9. Practical: Building a Unit Test Framework

In this chapter you'll return to cutting code and develop a simple unit testing framework for Lisp. This will give you a chance to use some of the features you've learned about since Chapter 3, including macros and dynamic variables, in real code.

The main design goal of the test framework will be to make it as easy as possible to add new tests, to run various suites of tests, and to track down test failures. For now you'll focus on designing a framework you can use during interactive development.

The key feature of an automated testing framework is that the framework is responsible for telling you whether all the tests passed. You don't want to spend your time slogging through test output checking answers when the computer can do it much more quickly and accurately. Consequently, each test case must be an expression that yields a boolean value--true or false, pass or fail. For instance, if you were writing tests for the built-in **+** function, these might be reasonable test cases:[1]

```
(= (+ 1 2) 3)
(= (+ 1 2 3) 6)
(= (+ -1 -3) -4)
```

Functions that have side effects will be tested slightly differently--you'll have to call the function and then check for evidence of the expected side effects.[2] But in the end, every test case has to boil down to a boolean expression, thumbs up or thumbs down.

Two First Tries

If you were doing ad hoc testing, you could enter these expressions at the REPL and check that they return **`T`**. But you want a framework that makes it easy to organize and run these test cases whenever you want. If you want to start with the simplest thing that could possibly work, you can just write a function that evaluates the test cases and **`AND`**s the results together.

```
(defun test-+ ()
  (and
    (= (+ 1 2) 3)
    (= (+ 1 2 3) 6)
    (= (+ -1 -3) -4)))
```

Whenever you want to run this set of test cases, you can call `test-+`.

```
CL-USER> (test-+)
T
```

As long as it returns **T**, you know the test cases are passing. This way of organizing tests is also pleasantly concise--you don't have to write a bunch of test bookkeeping code. However, as you'll discover the first time a test case fails, the result reporting leaves something to be desired. When `test-+` returns **NIL**, you'll know something failed, but you'll have no idea which test case it was.

So let's try another simple--even simpleminded--approach. To find out what happens to each test case, you could write something like this:

```
(defun test-+ ()
  (format t "~:[FAIL~;pass~] ... ~a~%" (= (+ 1 2) 3) '(= (+ 1 2) 3))
  (format t "~:[FAIL~;pass~] ... ~a~%" (= (+ 1 2 3) 6) '(= (+ 1 2 3) 6))
  (format t "~:[FAIL~;pass~] ... ~a~%" (= (+ -1 -3) -4) '(= (+ -1 -3) -4)))
```

Now each test case will be reported individually. The `~:[FAIL~;pass~]` part of the **FORMAT** directive causes **FORMAT** to print "FAIL" if the first format argument is false and "pass" otherwise.[3] Then you label the result with the test expression itself. Now running `test-+` shows you exactly what's going on.

```
CL-USER> (test-+)
pass ... (= (+ 1 2) 3)
pass ... (= (+ 1 2 3) 6)
pass ... (= (+ -1 -3) -4)
NIL
```

This time the result reporting is more like what you want, but the code itself is pretty gross. The repeated calls to **FORMAT** as well as the tedious duplication of the test expression cry out to be refactored. The duplication of the test expression is particularly grating because if you mistype it, the test results will be mislabeled.

Another problem is that you don't get a single indicator whether all the test cases passed. It's easy enough, with only three test cases, to scan the output looking for "FAIL"; however, when you have hundreds of test cases, it'll be more of a hassle.

Refactoring

What you'd really like is a way to write test functions as streamlined as the first `test-+` that return a single **T** or **NIL** value but that also report on the results of individual test cases like the second version. Since the second version is close to what you want in terms of functionality, your best bet is to see if you can factor out some of the annoying duplication.

The simplest way to get rid of the repeated similar calls to **FORMAT** is to create a new function.

```
(defun report-result (result form)
  (format t "~:[FAIL~;pass~] ... ~a~%" result form))
```

Now you can write `test-+` with calls to `report-result` instead of **`FORMAT`**. It's not a huge improvement, but at least now if you decide to change the way you report results, there's only one place you have to change.

```
(defun test-+ ()
  (report-result (= (+ 1 2) 3) '(= (+ 1 2) 3))
  (report-result (= (+ 1 2 3) 6) '(= (+ 1 2 3) 6))
  (report-result (= (+ -1 -3) -4) '(= (+ -1 -3) -4)))
```

Next you need to get rid of the duplication of the test case expression, with its attendant risk of mislabeling of results. What you'd really like is to be able to treat the expression as both code (to get the result) and data (to use as the label). Whenever you want to treat code as data, that's a sure sign you need a macro. Or, to look at it another way, what you need is a way to automate writing the error-prone `report-result` calls. You'd like to be able to say something like this:

```
(check (= (+ 1 2) 3))
```

and have it mean the following:

```
(report-result (= (+ 1 2) 3) '(= (+ 1 2) 3))
```

Writing a macro to do this translation is trivial.

```
(defmacro check (form)
  `(report-result ,form ',form))
```

Now you can change `test-+` to use `check`.

```
(defun test-+ ()
  (check (= (+ 1 2) 3))
  (check (= (+ 1 2 3) 6))
  (check (= (+ -1 -3) -4)))
```

Since you're on the hunt for duplication, why not get rid of those repeated calls to `check`? You can define `check` to take an arbitrary number of forms and wrap them each in a call to `report-result`.

```
(defmacro check (&body forms)
  `(progn
     ,@(loop for f in forms collect `(report-result ,f ',f))))
```

This definition uses a common macro idiom of wrapping a **`PROGN`** around a series of forms in order to turn them into a single form. Notice also how you can use `,@` to splice in the result of an expression that returns a list of expressions that are themselves generated with a backquote template.

With the new version of `check` you can write a new version of `test-+` like this:

```
(defun test-+ ()
  (check
    (= (+ 1 2) 3)
    (= (+ 1 2 3) 6)
    (= (+ -1 -3) -4)))
```

that is equivalent to the following code:

```
(defun test-+ ()
  (progn
    (report-result (= (+ 1 2) 3) '(= (+ 1 2) 3))
    (report-result (= (+ 1 2 3) 6) '(= (+ 1 2 3) 6))
    (report-result (= (+ -1 -3) -4) '(= (+ -1 -3) -4))))
```

Thanks to `check`, this version is as concise as the first version of `test-+` but expands into code that does the same thing as the second version. And now any changes you want to make to how `test-+` behaves, you can make by changing `check`.

Fixing the Return Value

You can start with fixing `test-+` so its return value indicates whether all the test cases passed. Since `check` is responsible for generating the code that ultimately runs the test cases, you just need to change it to generate code that also keeps track of the results.

As a first step, you can make a small change to `report-result` so it returns the result of the test case it's reporting.

```
(defun report-result (result form)
  (format t "~:[FAIL~;pass~] ... ~a~%" result form)
  result)
```

Now that `report-result` returns the result of its test case, it might seem you could just change the **`PROGN`** to an **`AND`** to combine the results. Unfortunately, **`AND`** doesn't do quite what you want in this case because of its short-circuiting behavior: as soon as one test case fails, **`AND`** will skip the rest. On the other hand, if you had a construct that worked like **`AND`** without the short-circuiting, you could use it in the place of **`PROGN`**, and you'd be done. Common Lisp doesn't provide such a construct, but that's no reason you can't use it: it's a trivial matter to write a macro to provide it yourself.

Leaving test cases aside for a moment, what you want is a macro--let's call it `combine-results`--that will let you say this:

```
(combine-results
  (foo)
  (bar)
  (baz))
```

and have it mean something like this:

```
(let ((result t))
  (unless (foo) (setf result nil))
  (unless (bar) (setf result nil))
  (unless (baz) (setf result nil))
  result)
```

The only tricky bit to writing this macro is that you need to introduce a variable--`result` in the previous code--in the expansion. As you saw in the previous chapter, using a literal name for

variables in macro expansions can introduce a leak in your macro abstraction, so you'll need to create a unique name. This is a job for `with-gensyms`. You can define `combine-results` like this:

```
(defmacro combine-results (&body forms)
  (with-gensyms (result)
    `(let ((,result t))
       ,@(loop for f in forms collect `(unless ,f (setf ,result nil)))
       ,result)))
```

Now you can fix `check` by simply changing the expansion to use `combine-results` instead of **`PROGN`**.

```
(defmacro check (&body forms)
  `(combine-results
    ,@(loop for f in forms collect `(report-result ,f ',f))))
```

With that version of `check`, `test-+` should emit the results of its three test expressions and then return **`T`** to indicate that everything passed.[4]

```
CL-USER> (test-+)
pass ... (= (+ 1 2) 3)
pass ... (= (+ 1 2 3) 6)
pass ... (= (+ -1 -3) -4)
T
```

And if you change one of the test cases so it fails,[5] the final return value changes to **`NIL`**.

```
CL-USER> (test-+)
pass ... (= (+ 1 2) 3)
pass ... (= (+ 1 2 3) 6)
FAIL ... (= (+ -1 -3) -5)
NIL
```

Better Result Reporting

As long as you have only one test function, the current result reporting is pretty clear. If a particular test case fails, all you have to do is find the test case in the `check` form and figure out why it's failing. But if you write a lot of tests, you'll probably want to organize them somehow, rather than shoving them all into one function. For instance, suppose you wanted to add some test cases for the `*` function. You might write a new test function.

```
(defun test-* ()
  (check
    (= (* 2 2) 4)
    (= (* 3 5) 15)))
```

Now that you have two test functions, you'll probably want another function that runs all the tests. That's easy enough.

```
(defun test-arithmetic ()
  (combine-results
   (test-+)
   (test-*)))
```

In this function you use `combine-results` instead of `check` since both `test-+` and `test-*` will take care of reporting their own results. When you run `test-arithmetic`, you'll get the following results:

```
CL-USER> (test-arithmetic)
pass ... (= (+ 1 2) 3)
pass ... (= (+ 1 2 3) 6)
pass ... (= (+ -1 -3) -4)
pass ... (= (* 2 2) 4)
pass ... (= (* 3 5) 15)
T
```

Now imagine that one of the test cases failed and you need to track down the problem. With only five test cases and two test functions, it won't be too hard to find the code of the failing test case. But suppose you had 500 test cases spread across 20 functions. It might be nice if the results told you what function each test case came from.

Since the code that prints the results is centralized in `report-result`, you need a way to pass information about what test function you're in to `report-result`. You could add a parameter to `report-result` to pass this information, but `check`, which generates the calls to `report-result`, doesn't know what function it's being called from, which means you'd also have to change the way you call `check`, passing it an argument that it simply passes onto `report-result`.

This is exactly the kind of problem dynamic variables were designed to solve. If you create a dynamic variable that each test function binds to the name of the function before calling `check`, then `report-result` can use it without `check` having to know anything about it.

Step one is to declare the variable at the top level.

```
(defvar *test-name* nil)
```

Now you need to make another tiny change to `report-result` to include `*test-name*` in the **`FORMAT`** output.

```
(format t "~:[FAIL~;pass~] ... ~a: ~a~%" result *test-name* form)
```

With those changes, the test functions will still work but will produce the following output because `*test-name*` is never rebound:

```
CL-USER> (test-arithmetic)
pass ... NIL: (= (+ 1 2) 3)
pass ... NIL: (= (+ 1 2 3) 6)
pass ... NIL: (= (+ -1 -3) -4)
pass ... NIL: (= (* 2 2) 4)
pass ... NIL: (= (* 3 5) 15)
T
```

For the name to be reported properly, you need to change the two test functions.

```
(defun test-+ ()
  (let ((*test-name* 'test-+))
```

```
    (check
      (= (+ 1 2) 3)
      (= (+ 1 2 3) 6)
      (= (+ -1 -3) -4))))

(defun test-* ()
  (let ((*test-name* 'test-*))
    (check
      (= (* 2 2) 4)
      (= (* 3 5) 15))))
```

Now the results are properly labeled.

```
CL-USER> (test-arithmetic)
pass ... TEST-+: (= (+ 1 2) 3)
pass ... TEST-+: (= (+ 1 2 3) 6)
pass ... TEST-+: (= (+ -1 -3) -4)
pass ... TEST-*: (= (* 2 2) 4)
pass ... TEST-*: (= (* 3 5) 15)
T
```

An Abstraction Emerges

In fixing the test functions, you've introduced several new bits of duplication. Not only does each function have to include the name of the function twice--once as the name in the **`DEFUN`** and once in the binding of `*test-name*`--but the same three-line code pattern is duplicated between the two functions. You could remove the duplication simply on the grounds that duplication is bad. But if you look more closely at the root cause of the duplication, you can learn an important lesson about how to use macros.

The reason both these functions start the same way is because they're both test functions. The duplication arises because, at the moment, *test function* is only half an abstraction. The abstraction exists in your mind, but in the code there's no way to express "this is a test function" other than to write code that follows a particular pattern.

Unfortunately, partial abstractions are a crummy tool for building software. Because a half abstraction is expressed in code by a manifestation of the pattern, you're guaranteed to have massive code duplication with all the normal bad consequences that implies for maintainability. More subtly, because the abstraction exists only in the minds of programmers, there's no mechanism to make sure different programmers (or even the same programmer working at different times) actually understand the abstraction the same way. To make a complete abstraction, you need a way to express "this is a test function" and have all the code required by the pattern be generated for you. In other words, you need a macro.

Because the pattern you're trying to capture is a **`DEFUN`** plus some boilerplate code, you need to write a macro that will expand into a **`DEFUN`**. You'll then use this macro, instead of a plain **`DEFUN`** to define test functions, so it makes sense to call it `deftest`.

```
(defmacro deftest (name parameters &body body)
  `(defun ,name ,parameters
    (let ((*test-name* ',name))
      ,@body)))
```

With this macro you can rewrite `test-+` as follows:

```
(deftest test-+ ()
  (check
    (= (+ 1 2) 3)
    (= (+ 1 2 3) 6)
    (= (+ -1 -3) -4)))
```

A Hierarchy of Tests

Now that you've established test functions as first-class citizens, the question might arise, should `test-arithmetic` be a test function? As things stand, it doesn't really matter--if you did define it with `deftest`, its binding of `*test-name*` would be shadowed by the bindings in `test-+` and `test-*` before any results are reported.

But now imagine you've got thousands of test cases to organize. The first level of organization is provided by test functions such as `test-+` and `test-*` that directly call `check`. But with thousands of test cases, you'll likely need other levels of organization. Functions such as `test-arithmetic` can group related test functions into test suites. Now suppose some low-level test functions are called from multiple test suites. It's not unheard of for a test case to pass in one context but fail in another. If that happens, you'll probably want to know more than just what low-level test function contains the test case.

If you define the test suite functions such as `test-arithmetic` with `deftest` and make a small change to the `*test-name*` bookkeeping, you can have results reported with a "fully qualified" path to the test case, something like this:

```
pass ... (TEST-ARITHMETIC TEST-+): (= (+ 1 2) 3)
```

Because you've already abstracted the process of defining a test function, you can change the bookkeeping details without modifying the code of the test functions.[6] To make `*test-name*` hold a list of test function names instead of just the name of the most recently entered test function, you just need to change this binding form:

```
(let ((*test-name* ',name))
```

to the following:

```
(let ((*test-name* (append *test-name* (list ',name))))
```

Since **`APPEND`** returns a new list made up of the elements of its arguments, this version will bind `*test-name*` to a list containing the old contents of `*test-name*` with the new name tacked onto the end.[7] When each test function returns, the old value of `*test-name*` will be restored.

Now you can redefine `test-arithmetic` with `deftest` instead of **`DEFUN`**.

```
(deftest test-arithmetic ()
  (combine-results
   (test-+)
   (test-*)))
```

The results now show exactly how you got to each test expression.

```
CL-USER> (test-arithmetic)
pass ... (TEST-ARITHMETIC TEST-+): (= (+ 1 2) 3)
pass ... (TEST-ARITHMETIC TEST-+): (= (+ 1 2 3) 6)
pass ... (TEST-ARITHMETIC TEST-+): (= (+ -1 -3) -4)
pass ... (TEST-ARITHMETIC TEST-*): (= (* 2 2) 4)
pass ... (TEST-ARITHMETIC TEST-*): (= (* 3 5) 15)
T
```

As your test suite grows, you can add new layers of test functions; as long as they're defined with `deftest`, the results will be reported correctly. For instance, the following:

```
(deftest test-math ()
  (test-arithmetic))
```

would generate these results:

```
CL-USER> (test-math)
pass ... (TEST-MATH TEST-ARITHMETIC TEST-+): (= (+ 1 2) 3)
pass ... (TEST-MATH TEST-ARITHMETIC TEST-+): (= (+ 1 2 3) 6)
pass ... (TEST-MATH TEST-ARITHMETIC TEST-+): (= (+ -1 -3) -4)
pass ... (TEST-MATH TEST-ARITHMETIC TEST-*): (= (* 2 2) 4)
pass ... (TEST-MATH TEST-ARITHMETIC TEST-*): (= (* 3 5) 15)
T
```

Wrapping Up

You could keep going, adding more features to this test framework. But as a framework for writing tests with a minimum of busywork and easily running them from the REPL, this is a reasonable start. Here's the complete code, all 26 lines of it:

```
(defvar *test-name* nil)

(defmacro deftest (name parameters &body body)
  "Define a test function. Within a test function we can call
   other test functions or use 'check' to run individual test
   cases."
  `(defun ,name ,parameters
     (let ((*test-name* (append *test-name* (list ',name))))
       ,@body)))

(defmacro check (&body forms)
  "Run each expression in 'forms' as a test case."
  `(combine-results
     ,@(loop for f in forms collect `(report-result ,f ',f))))

(defmacro combine-results (&body forms)
  "Combine the results (as booleans) of evaluating 'forms' in order."
  (with-gensyms (result)
    `(let ((,result t))
       ,@(loop for f in forms collect `(unless ,f (setf ,result nil)))
       ,result)))

(defun report-result (result form)
  "Report the results of a single test case. Called by 'check'."
```

```
    (format t "~:[FAIL~;pass~] ... ~a: ~a~%" result *test-name* form)
    result)
```

It's worth reviewing how you got here because it's illustrative of how programming in Lisp often goes.

You started by defining a simple version of your problem--how to evaluate a bunch of boolean expressions and find out if they all returned true. Just **AND**ing them together worked and was syntactically clean but revealed the need for better result reporting. So you wrote some really simpleminded code, chock-full of duplication and error-prone idioms that reported the results the way you wanted.

The next step was to see if you could refactor the second version into something as clean as the former. You started with a standard refactoring technique of extracting some code into a function, `report-result`. Unfortunately, you could see that using `report-result` was going to be tedious and error-prone since you had to pass the test expression twice, once for the value and once as quoted data. So you wrote the `check` macro to automate the details of calling `report-result` correctly.

While writing `check`, you realized as long as you were generating code, you could make a single call to `check` to generate multiple calls to `report-result`, getting you back to a version of `test-+` about as concise as the original **AND** version.

At that point you had the `check` API nailed down, which allowed you to start mucking with how it worked on the inside. The next task was to fix `check` so the code it generated would return a boolean indicating whether all the test cases had passed. Rather than immediately hacking away at `check`, you paused to indulge in a little language design by fantasy. What if--you fantasized--there was already a non-short-circuiting **AND** construct. Then fixing `check` would be trivial. Returning from fantasyland you realized there was no such construct but that you could write one in a few lines. After writing `combine-results`, the fix to `check` was indeed trivial.

At that point all that was left was to make a few more improvements to the way you reported test results. Once you started making changes to the test functions, you realized those functions represented a special category of function that deserved its own abstraction. So you wrote `deftest` to abstract the pattern of code that turns a regular function into a test function.

With `deftest` providing an abstraction barrier between the test definitions and the underlying machinery, you were able to enhance the result reporting without touching the test functions.

Now, with the basics of functions, variables, and macros mastered, and a little practical experience using them, you're ready to start exploring Common Lisp's rich standard library of functions and data types.

---

[1]This is for illustrative purposes only--obviously, writing test cases for built-in functions such as **+** is a bit silly, since if such basic things aren't working, the chances the tests will be running the way you expect is pretty slim. On the other hand, most Common Lisps are implemented largely in Common Lisp, so it's not crazy to imagine writing test suites in Common Lisp to test the standard library functions.

[2]Side effects can include such things as signaling errors; I'll discuss Common Lisp's error handling system in Chapter 19. You may, after reading that chapter, want to think about how to incorporate tests that check whether a function does or does not signal a particular error in certain situations.

[3]I'll discuss this and other **FORMAT** directives in more detail in Chapter 18.

[4]If `test-+` has been compiled--which may happen implicitly in certain Lisp implementations--you may need to reevaluate the definition of `test-+` to get the changed definition of `check` to affect the behavior of `test-+`. Interpreted code, on the other hand, typically expands macros anew each time the code is interpreted, allowing the effects of macro redefinitions to be seen immediately.

[5]You have to change the test to make it fail since you can't change the behavior of **+**.

[6]Though, again, if the test functions have been compiled, you'll have to recompile them after changing the macro.

[7]As you'll see in Chapter 12, **APPEND**ing to the end of a list isn't the most efficient way to build a list. But for now this is sufficient--as long as the test hierarchies aren't too deep, it should be fine. And if it becomes a problem, all you'll have to do is change the definition of `deftest`.