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

Laboratory: Local Procedures and Recursion

Summary: In this laboratory, we consider the various techniques for creating local recursive procedures, particularly `letrec` and named `let`. We also review related issues, such as husk-and-kernel programming.

Contents:

Preparation

a. If any of the following procedures are not in your library, add them to your library.

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

;;; Procedure:
;;;   rgb.dark?
;;; Parameters:
;;;   color, an RGB color
;;; Purpose:
;;;   Determine if the color appears dark.
```

```
;;; Produces:
;;;   dark?, a Boolean
(define rgb.dark?
  (lambda (color)
     (> 34 (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 (/ (+ (* 0.30 (rgb.red color))
                        (* 0.59 (rgb.green color))
                        (* 0.11 (rgb.blue color)))
                     255)))))

;;; Procedure:
;;;   rgb.brighter?
;;; Parameters:
;;;   color1, a color
;;;   color2, a color
;;; Purpose:
;;;   Determine if color1 is strictly brighter than color 2.
;;; Produces:
;;;   brighter?, a Boolean
;;; Preconditions:
;;;   [No additional preconditions.]
;;; Postconditions:
;;;   If (rgb.brightness color1) > (rgb.brightness color2)
;;;     then brighter is true (#t)
;;;   Otherwise
;;;     brighter is false (#f)
(define rgb.brighter?
  (lambda (color1 color2)
    (> (rgb.brightness color1) (rgb.brightness color2))))
```

b. Create a new window in which you load your library file.

Exercises

Exercise 1: The Brightest Color, Revisited

As the reading suggested, local kernels are particularly appropriate for checking preconditions. Local kernels are also appropriate for writing the helpers for the recursive helper procedures that take an extra parameter, one that accumulates a result.

For example, here's the standard helper-based definition of `rgb.brightest`.

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

a. Rewrite this to make the helper local and to name the helper `kernel`.

b. Test your procedure on a few simple lists of colors. For example, you might try some of the following.

```
(rgb->string (rgb.brightest (list color.red color.green color.blue)))
(rgb->string (rgb.brightest (list color.black)))
(rgb->string (rgb.brightest (list color.red. color.black color.green color.yellow)))
(rgb->string (rgb.brightest (list color.yellow color.red. color.black color.green)))
```

c. When you are done, you may want to compare your answer to the sample solution in the notes on this problem.

Exercise 2: Safely Computing the Brightest Color

The procedure given above, and your rewrite of that procedure, will fail miserably if given an inappropriate parameter, such as an empty list, a list that contains values that are not colors, or a non-list. Rewrite `rgb.brightest` so that it checks its preconditions (including that the list contains only colors) before calling the kernel.

Exercise 3: Computing the Brightest Color, Revisited Again

Rewrite `rgb.brightest-helper` using a named `let` rather than `letrec`. (The goal of this problem is to give you experience using named `let`, so you need not check preconditions for this exercise.)

When you are done, you may want to compare your answer to the sample solution in the notes on this problem.

Exercise 4: Alternating Lists

A list of spots is a *light-dark-alternator* if its elements are alternately light spots and dark spots, beginning with a light spot. A list of spots is a *dark-light-alternator* if its elements are alternately dark and light, beginning with a dark spot. (We say that the empty list is both a light-dark-alternator and a dark-light-alternator.)

a. Write a predicate, `spots.alternating-brightness?`, that determines whether a non-empty list of spots is either a light-dark-alternator or a dark-light-alternator. Your definition should include a a `letrec` expression in which the identifiers `light-dark-alternator?` and `dark-light-alternator?` are bound to *mutually recursive* predicates, each of which determines whether a given list has the indicated characteristic.

```
(define spots.alternating-brightness?
  (lambda (spots)
    (letrec ((light-dark-alternator?
              (lambda (spots) ...))
             (dark-light-alternator?
              (lambda (spots) ...)))
      ...)))
```

When you are done, you may want to compare your answer to the sample solution in the notes on this problem.

b. Here are a few lists of spots. For which do you expect `spots.alternating-brightness?` to hold?

```
(define sample0 null)

(define sample1
  (list (spot.new 0 0 color.white)))

(define sample2
  (list (spot.new 0 0 color.black)))

(define sample3
  (list (spot.new 0 0 color.white)
        (spot.new 0 1 color.black)))

(define sample4
  (list (spot.new 0 0 color.black)
        (spot.new 0 1 color.white)))

(define sample5
  (list (spot.new 0 0 color.black)
        (spot.new 0 1 color.black)))

(define sample6
  (list (spot.new 0 0 color.white)
        (spot.new 0 1 color.black)
        (spot.new 0 2 color.white)))

(define sample7
```

```
  (list (spot.new 0 0 color.white)
        (spot.new 0 1 color.black)
        (spot.new 0 2 color.white)
        (spot.new 0 3 color.black)
        (spot.new 0 4 color.white)))

(define sample8
  (list (spot.new 0 0 color.red)
        (spot.new 1 0 color.yellow)
        (spot.new 2 0 color.blue)
        (spot.new 3 0 color.white)
        (spot.new 4 0 color.black)))
```

c. Check your answers experimentally.

Exercise 5: Taking Some Elements

Define and test a procedure, (`list.take` *`lst n`*), returns a list consisting of the first *n* elements of the list, `lst`, in their original order. You might also think of `take` as returning all the values that appear before index `n`.

For example,

```
> (list.take (list "a" "b" "c" "d" "e") 3)
("a" "b" "c")
> (list.take (list 2 3 5 7 9 11 13 17) 2)
(2 3)
> (list.take (list "here" "are" "some" "words") 0)
()
> (list.take (list null null) 2)
(() ())
> (map rgb->string (list.take (list rgb.black rgb.white rgb.green) 1))
("0/0/0")
```

The procedure should signal an error if `lst` is not a list, if `n` is not an exact integer, if `n` is negative, or if `n` is greater than the length of `lst`.

Note that in order to signal such errors, you may want to take advantage of the husk-and-kernel programming style.

Exercise 6: Taking Some More Elements

Rewrite `list.take` to use whichever of named `let` and `letrec` you didn't use in the previous exercise.

Exercise 7: Reflection

You've now seen two examples in which you've written two different solutions, one using `letrec` and one use named let. Reflect on which of the two strategies you prefer and why.

For Those With Extra Time

Extra 1: Iota, Revisited

As you may recall, the `iota` procedure takes a natural number as a parameter and returns a list of all the lesser natural numbers in ascending order. For example,

```
> (iota 5)
(0 1 2 3 4)
```

a. Define and test a version of the `iota` procedure that uses `letrec` to pack an appropriate kernel inside a husk. The husk should do precondition testing and the kernel should build the list. This version of `iota` should look something like

```
(define iota
  (lambda (num)
    (letrec ((kernel (lambda (...) ...)))
      (cond
        ((fails-precondition) (error ...))
        ...
        (else (kernel num))))))
```

b. Define and test a version of the `iota` procedure that uses a named `let`. This version of `iota` should look something like

```
(define iota
  (lambda (num)
    (cond
      ((fails-precondition) (error ...))
      ...
      (else
       (let kernel (...)
         ...)))))
```

Notes

Notes on Exercise 1: The Brightest Color, Revisited

Here's one possible solution.

```
(define rgb.brightest
  (lambda (colors)
    (letrec ((kernel
              (lambda (brightest-so-far remaining-colors)
                (cond
                  ((null? remaining-colors)
                   brightest-so-far)
                  ((rgb.brighter? (car remaining-colors) brightest-so-far)
                   (kernel (car remaining-colors)
                           (cdr remaining-colors)))
```

```
                (else
                 (kernel brightest-so-far
                         (cdr remaining-colors)))))))
      (kernel (car colors) (cdr colors)))))
```

Notes on Exercise 3: Computing the Brightest Color, Revisited Again

Here's one possible solution.

```
(define rgb.brightest
  (lambda (colors)
    (let kernel ((brightest-so-far (car colors))
                 (remaining-colors (cdr colors)))
           (cond
             ((null? remaining-colors)
              brightest-so-far)
             ((rgb.brighter? (car remaining-colors) brightest-so-far)
              (kernel (car remaining-colors)
                      (cdr remaining-colors)))
             (else
              (kernel brightest-so-far
                      (cdr remaining-colors)))))))
```

Notes on Exercise 4: Alternating Lists

Once `light-dark-alternator?` and `dark-light-alternator?` are written, the definition is fairly straightforward. A non-empty list of spots has alternating brightness if it's not empty and it's either a light-dark-alternator or a dark-light-alternator.

```
      (and (not (null? spots))
           (or (light-dark-alternator? spots)
               (dark-light-alternator? spots)))
```

Each definition is also fairly straightforward. A list is a light-dark-alternator if it's empty or if the car is light and the cdr is a dark-light-alternator.

```
             (light-dark-alternator?
              (lambda (spots)
                (or (null? spots)
                    (and (rgb.light? (spot.color (car spots)))
                         (dark-light-alternator? (cdr spots))))))
```

The definition of `dark-light-alternator?` is so similar that we will not give it separately.

Putting everything together, we get

```
(define spots.alternating-brightness?
  (lambda (spots)
    (letrec ((light-dark-alternator?
              (lambda (spots)
                (or (null? spots)
                    (and (rgb.light? (spot.color (car spots)))
                         (dark-light-alternator? (cdr spots))))))
             (dark-light-alternator?
```

```
          (lambda (spots)
            (or (null? spots)
                (and (rgb.dark? (spot.color (car spots)))
                     (light-dark-alternator? (cdr spots)))))))
    (and (not (null? spots))
         (or (light-dark-alternator? spots)
             (dark-light-alternator? spots))))))
```

Note that there's a hidden moral here: The procedures defined in a `letrec` can be *mutually* recursive.
