

Repetition Through Recursion

Summary: In this laboratory, you will begin your experiments with recursion by (1) considering some pre-defined recursive procedures and (2) designing your own recursive procedures.

Contents:

- Exercises
 - Exercise 0: Preparation
 - Exercise 1: Sum
 - Exercise 2: The Largest Element in a List
 - Exercise 3: Another Largest-of-List Procedure
 - Exercise 4: Another Size Metric
 - Exercise 5: Product
- For Those With Extra Time
 - Extra 1: Closest to Zero
 - Extra 2: Squaring List Elements
 - Extra 3: Moving On

Exercises

Exercise 0: Preparation

- a. Reflect on the key aspects of recursion.
- b. Start DrScheme.

Exercise 1: Sum

Here are the two versions of `sum` as given in the reading on recursion.

```
;;; Procedures:  
;;; sum  
;;; new-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 ls.  
;;; 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 new-sum
  (lambda (numbers)
    (new-sum-helper 0 numbers)))

;;; Procedure:
;;; new-sum-helper
;;; Parameters:
;;; sum-so-far, a number.
;;; remaining, a list of numbers.
;;; Purpose:
;;; Add sum-so-far to the sum of the elements of a given list of numbers
;;; Produces:
;;; total, a number.
;;; Preconditions:
;;; All the elements of remaining must be numbers.
;;; sum-so-far must be a number.
;;; Postcondition:
;;; total is the result of adding together sum-so-far and all of the
;;; elements of remaining.
;;; If both sum-so-far and all the values in remaining are exact,
;;; total is exact.
;;; If either sum-so-far or any values in remaining are inexact,
;;; total is inexact.
(define new-sum-helper
  (lambda (sum-so-far remaining)
    (if (null? remaining)
        sum-so-far
        (new-sum-helper (+ sum-so-far (car remaining))
                        (cdr remaining)))))

```

- a. Verify experimentally that they work.
- b. Which version do you prefer? Why?
- c. Here's an alternative definition of new-sum.

```

(define newer-sum
  (lambda (numbers)
    (new-sum-helper (car numbers) (cdr numbers))))

```

Is it superior or inferior to the previous definition? Why?

Exercise 2: The Largest Element in a List

Here is a variant of the largest-of-list procedure given in the reading on recursion.

```

;;; Procedures:
;;; largest-of-list
;;; Parameters:
;;; numbers, a list of real numbers.
;;; Purpose:
;;; Find the largest element of a given list of real numbers
;;; Produces:
;;; largest, a real number.
;;; Preconditions:
;;; numbers is not empty.
;;; All the values in numbers are real numbers. That is, numbers
;;; contains only numbers, and none of those numbers are complex.
;;; Postconditions:
;;; largest is an element of numbers (and, by implication, is real).
;;; largest is greater than or equal to every element of numbers.
(define largest-of-list
  (lambda (numbers)
    ; If the list has only one element
    (if (null? (cdr numbers))
        ; Use that one element
        (car numbers)
        ; Otherwise, take the greater of
        ; (a) the first element of the list
        ; (b) the largest remaining element
        (max (car numbers)
             (largest-of-list (cdr numbers))))))

```

a. Experiment with this procedure to verify that it works as advertised.

b. There's at least one kind of list for which this procedure fails to work. Can you tell what kind? Why might we have made that decision?

Exercise 3: Another Largest-of-List Procedure

Here's another version of `largest-of-list`, one that uses a technique like `new-sum` (that is, it has a helper that adds an extra parameter to keep track of the largest value so far).

```

(define new-largest-of-list
  (lambda (numbers)
    (new-largest-of-list-helper (car numbers) (cdr numbers))))

(define new-largest-of-list-helper
  (lambda (largest-so-far remaining-numbers)
    ; If no elements remain ...
    (if (null? remaining-numbers)
        ; Use the largest value seen so far
        largest-so-far
        ; Otherwise, update the guess using the next element
        ; and continue
        (new-largest-of-list-helper (max largest-so-far
                                         (car remaining-numbers))
                                   (cdr remaining-numbers))))))

```

- a. Verify experimentally that this procedure works correctly.
- b. You may find it helpful to “watch” `new-largest-of-list-help`.

Add the following two lines to `new-largest-of-list-helper` (right before the outermost `if`).

```
(display (list 'new-largest-of-list-helper remaining-numbers largest-so-far))
(newline)
```

What effect does this change have?

- c. How is `new-largest-of-list` procedure similar to `largest-of-list`? How is it different?
- d. Which of the two versions do you prefer? Why?

Exercise 4: Another Size Metric

As some students note, there is no need for us to define any of the variants of `largest-of-list`, since Scheme gives us an equivalent procedure in `max`. While that complaint has a glimmer of validity, it is, in fact, useful to figure out how to write many of the built-in Scheme procedures. One reason is that learning the structure of those procedures makes it easier to write other, similar, procedures.

Consider the problem of finding one of the longest strings in a list of strings, a string whose length is at least as large as the length of each of the other strings. (We say “one of the longest strings” because more than one string may have the same length.) For example,

```
> (longest-string-in-list (list "zebras" "have" "stripes"))
"stripes"
> (longest-string-in-list (list "stripes" "often" "appear" "on" "zebras"))
"stripes"
> (longest-string-in-list (list "this" "is" "a" "silly" "list"))
"silly"
> (longest-string-in-list (list "this" "is" "a" "long" "list"))
"this" ; or "list", or although it seems unlikely, "long"
```

We might document this procedure as follows:

```
;;; Procedure:
;;; longest-string-in-list
;;; Parameters:
;;; los, a nonempty list of strings
;;; Purpose:
;;; Finds a longest string in los.
;;; Produces:
;;; longest, a string
;;; Preconditions:
;;; (none)
;;; Postconditions:
;;; For any string in los, (string-length longest) >= (string-length str) .
;;; longest is an element of los.
;;; Problems:
;;; los may contain many strings that meet the postconditions. We do
;;; not specify which of those strings is returned.
```

Using either of the patterns of `largest-of-list`, write `longest-string-in-list`.

Note that you might find it helpful to write a `(longer-of-two str1 str2)` procedure that finds the larger of two strings.

Exercise 5: Product

Define and test a Scheme procedure, `(product values)`, that takes a list of numbers as its argument and returns the result of multiplying them all together. For example,

```
> (product (list 3 5 8))
120
> (product (list 1 2 3 4 5 0))
0
```

Warning: `(product null)` should *not* be 0. It should be the identity for multiplication, just as `(sum null)` is the identity for addition. Explain why.

For Those With Extra Time

Extra 1: Closest to Zero

Write a Scheme procedure, `(closest-to-zero values)`, that, given a list of numbers (including both negative and positive numbers), returns the value closest to zero in the list.

Hint: Think about how, given two numbers, you determine which is closer to zero.

Hint: Think about how this problem is similar to a problem or problems we've solved before.

Extra 2: Squaring List Elements

Define and test a Scheme procedure, `(square-each-element values)`, that takes a list of numbers as its argument and returns a list of their squares.

```
> (square-each-element (list -7 3 12 0 4/5))
(49 9 144 0 16/25)
```

Hint: For the base case, consider what the procedure should return when given a null list; for the other case, separate the car and the cdr of the given list and consider how to operate on them so as to construct the desired result.

Extra 3: Moving On

If you find that you not only finish this laboratory early, but also finish the two extra exercises early, you can start your next homework assignment.

Copyright © 2007 Samuel A. Rebelsky. 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.