

Assignment 9: Recursion

Due: 4:00 p.m., Friday, 5 October 2007

No extensions!

Summary: In this assignment, you will further practice the design of recursive procedures.

Purposes: To give you more experience with a variety of styles of recursion over lists.

Expected Time: 1-2 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:

- Preparation
- Assignment
 - Problem 1: Reversing Lists
 - Problem 2: Reversing Lists, Revisited
 - Problem 3: Are they all colors?
 - Problem 4: Are they all spots?
 - Problem 5: Keeping Spots
 - Problem 6: Is it there?
- Important Evaluation Criteria
- Submitting Your Homework

Preparation

You will find the following definitions from the lab on list iteration useful.

```
(define spot.new
  (lambda (col row color)
    (list col row color)))

(define spot.col
  (lambda (spot)
    (car spot)))
(define spot.row
  (lambda (spot)
```

```
(cadr spot)))
(define spot.color
  (lambda (spot)
    (caddr spot)))
```

Assignment

Problem 1: Reversing Lists

Suppose the reverse procedure were not included in Scheme. Could you write it yourself? Certainly! It should be possible to implement reverse recursively.

One strategy is to start with the standard formulation of recursive procedures.

```
(define my-reverse
  (lambda (lst)
    (if (null? lst)
        ___base-case___
        (___combine___ (car lst) (my-reverse (cdr lst))))))
```

Finish this implementation.

When using this strategy, you'll need to think about the question: "Suppose I've reversed the cdr of a list. What do I do with the car to get the reversal of the complete list?"

Problem 2: Reversing Lists, Revisited

Of course, you've seen more than one strategy for writing recursive procedures. Another possibility is to use a helper that includes a "what I've done so far" parameter. For example,

```
(define my-other-reverse
  (lambda (lst)
    (my-other-reverse-helper null lst)))

(define my-other-reverse-helper
  (lambda (reversed-so-far remaining-elements)
    (if (null? remaining-elements)
        ___base-case___
        (my-other-reverse-helper (___modify___ reversed-so-far)
                                   (cdr remaining-elements)))))
```

Finish this implementation.

Problem 3: Are they all colors?

Define and test a Scheme predicate, (`all-rgb? values`), that takes a list as argument and determines whether all of its elements are rgb colors, as determined by the `rgb?` predicate.

For example,

```
> (all-rgb? (list color.red color.blue color.green))
#t
> (all-rgb? (list "red" color.red))
#f
> (all-rgb? (list color.red color.blue pi 3.2 color.green))
#f
> (all-rgb? (list 1 2 3))
#t
```

Why is the last example `#t`, since they are numbers, rather than colors? Because we represent colors as integers, the `rgb?` predicate assumes that every integer is a color.

Problem 4: Are they all spots?

Define and test a Scheme predicate, (`all-spots? values`), that takes a list as argument and determines whether all of its elements are spots. Recall that a spot is a list consisting of three elements: a column (an integer), a row (also an integer), and an rgb color.

For example,

```
> (all-spots? (list (spot.new 0 0 color.red)
                    (spot.new 0 1 color.blue)
                    (spot.new 0 2 color.green)))
#t
> (all-spots? (list 42
                    (spot.new 0 0 color.red)
                    (spot.new 0 1 color.blue)
                    (spot.new 0 2 color.green)))
#f
> (all-spots? (list (spot.new 0 0 color.red)
                    (spot.new 0 1 "blue")
                    (spot.new 0 2 color.green)))
#f
> (all-spots? (list (spot.new 0 0 color.red)
                    (spot.new 0 1 color.blue)
                    (spot.new 0 2.01 color.green)))
#f
> (all-spots? (list (spot.new 0 0 color.red)
                    color.blue
                    "out damned spot"))
#f
```

As a hint, you may wish to define a predicate (`spot? value`) that tests whether a value is a valid spot.

Problem 5: Keeping Spots

Write a procedure, (`keep-spots lst`), that, given a list of Scheme values, builds a new list that contains all the values in `lst` that are spots, and no other values.

Problem 6: Is it there?

Define and test a Scheme predicate, `(member? value lst)`, that takes two arguments, a value and a list, and determines whether the given value appears within the given list. (Hint: use the `equal?` predicate to test whether the value is equal to any of the members of the list. Think about what your procedure should return for the base case.)

For example,

```
> (member? color.black (list color.red color.green color.blue color.black color.white))
#t
> (member? color.yellow (list color.red color.green color.blue color.black color.white))
#f
> (member? (spot.new 0 1 color.blue) (list (spot.new 0 0 color.red)
                                           (spot.new 0 1 color.blue)
                                           (spot.new 0 2 color.green)))
#t
> (member? (spot.new 0 1 color.teal) (list (spot.new 0 0 color.red)
                                           (spot.new 0 1 color.blue)
                                           (spot.new 0 2 color.green)))
#f
> (member? (spot.new 1 1 color.blue) (list (spot.new 0 0 color.red)
                                           (spot.new 0 1 color.blue)
                                           (spot.new 0 2 color.green)))
#f
```

Important Evaluation Criteria

I will primarily look to see that your code is correct and uses appropriate style. I may reward particularly elegant or well-crafted code, as well as creative extensions of the required work.

Submitting Your Homework

Please submit this work via email. The email should be titled CSC151.02 Assignment 09 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.