

Assignment 11: Testing

Due: 4:00 p.m., Friday, 19 October 2007

Summary: In this assignment, you will have the opportunity to develop test suites for two interesting procedures. You will also develop versions of those procedures.

Purposes: To give you experience in thinking about and writing test suites. To introduce you to two important procedures, one useful primarily for image processing, the other more generally useful.

Expected Time: One to two hours.

Collaboration: I encourage you to work in groups of size three. You may, however, work alone or work in a group of size two or size four. You may discuss this assignment with anyone, provided you credit such discussions when you submit the assignment.

Submitting: Email me your answer. More details below.

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

Contents:

- Assignment
 - Problem 1: Weighted Color Averages
 - Problem 2: Removing Duplicates
- Important Evaluation Criteria
- Helpful Hints
 - Testing list operations with `test-permutation!`
 - The `member?` Predicate
 - Removing elements with `remove`
- Submitting Your Homework

Assignment

Problem 1: Weighted Color Averages

In this problem, you'll complete and extend Exercise 5 from the lab on verifying preconditions.

In a number of exercises, we were required to blend two colors. For example, we blended colors in a variety of ways to make interesting images, and we made a color more grey by averaging it with grey. In blending two colors, we are, in essence, creating an average of the two colors, but an average in which each color contributes a different fraction.

For this problem, we might write a procedure, (`rgb.weighted-average fraction color1 color2`) that makes a new color, each of whose components is computed by multiplying the corresponding component of *color1* by *fraction* and adding that to the result of multiplying the corresponding component of *color2* by $(1-fraction)$. For example, we might compute the red component with

```
(+ (* fraction (rgb.red color1)) (* (- 1 fraction) (rgb.red color2)))
```

- a. What preconditions should `rgb.weighted-average` have? (Think about restrictions on *percent*, *color1*, and *color2*.)
- b. How might you formally specify the postconditions for `rgb.weighted-average`?
- c. Document `rgb.weighted-average`.
- d. Because we so frequently find weighted averages of colors, it's really important that our weighted average code work correctly. Write a test suite for `rgb.weighted-average`, as in the lab on testing. (By writing the tests before you write the code, you are practicing test-driven development.)
- e. Write the code for `rgb.weighted-average`, making sure to verify each precondition.

Problem 2: Removing Duplicates

In many situations, we end up building lists of values, such as a list of colors that appear in an image. In some of those situations, we might want to remove duplicates from that list, such as removing duplicate color names or duplicate spots. For such situations, it would be useful to have a procedure, (`list.remove-duplicates lst`), which removes duplicated values. For example,

```
> (list.remove-duplicates (list "red" "green" "blue" "yellow" "red" "orange"))
("green" "blue" "yellow" "red" "orange")
> (list.remove-duplicates (list "red" "white" "red" "white" "red" "green" "red"))
("white" "green" "red")
> (list.remove-duplicates null)
()
```

If our general goal is to “remove duplicates”, then it doesn't really matter which duplicate we remove, as long as only one copy is left. Hence, it would be equally correct if the procedure had returned the following.

```
> (list.remove-duplicates (list "red" "green" "blue" "yellow" "red" "orange"))
("red" "green" "blue" "yellow" "orange")
> (list.remove-duplicates (list "red" "white" "red" "white" "red" "green" "red"))
("red" "white" "green")
```

- a. Document `list.remove-duplicates`, making sure to incorporate the idea that it doesn't matter which duplicate remains.
- b. Write a test suite for `list.remove-duplicates`. Because we didn't say which duplicate to remove, you may wish to use the `test-permutation!` procedure described below.

c. Implement `list.remove-duplicates`, making sure to verify each precondition. You may find the `member?` procedure we wrote for Assignment 9 to be helpful. You may instead find it helpful to write a procedure, `(list.remove vals val)`, that builds a new list by removing every instance of `val` from `vals`.

Important Evaluation Criteria

I will look primarily at the precision of your documentation, the thoroughness of your tests, and whether each procedure is implemented correctly.

Students whose test suites catch errors in other students' code or whose code passes all of the class's tests may receive a higher grade.

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!`:

```
(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))))))
```

Removing elements with `remove`

As the hints on `list.remove-duplicates` suggest, you may find it useful to use `(list.remove vals val)`, which removes all copies of `val` from `vals`. Here's a definition for `remove`.

```
;;; Procedure:
;;; list.remove
;;; Parameters:
;;; vals, a list [unverified]
;;; val, a Scheme value [unverified]
;;; Purpose:
;;; Remove all instances of val from vals.
;;; Produces:
;;; newvals, a list
;;; Preconditions:
;;; [No additional preconditions.]
;;; Postconditions:
;;; No element in newvals is equal to val.
;;; Each value that appears in newvals also appears in vals, and appears
;;; the same number of times in both lists. That is,
;;; (tally nv newvals) = (tally nv vals)
;;; The elements in newvals appear in the same order that they appear
;;; in vals.
(define list.remove
  (lambda (vals val)
```

```
(cond
  ((null? vals) null)
  ((equal? (car vals) val) (remove (cdr vals) val))
  (else (cons (car vals) (remove (cdr vals) val))))))
```

Submitting Your Homework

Please submit this work via email. The email should be titled CSC151.02 Assignment 11 and should contain your answers to all parts of this assignment.

Please send your work as the body of an email message. I don't like attachments, and prefer not to receive them when they can be avoided.

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.