test-driven development in Lisp

brief

The idea behind test-driven development is that you write your tests early-on in the process of building a solution (before you start building a solution is good!) and that you automate the running of these tests. This makes it easy to run & re-run your tests and also gives you a benchmark for determining the success of your solution – if it passes all the tests it works.

There is lots more to be said about the use of/motivation for test-driven development – check other documents.

Due to its functional nature, testers are easy to write in Lisp and many Lisp programmers have their own tester(s) as part of their toolkit.

This document presents a simple tester. Play around with it, tailor it to your needs, make it your own.

a sample problem

Suppose we wish to build a compare function *cmp* which works roughly like *equal* except that it also allows wild-card symbols ("=") to be included in its first argument, so it should work as follows...

```
(cmp '(a b c) '(a b c)) => t
(cmp '(a x c) '(a b c)) => nil
(cmp '(a = c) '(a b c)) => t
```

Lisp allows us to store expressions as if they were data and evaluate them when we want to. Before looking at the cmp function, I will show this working with member...

```
;; remember what member returns...
cg-user(13): (member 'cat '(rat cat mat))
(cat mat)

;; store a call to member in a variable
;; notice (but do not worry about) the nested quotes
cg-user(9): (setf t1 '(member 'cat '(rat cat mat)))
(member 'cat '(rat cat mat))

;; now get the value of t1 using eval
cg-user(10): (eval t1)
(cat mat)
```

```
;; here is another call to member
cg-user(11): (setf t2 '(member 'bat '(rat cat mat)))
(member 'bat '(rat cat mat))
cg-user(12): (eval t2)
nil
```

For convenience I would like to wrap my function calls with the results I expect, and give them a name. I will call these *test rules*, eg:

```
;; a test which should succeed
cg-user(14): (setf t1 '(test1 - (+ 1 2 3) => 6))
(test1 - (+ 1 2 3) => 6)

;; and one which should fail
cg-user(18): (setf t2 '(test2 - (+ 1 2 3) => 164))
(test2 - (+ 1 2 3) => 164)
```

automating tests – step1

The next step is to write some kind of function which takes in data of the shape stored in t1 & t2 above, evaluates the function call & checks the result.

One function is shown below. This returns nil if the test is equal to the result and returns some test-failed message if the result is not as expected. The function is built using defmatch because this makes it easier to deconstruct the test rule.

```
(defmatch run-a-test ((?n - ?call => ?result))
  (unless (equal #?result (eval #?call))
    (pprint (match>> '(failed test. ?n -- ?call)))))

cg-user(15): (setf t1 '(test1 - (+ 1 2 3) => 6))
(test1 - (+ 1 2 3) => 6)
cg-user(16): (run-a-test t1)
nil

cg-user(18): (setf t2 '(test2 - (+ 1 2 3) => 164))
(test2 - (+ 1 2 3) => 164)
cg-user(20): (run-a-test t2)
(failed test. test2 -- (+ 1 2 3))
```

automating tests – step2

Step2 is to build a function which takes a collection of test and runs them all. We will use mapcar for this & return some symbol to indicate that the function has completed.

```
(defun tester (tests)
  (mapcar #'run-a-test tests)
  'test-completed)

;; some tests (including 2 that fail)
(setf tests1 '((test1 - (+ 1 2 3) => 6)
               (test2 - (+ 1 2 3) => 164)
               (m1    - (member 'cat '(rat cat mat)) => (cat mat))
               (m2    - (member 'bat '(rat cat mat)) => nil)
               (m3    - (member 'bat '(rat cat mat)) => banana)
               ))

cg-user(25): (tester tests1)
(failed test. test2 -- (+ 1 2 3))
(failed test. m3 -- (member 'bat '(rat cat mat)))
test-completed
```

developing the cmp function

Now we have a tester we can get back to building the cmp function. The first step is to specify a bunch of tests we want cmp to handle...

```
(defparameter *cmp-tests*
  '((1 - (cmp nil nil) => t)
    (2 - (cmp nil 'x) => nil)
    (3 - (cmp 'x nil) => nil)
    (4 - (cmp '(a b c) '(a b c)) => t)
    (5 - (cmp '(a x c) '(a b c)) => nil)
    (6 - (cmp '(a b c) '(a x c)) => nil)
    (7 - (cmp '(a b c) '(a b )) => nil)
    (8 - (cmp '(a = c) '(a b c)) => t)
    (9 - (cmp '(a = c) '(a (b x) c)) => t)
    ))
```

Next we write an empty function body for cmp...

cmp version 1

```
(defun cmp (pat lis)
  )

;; note that some of our tests work already!!
cg-user(27): (tester *cmp-tests*)
(failed test. 1 -- (cmp nil nil))
(failed test. 4 -- (cmp '(a b c) '(a b c)))
(failed test. 8 -- (cmp '(a = c) '(a b c)))
(failed test. 9 -- (cmp '(a = c) '(a (b x) c)))
test-completed
```

then we start extending our function...

cmp version 2

This one does not handle wild-cards...

```
(defun cmp (pat lis)
  (cond ((eq pat lis) t)
        ((and (consp pat) (consp lis))
         (and (cmp (car pat) (car lis))
              (cmp (cdr pat) (cdr lis))
              ))
        (t nil)
        ))

cg-user(28): (tester *cmp-tests*)
(failed test. 8 -- (cmp '(a = c) '(a b c)))
(failed test. 9 -- (cmp '(a = c) '(a (b x) c)))
test-completed
```

cmp final version

This one does handle wild-cards...

```
(defun cmp (pat lis)
  (cond ((eq pat lis) t)
        ((eq pat '=)  t)
        ((and (consp pat) (consp lis))
         (and (cmp (car pat) (car lis))
              (cmp (cdr pat) (cdr lis))
              ))
        (t nil)
        ))

cg-user(29): (tester *cmp-tests*)
test-completed
```