Fundamentals of Computer Science I: Media Computing (CS151.02 2007F)

Laboratory: Tail Recursion

Summary: In this laboratory, you will explore recursive techniques in which we pass along intermediate computations. When this technique is used in conjunction with a structure in which the recursive result is returned unchanged, this technique supports *tail recursion*.

Contents:

Reference:

Preparation

Make a copy of `new-sum`, `new-difference`, and `newer-difference` from the reading.

Create a list of twelve or so RGB colors and call it `my-colors`.

Exercises

Exercise 1: Exploring The Basic Procedures

Test `new-sum`, `new-diff`, and `newer-diff`, to determine whether or not they behave as they should.

Exercise 2: Product, Revisited

Rewrite the `product` procedure, which computes the product of a list of values, using the same technique used for `new-sum`.

Exercise 3: Summing Red Components, Revisited

In the previous lab, you wrote a procedure, `rgb-list.sum-red`, that sums all the red components in a list of colors. Rewrite that procedure using the technique of carrying along intermediate values. Your procedure should look something like the following:

```
(define rgb-list.sum-red
  (lambda (colors)
     (rgb-list.sum-red-helper 0 colors)))
(define rgb-list.sum-red-helper
  (lambda (sum-so-far remaining-colors)
    ...))
```

Exercise 4: Summing Multiple Components, Simultaneously

One advantage of this technique is that we can pass along multiple intermediate results. For example, in averaging a list of colors, we can keep track of the sum of reds, the sum of greens, the sum of blues, and the count of colors. In the end, we can build a new color from these computed values.

```
(define rgb-list.average
  (lambda (colors)
    (rgb-list.average-helper 0 0 0 0 colors)))
(define rgb-list.average-helper
  (lambda (red-so-far green-so-far blue-so-far count remaining-colors)
    (if (null? remaining-colors)
        (rgb.new (/ red-so-far count)
                 (/ green-so-far count)
                 (/ blue-so-far count))
        (rgb-list.average-helper ...))))
```

a. Fill in the remaining code in the recursive call.

b. Test this code to average white and black.

```
(rgb->string (rgb-list.average (list color.white color.black)))
```

c. Test this code on a few other colors of your choice.

d. What do you expect to have happen if you provide `rgb-list.average` with the empty list?

e. Check your answer experimentally.

Exercise 5: Filtering, Revisited

a. Print out the RGB values in your list of colors with

```
(map rgb->string my-colors)
```

b. Here's a procedure using the "results so far" technique that filters out any colors with a red component of at least 128.

```
(define rgb-list.filter-out-high-red
  (lambda (colors)
    (rgb-list.filter-out-high-red-helper null colors)))
(define rgb-list.filter-out-high-red-helper
  (lambda (colors-so-far remaining-colors)
    (cond
      ((null? remaining-colors)
       colors-so-far)
```

```
      ((<= 128 (rgb.red (car remaining-colors)))
       (rgb-list.filter-out-high-red-helper colors-so-far (cdr remaining-colors)))
      (else
       (rgb-list.filter-out-high-red-helper
         (cons (car remaining-colors) colors-so-far)
         (cdr remaining-colors))))))
```

c. What do you expect this procedure to do when given the empty list? Check your result experimentally.

d. What do you expect this procedure to do when given the list of the color white? Check your result experimentally.

e. What do you expect this procedure to do when given the list of the color black? Check your result experimentally.

f. What do you expect this procedure to do when given the list of the color blue? Check your result experimentally.

g. What do you expect this procedure to do when given the list of the color yellow? Check your result experimentally.

h. What do you expect this procedure to do when given the list of the color green? Check your result experimentally.

g. What do you expect this procedure to do when given the list of the colors white, black, blue, yellow, green? Check your result experimentally.

```
(map rgb->string (rgb-list.filter-out-high-red (list color.red color.green color.blue color.yellow color.black color.white)))
```

h. In at least one case above, you should have received a somewhat strange result. Do your best to explain that result (ask me or a TA if you're confused). Then, fix the code so that the result is not so strange.

For Those With Extra Time

Extra 1: Finding The Brightest Color

You may recall the `brightness` procedure that we've used in a number of labs.

```
;;; 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)))))
```

Write a procedure, `(rgb-list.brightest` *`colors`*`)`, that finds the brightest color in a list of colors.

Hint: Reconsider the `newer-difference` procedure, and how we chose the initial value used for a preliminary result. What could we use as the preliminary result here?
