

## Laboratory: Recursion Basics

**Summary:** In this laboratory, you will explore some basic concepts in recursing over lists.

### Contents:

- Preparation
- Exercises
  - Exercise 1: Testing Sum
  - Exercise 2: Removing Dark Colors
  - Exercise 3: Counting Values
  - Exercise 4: Product
  - Exercise 5: Counting Special Values
  - Exercise 6: Summing Components
  - Exercise 7: Filtering Out Reds
- For Those With Extra Time
  - Extra 1: Filtering Out Spots
- Notes
  - Notes on Problem 5: Summing Components

### Reference:

Here are the definitions from the reading.

```
;;; Procedure:
;;;   sum
;;; Parameters:
;;;   numbers, a list of numbers.
;;; Purpose:
;;;   Find the sum of the elements of a given list of numbers
;;; Produces:
;;;   total, a number.
;;; Preconditions:
;;;   All the elements of numbers must be numbers.
;;; Postcondition:
;;;   total is the result of adding together all of the elements of numbers.
;;;   If all the values in numbers are exact, total is exact.
;;;   If any values in numbers are inexact, total is inexact.
(define sum
  (lambda (numbers)
    (if (null? numbers)
        0
        (+ (car numbers) (sum (cdr numbers))))))

(define rgb.filter-out-dark
  (lambda (colors)
```

```

(cond
  ((null? colors) null)
  ((rgb.dark? (car colors)) (rgb.filter-out-dark (cdr colors)))
  (else (cons (car colors) (rgb.filter-out-dark (cdr colors))))))

```

The latter procedure requires `rgb.dark?`, which follows.

```

;;; Procedure:
;;;   rgb.dark?
;;; Parameters:
;;;   color, an RGB color
;;; Purpose:
;;;   Determine if the color appears dark.
;;; Produces:
;;;   dark?, a Boolean
(define rgb.dark?
  (lambda (color)
    (> 33 (rgb.brightness color))))

;;; Procedure:
;;;   rgb.brightness
;;; Parameters:
;;;   color, an RGB color
;;; Purpose:
;;;   Computes the brightness of color on a 0 (dark) to 100 (light) scale.
;;; Produces:
;;;   b, an integer
;;; Preconditions:
;;;   color is a valid RGB color. That is, each component is between
;;;   0 and 255, inclusive.
;;; Postconditions:
;;;   If color1 is likely to be perceived as lighter than color2,
;;;   then (brightness color1) > (brightness color2).
(define rgb.brightness
  (lambda (color)
    (round (* 100 (/ (+ (* .30 (rgb.red color))
                       (* .59 (rgb.green color))
                       (* .11 (rgb.blue color)))
                   255)))))

```

## Preparation

In this laboratory, we will not be working with images (just with colors and with lists), so you need not create an image.

- a. Copy the `sum` and `rgb.filter-out-dark` procedures to your definitions pane.
- b. Copy the `rgb.dark?` and `rgb.brightness` procedures to your definitions pane.
- c. Create a list of a dozen or so colors (red, black, green, blue, yellow, orange, magenta, white, black, etc.). Name it `my-colors`.

## Exercises

### Exercise 1: Testing Sum

- Read through `sum` so that you have a sense of its purpose.
- Verify that `sum` produces the same result as in the corresponding reading.
- What value do you expect `sum` to produce for the empty list? Check your answer experimentally.
- What value do you expect `sum` to produce for a singleton list? Check your answer experimentally.
- Try `sum` for a few other lists, too.

### Exercise 2: Removing Dark Colors

- Read through the definition of `rgb.filter-out-dark` to try to understand what it does.
- Determine which colors in `my-colors` are dark with `(map rgb.dark? my-colors)`.
- Create a list of non-dark colors with `(rgb.filter-out-dark my-colors)`.
- Verify that all the resulting colors are not dark.

### Exercise 3: Counting Values

Suppose the `length` procedure were not defined. We could define it by recursing through the list, counting 1 for each value in the list. In some sense, this is much like the definition of `sum`, except that we use the value 1 rather than the value at each spot.

- Using this idea, write a procedure, `(my-length lst)` that finds the length of a list (without using `length`).
- Check your answer.

### Exercise 4: Product

Write a procedure, `(product nums)`, that, given a list of numbers, computes the product of those numbers. You should feel free to use `sum` as a template. However, you should think carefully about the base case.

### Exercise 5: Counting Special Values

What if we don't want to count every value in a list? For example, what if we only want to count the dark values in a list of colors. In this case, we still recurse through the list, but we sometimes count 1 (when the color is dark) and sometimes count 0 (when the color is not dark).

a. Using this idea, write a procedure, `(rgb.tally-dark colors)`, that, given a list of colors, counts how many are dark.

b. Test your procedure.

## Exercise 6: Summing Components

In the past, we've found it useful to find the average of two colors. Let's consider how we might find the average of a list of colors. First, we would need to find the number of colors in the list. That's easy, we just use the `length` procedure. Next, we need to find the sum of each component. That's a bit harder, but let's suppose we can do it. We next divide each sum by the length, and get the average of that component. Finally, we put it all together with `rgb.new`.

That is, we might write

```
(define average-color
  (lambda (colors)
    (rgb.new (/ (rgb-list.sum-red colors) (length colors))
             (/ (rgb-list.sum-green colors) (length colors))
             (/ (rgb-list.sum-blue colors) (length colors)))))
```

Of course, for this to work, we need to write `rgb-list.sum-red`, `rgb-list.sum-blue`, and `rgb-list.sum-green`. For now, we'll write one of the three. (The other two should be obvious.)

a. Write a procedure, `(rgb-list.sum-red colors)`, that computes the sum of the red components in a color.

b. Test your procedure on a list of a single color.

c. Test your procedure on the `my-colors` list you wrote earlier.

d. It is possible to write `rgb-list.sum-red` without using direct recursion. How? By an appropriate combination of `map` and `sum`. Try doing so. (If you can't find a solution, look at the notes on the problem.)

## Exercise 7: Filtering Out Reds

Write a procedure, `(rgb-list.filter-reds colors)`, that filters out all elements of `colors` with a red component of at least 128.

## For Those With Extra Time

### Extra 1: Filtering Out Spots

Write a procedure, `(spots.bound spots left top right bottom)`, that filters out all elements of `spots` which do not appear in the rectangle bounded at the left by `left`, at the top by `top`, at the right by `right`, and at the bottom by `bottom`.

For example, if *left* is 5, *top* is 10, *right* is 8, and *bottom* is 20, the procedure should remove any spots whose column is less than 5 or more than 8 or whose row is less than 10 or more than 20.

## Notes

### Notes on Problem 5: Summing Components

We can use `map` to extract the red component of each color.

```
(map rgb.red colors)
```

That gives us a list of numbers, which we can sum with `sum`.

```
(sum (map rgb.red colors))
```

Putting it all together in a procedure

```
(define rgb-list.sum-red  
  (lambda (colors)  
    (sum (map rgb.red colors))))
```

---

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.