

## Analyzing Procedures

**Summary:** In the laboratory, you will explore the running time for a few algorithm variants.

### Contents:

- Preparation
- Exercises
  - Exercise 1: Manual Analysis
  - Exercise 2: Automatic Analysis
  - Exercise 3: Additional Calls
  - Exercise 4: Predicting Calls
  - Exercise 5: The Largest Element, Revisited
  - Exercise 6: Another Version
- For Those with Extra Time
  - Extra 1: Yet Another `largest-of-list`
  - Extra 2: Iota, Revisited

## Preparation

a. In DrScheme, create a new file for this lab, called `analysis-examples.scm`.

b. Make the first line of that file an instruction to load `analyst.scm`.

```
(load "/home/rebelsky/Web/Courses/CS151/2007S/Examples/analyst.scm")
```

c. Add the comments and code for `reverse-1`, `reverse-2`, and `my-append` from the corresponding reading to your file.

## Exercises

### Exercise 1: Manual Analysis

a. Add the following line to the beginning of `myappend` (again, immediately after the `lambda`).

```
(display (list 'myappend front back)) (newline)
```

b. Determine how many times `myappend` is called when reversing a list of length seven using `reverse-1`.

c. Add the following line to the kernel of `reverse-2` (immediately after the `lambda`).

```
(display (list 'kernel remaining reversed)) (newline)
```

- d. Determine how many times `kernel` is called when reversing a list of length seven using `reverse-2`.
- e. Comment out the lines that you just added by prefixing them with a semicolon.

## Exercise 2: Automatic Analysis

- a. Replace the `define` for `reverse-1` with `define$`, as in the following.

```
(define$ reverse-1
  (lambda (lst)
    ...))
```

- b. Find out how many times `myappend` is called in reversing a list of seven elements by entering the following command in the interactions pane.

```
> (analyze (reverse-1 (list 1 2 3 4 5 6 7)) myappend)
```

- c. Did you get the same answer as in the previous exercise? If not, why do you think you got a different result?

- d. One potential issue is that we haven't told the analyst to include the recursive calls in `myappend`. We can do so by replacing `define` with `define$` in the definition of `myappend`.

- e. Once again, find out how many times `myappend` is called in reversing a list of seven elements by entering the following command in the interactions pane.

```
> (analyze (reverse-1 (list 1 2 3 4 5 6 7)) myappend)
```

- f. Did you get the same answer as in exercise 1? If not, what difference do you see?

- g. Replace the `define` in `reverse-2` with `define$`.

- h. Find out how many times `kernel` is called in reversing a list of seven elements by entering the following command in the interactions pane.

```
> (analyze (reverse-2 (list 1 2 3 4 5 6 7)) kernel)
```

- i. Did you get the same answer as in exercise 1? If not, what difference do you see?

## Exercise 3: Additional Calls

In the previous exercise, you considered only a single procedure in each case (`myappend` for `reverse-1`, `kernel` for `reverse-2`). Suppose we incorporate all of the other procedures. What effect does it have?

- a. Find out how many total procedure calls are done in reversing a list of length seven, using `reverse-1`, with the following.

```
> (analyze (reverse-1 (list 1 2 3 4 5 6 7)))
```

b. How does that number of calls seem to relate to the number of calls to myappend?

c. Are there any procedures you're surprised to see?

d. Find out how many total procedure calls are done in reversing a list of length seven, using reverse-2, with the following.

```
> (analyze (reverse-2 (list 1 2 3 4 5 6 7)))
```

e. How does that number of calls seem to relate to the number of calls to kernel?

f. Are there any procedures you're surprised to see?

## Exercise 4: Predicting Calls

a. Fill in the following chart to the best of your ability.

List Length	r1: Calls to myappend	r1: Total calls	r2: Calls to kernel	r2: Total calls
2				
4				
8				
16				

b. Predict what the entries will be for a list size of 32.

c. Check your results experimentally.

d. Write a formula for the columns, to the best of your ability.

## Exercise 5: The Largest Element, Revisited

Here is the more efficient version of largest-of-list from the corresponding reading.

```
;;; Procedures:  
;;; largest-of-list-2  
;;; Parameters:  
;;; lst, a nonempty list of real numbers [verified]  
;;; Purpose:  
;;; Find the largest number in lst.  
;;; Produces:  
;;; largest, a real number  
;;; Preconditions:  
;;; [No additional preconditions]  
;;; Postconditions:
```

```

;;; largest is an element of lst.
;;; For all valid indices i, largest >= (list-ref lst i)
(define largest-of-list-2
  (lambda (lst)
    (if (null? (cdr lst))
        (car lst)
        (let ((largest-remaining (largest-of-list-2 (cdr lst))))
          (if (> (car lst) largest-remaining)
              (car lst)
              largest-remaining))))))

```

- Find out how many steps this procedure takes on lists of length 2, 4, 8, and 16 in which the elements are arranged from largest to smallest.
- Find out how many steps this procedure takes on lists of length 2, 4, 8, and 16 in which the elements are arranged from smallest to largest.
- Find out how many steps this procedure takes on lists of length 2, 4, 8, and 16 in which the elements are in no particular order.
- Predict the number of steps this procedure will take on each kind of list, where the length is 32.

## Exercise 6: Another Version

Some people prefer to test for a one element list by checking the length. They might rewrite the preceding procedure as follows:

```

(define$ largest-of-list-3
  (lambda (lst)
    (if (= 1 (length lst))
        (car lst)
        (let ((largest-remaining (largest-of-list-3 (cdr lst))))
          (if (> (car lst) largest-remaining)
              (car lst)
              largest-remaining))))))

```

- Does the change seem to have an effect? If so, what is that effect?
- One problem with the preceding analysis is that we don't know how many procedure calls the length procedure makes. So, let's write our own.

```

(define$ mylength
  (lambda (lst)
    (if (null? lst)
        0
        (+ 1 (mylength (cdr lst))))))

```

- Add this procedure to your file for this lab.
- Replace the call to length in largest-of-list-3 with a call to mylength.

c. Repeat your analysis. What do you learn?

## For Those with Extra Time

### Extra 1: Yet Another largest-of-list

Here is yet another version of `largest-of-list` that uses a recursive kernel to keep track of the largest element found so far and the position in the list.

```
(define$ largest-of-list-4
  (letrec ((kernel
            (lambda (largest lst pos len)
              (if (>= pos len)
                  largest
                  (kernel (max largest (my-list-ref lst pos))
                        lst
                        (+ pos 1)
                        len))))))
    (lambda (lst)
      (kernel (car lst) lst 1 (length lst))))
(define$ my-list-ref
  (lambda (lst pos)
    (if (zero? pos)
        (car lst)
        (my-list-ref (cdr lst) (- pos 1)))))
```

a. Do you expect this to be more or less efficient than `largest-of-list-2`? Why or why not?

b. Confirm your answer experimentally.

### Extra 2: Iota, Revisited

You may recall the `iota` procedure from a previous reading. Given a positive integer, `n`, as a parameter, `iota` creates a list of all the values between 0 and `n-1`. Here are two common definitions of `iota`, one that uses a helper and one that does not.

```
(define$ iotal
  (lambda (n)
    (if (zero? n)
        null
        (myappend (iotal (- n 1)) (list (- n 1))))))
(define$ iota2
  (letrec ((kernel (lambda (n)
                    (if (zero? n)
                        null
                        (cons (- n 1) (kernel (- n 1)))))))
    (lambda (n)
      (reverse-2 (kernel n)))))
```

- a. Which do you expect to be more efficient? Why?
  - b. Check your results experimentally.
  - c. See if you can figure out a formula for the number of procedure calls made in each version for a given input size.
- 

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.