducted.
3 tests failed
No errors encountered.
3 other tests failed to get the expected result:
  For (tally 5 (list 5 5 5 5 5)) expected [5] got [1]
  For (tally 5 (list 5 17 5 17 5)) expected [3] got [1]
  For (tally 5 (list 17 5 5 17 17)) expected [2] got [1]
Sorry. You'll need to fix your code.
```

Ah! We've made a common mistake of reorganizing code: We've kept the recursive call in one place, but we need it in more than one place.

```
(define tally
  (lambda (val lst)
    (cond
      ((null? lst) 0)
      ((= val (car lst)) (+ 1 (tally val (cdr lst))))
      (else (tally val (cdr lst))))))
```

We cross our fingers and run the tests again.

```
10 tests conducted.
No tests failed.
CONGRATULATIONS! All tests passed.
```

Now, we've left out some tests. What if *val* is a string, a spot, or even a list? We'll add a few more tests to `tally-test.sct` to cover these cases. We don't need as many tests for each, since we already know that the overall structure works.

```

(test! (tally "hello" (list "hello" "goodbye" "and" "hello"))) 2)
(test! (tally (spot.new 0 0 color.white)
              (list (spot.new 0 0 color.white) (spot.new 0 0 color.yellow)))
      1)
(test! (tally (spot.new 0 0 color.white)
              (list (spot.new 1 1 color.white)
                    (spot.new 0 1 color.white)
                    (spot.new 0 0 color.yellow))))
      1)
(test! (tally null (list null "null" 0)) 1)

```

As you'll see in the laboratory, these tests will review fascinating new errors in the code. The lab will give you the opportunity to correct the errors.

Appendix: A Historical Tale

Many of us are reminded of the need for unit testing by the following story by Doug McIlroy, posted to *The Risks Digest: Forum on Risks to the Public in Computers and Related Systems*.

Sometime around 1961, a customer of the Bell Labs computing center questioned a value returned by the sine routine. The cause was simple: a card had dropped out of the assembly language source. Bob Morris pinned down the exact date by checking the dutifully filed reversion tests for system builds. Each time test values of the sine routine (and the rest of the library) had been printed out. Essentially the acceptance criterion was that the printout was the right thickness; the important point was that the tests ran to conclusion, not that they gave right answers. The trouble persisted through several deployed generations of the system.

McIlroy, Doug (2006). Trig routine risk: An Oldie. *Risks Digest* **24**(49), December 2006.

If, instead of a thick printout, Bell Labs had arranged for a count of successes and a list of failures, they (and their customers) would have been in much better shape.

Copyright © 2007 Janet Davis, Matthew Kluber, and Samuel A. Rebelsky. (Selected materials copyright by John David Stone and Henry Walker and used by permission.) This material is based upon work partially supported by the National Science Foundation under Grant No. CCLI-0633090. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. 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.