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

Laboratory: Recursion, Revisited

Summary: In this laboratory, you will continue your exploration of recursion.

Contents:

Reference:

Preparation

a. Create a list of a dozen or so colors and call it `my-colors`. (Put this definition in the definitions pane.)

b. Create a list of the names of all of the colors with green in the name with

```
(define green-names (cname.list "green"))
```

c. Create a list of the RGB equivalents of all of those colors with

```
(define greens (map cname->rgb green-names))
```

d. Create a list of a few shades of grey with

```
(define greys
  (map (lambda (n) (rgb.new n n n))
       (list 0 16 32 48 64 96 112 128 144 160 176 192 208 224 240 255)))
```

e. Add the following procedures to your definitions pane.

```
;;; Procedure:
;;;   rgb.bright?
;;; Parameters:
;;;   color, an RGB color
;;; Purpose:
;;;   Determine if the color appears bright.
;;; Produces:
;;;   bright?, a Boolean
(define rgb.bright?
  (lambda (color)
     (< 66 (rgb.brightness color))))

;;; Procedure:
;;;   rgb.brightness
;;; Parameters:
;;;   color, an RGB color
;;; Purpose:
;;;   Computes the brightness of color on a 0 (dark) to 100 (bright) 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 brighter 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)))))

;;; Procedure:
;;;   rgb.distance
;;; Parameters:
;;;   color1, an RGB color
;;;   color2, an RGB color
;;; Purpose:
;;;   Finds the distance between color1 and color2 using some metric.
;;; Produces:
;;;   distance, an integer
(define rgb.distance
  (lambda (color1 color2)
    (+ (square (- (rgb.red color1) (rgb.red color2)))
       (square (- (rgb.green color1) (rgb.green color2)))
       (square (- (rgb.blue color1) (rgb.blue color2))))))
```

Exercises

Exercise 1: Appending Lists

You may recall that the procedure `append` takes as parameters two lists, and joins the two lists together. Let's generalize that procedure. Write a procedure, `lists.join`, that, given a list of lists as a parameter, joins each of the member lists together using `append`.

```
> (lists.join (list (list 1 2 3)))
(1 2 3)
> (lists.join (list (list 1 2 3) (list 10 11 12)))
(1 2 3 10 11 12)
> (lists.join (list (list 1 2 3) (list 10 11 12) (list 20 21)))
(1 2 3 10 11 12 20 21)
> (lists.join (list null (list 1 2 3) null null null null (list 100 99 98) null))
(1 2 3 100 99 98)
```

Exercise 2: The Brightest Color

a. Write a procedure, `(rgb.brightest colors)`, that, given a list of colors, finds the brightest color in that list. In writing your procedure, use the following structure:

- Base case: If the list has only one element, that element is the brightest.
- First recursive case: If the first element of the list is brighter than the brightest color in the rest of the list, use the first element of the list.
- Second recursive case: If the first element of the list is not brighter than the brightest color in the rest of the list, use the brightest color in the rest of the list. (Note that if the tests for the base case and the first recursive case fail, the test for this case must hold.)

b. When you are done, compare your answer to the solution that appears in the notes on this problem

c. *Save your work!*

d. Test this procedure on your list of colors, `my-list`.

e. Test this procedure on `greens`. For example, you might write

```
(rgb->cname (rgb.brightest greens))
```

f. What do you expect to have happen if you call this procedure on the empty list of colors?

g. Experimentally check your answer to the previous step.

h. Test this procedure on a few other lists.

Exercise 3: The Brightest Grey

a. Determine what happens when `rgb.brightest` is called on `greys`.

b. Determine what happens when `rgb.brightest` is called on `(reverse greys)`.

c. Did you notice any difference in the behavior (if not the result) of the two calls?

d. You should have observed that one of the two calls was much faster. Hypothesize as to why.

Exercise 4: The Brightest Color, Revisited

Is your solution to the previous exercise the only way to write a recursive procedure to find the brightest color in a list? Certainly not! For example, we might write something that follows the pattern of `spot-list.leftmost`.

```
(define spot-list.leftmost
  (lambda (spots)
    (if (null? (cdr spots))
        (car spots)
        (spot.leftmost (car spots) (spot-list.leftmost (cdr spots))))))
```

What’s the key idea in this procedure (other than using recursion and having a singleton base case)? We use a single procedure, `spot.leftmost`, to handle two cases: The recursive case in which the first element is to the left of the remaining elements, and the case in which the first element is not to the left of the remaining elements.

a. To use this pattern, you’ll need to create the equivalent of `spot.leftmost`, except that it finds the brighter of two colors. Write a procedure, `(rgb.brighter` *`color1 color2`*`)` that returns the brighter of *color1* and *color2*.

b. Finish your definition of `rgb.brightest` by replacing the appropriate parts of the aforementioned definition.

c. Compare your answer to the one that appears in the notes on this problem.

d. *Save your work!*

e. Determine what happens when the procedure is called on the empty list.

f. Determine what happens when the procedure is called on a singleton list.

g. Determine what happens when the procedure is called on `greens`.

h. Determine what happens when the procedure is called on `greys`.

i. Determine what happens when the procedure is called on `(reverse greys)`.

Exercise 5: The Brightest Color, Re-Revisited

In the reading and laboratory on tail recursion, we used a very different technique for writing recursive procedures: In addition to passing along a list that we were recursing over, we also passed along an intermediate result, which we used when we hit the base case. Let's try rewriting `rgb.brightest` that way.

```
(define rgb.brightest
  (lambda (colors)
     (rgb.brightest-helper (car colors) (cdr colors))))
(define rgb.brightest-helper
  (lambda (brightest-so-far remaining-colors)
    (if (null? remaining-colors)
        brightest-so-far
        (rgb.brightest-helper ______________________
                              (cdr remaining-colors)))))
```

a. Finish this definition.

b. Compare it to the definition in the notes on this problem.

c. Test it on your list of colors (`my-list`) and the list we generated algorithmically (`many-colors`).

Exercise 6: Reflect!

You've now come up with (or read) three definitions of `rgb.brightest`. Which do you prefer? Why?

Exercise 7: The Closest Color

Write and test a procedure, `(rgb.closest` *`color colors`*`)`, that finds the element of *colors* that is closest to *color*.

Exercise 8: Checking for Brightness

a. Write a procedure, `(all-bright?` *`colors`*`)`, that, given a list of colors, determines if all of the colors are bright.

b. Write a procedure, `(any-bright?` *`colors`*`)`, that, given a list of colors, determines if any of them are bright.

For Those With Extra Time

Extra 1: Templates

Write your own "fill in the blanks" template for the three kinds of recursion covered in the three forms of `rgb.brightest`.

Notes

Notes on Problem 2: The Brightest Color

Here's a solution that matches the description given earlier.

```
(define rgb.brightest
  (lambda (colors)
    (cond
      ((null? (cdr colors))
       (car colors))
      ((> (rgb.brightness (car colors))
          (rgb.brightness (rgb.brightest (cdr colors))))
       (car colors))
      (else (rgb.brightest (cdr colors))))))
```

Notes on Problem 3: The Brightest Grey

You should have observed that finding the brightest color in `(reverse greys)` is significantly faster than finding the brightest color in `greys`. Why should this be, particularly since the two lists have exactly the same values? It turns out that the order of the elements makes a difference. In `(reverse greys)`, the elements are in order from brightest to darkest. In `greys`, the elements are in order from darkest to brightest. So, in `(rgb.brightest (reverse greys))`, we always use the first recursive case. However, in `(rgb.brightest greys)`, we always use the second recursive case. That means that we call `rgb.brightest` twice.

Why isn't it the case that it's only half as fast to do this call? Well, suppose we have a list of ten elements. We recurse twice on a list of nine elements. Each of these recursive calls recurses twice on a list of eight elements. That means that we've done four calls on lists of eight elements. Since each of these does two calls on lists of seven elements, we do eight calls on lists of seven elements. Similarly, there are sixteen calls on lists of six elements, thirty two on lists of five elements, and so on and so forth.

We will return to this efficiency issue later in the semester.

Notes on Problem 4: The Brightest Color, Revisited

Here's a solution that matches the description given earlier.

```
(define rgb.brightest
  (lambda (colors)
    (if (null? (cdr colors))
        (car colors)
        (rgb.brighter (car colors) (rgb.brightest (cdr colors))))))
```

Notes on Problem 5: The Brightest Color, Re-Revisited

Here’s a solution that matches the description given earlier.

```
(define rgb.brightest
  (lambda (colors)
     (rgb.brightest-helper (car colors) (cdr colors))))
(define rgb.brightest-helper
  (lambda (brightest-so-far remaining-colors)
    (if (null? remaining-colors)
        brightest-so-far
        (rgb.brightest-helper
           (rgb.brighter brightest-so-far (car remaining-colors))
           (cdr remaining-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.