

Homework 9: Intersection

Assigned: Tuesday, February 27, 2007

Due: Friday, March 2, 2007

No extensions!

Summary: In this assignment, you will further explore issues pertaining to documentation, testing, and recursive procedures.

Purposes: To reinforce our discussions of documentation and testing. To give you experience writing tests.

Expected Time: One to two hours.

Collaboration: You may work in a group of any size between two and four, inclusive. You may consult others outside your group, provided you cite those others. You need only submit one assignment per group.

Submitting: Email me your work, using a subject of *CSC151 Homework 9*.

Warning: So that this exercise is a learning assignment for everyone, I may spend class time publicly critiquing your work.

Assignment

Your TA, Emily Jacobson, remembers with fondness writing a procedure, (`intersect left right`), which takes two lists of symbols as arguments and returns a list of which the elements are precisely those symbols that are elements of both `left` and `right`. (Although symbols can appear multiple times in `left` and `right`, no symbol may appear more than once in the result.) She's told me that she'd like you to share her joy.

- a. Emily's description is clearly casual. Write formal documentation for `intersect`, using the six P's.
- b. Although Emily's version was perfect, I recall that many of her colleagues had a few subtle bugs in their implementations. Write a test suite that would help her colleagues (and yours) determine whether or not the implementation contains errors. Note that you should be relatively confident that anything that passes your test suite is correct. Note also that the test suite should use the `unit-test.ss` library for testing.

To help you write this test, we have added another keyword to `unit-test.ss`. The `test-permutation!` keyword takes two parameters, an expression and a list. It then evaluates the expression and compares the result to the list. If the expression evaluates to a permutation of the list, it considers the test successful. If not, it considers the test a failure.

c. Write your own version of `intersect`.

Helpful Hints

Testing list operations with `test-permutation!`

The unit testing framework includes another keyword we haven't used yet: `test-permutation!` This lets you make sure an expression yields a list containing the correct items, but it doesn't care what order the items are in.

Here is a sample test suite using `test-permutation!`:

```
(load "/home/rebelsky/Web/Courses/CS151/2007S/Examples/unit-test.scm")
(begin-tests!)
(define one-to-five (list 1 2 3 4 5))
(test-permutation! (list 1 2 3 4 5) one-to-five)
(test-permutation! (list 1 3 5 2 4) one-to-five)
(test-permutation! (reverse one-to-five) one-to-five)
(test-permutation! (cdr one-to-five) one-to-five)      ; Should fail: missing an element
(test-permutation! null one-to-five)                  ; Should fail: missing all elements
(test-permutation! (list 6 5 1 4 2 3) one-to-five)    ; Should fail: extra element
(end-tests!)
```

The first three tests should succeed, because the expression to be tested gives a list that contains the elements 1, 2, 3, 4, and 5, even though they are not necessarily in that order. The rest of the tests should fail. (Of course, you shouldn't write tests that you expect to fail; I'm just trying to show you how `test-procedure!` works.)

What actually happens? Let's see:

```
6 tests conducted.
3 tests failed.
No errors encountered.
3 other tests failed to give the expected result:
  For (cdr one-to-five) expected [(permutation-of (1 2 3 4 5))] got [(2 3 4 5)]
  For null expected [(permutation-of (1 2 3 4 5))] got [()]
  For (list 6 5 1 4 2 3) expected [(permutation-of (1 2 3 4 5))] got [(6 5 1 4 2 3)]
Sorry. You'll need to fix your code.
```

Yes, that is what I expected to happen.

The `member?` Predicate

You will likely find it helpful to use a `member?` predicate, which you should have defined already. If you have difficulty finding your definition, here's mine:

```
;;; Procedure:
;;; member?
;;; Parameters:
;;;   val, a Scheme value
;;;   lst, a list of values
```

```

;;; Purpose:
;;; Determine whether val appears in lst.
;;; Produces:
;;; is-a-member, a Boolean
;;; Preconditions:
;;; (none)
;;; Postconditions:
;;; If there exists a position, p, such that
;;; (equals? val (list-ref lst p))
;;; then is-a-member is #t.
;;; If there is no such position, then is-a-member is #f.
(define member?
  (lambda (val lst)
    (and (not (null? lst))
         (or (equal? val (car lst))
             (member? val (cdr lst))))))

```

Important Evaluation Criteria

In evaluating your work, I will emphasize (a) the clarity and precision of your documentation, (b) the thoroughness of your tests, and (c) the correctness of your implementation.

I also plan to run most all the tests submitted on all of the implementations submitted. The authors of the test suites that correctly reject the most implementations are likely to receive some extra credit, as are the authors of the implementations that pass all of the test suites (including mine).

And yes, if you all write procedures that pass all of the test suites (including mine), then it is likely that you will all receive extra credit.

Copyright © 2007 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.