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).
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.

```scheme
(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.

```scheme
> (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.

```scheme
> (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.

```scheme
> (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.

<table>
<thead>
<tr>
<th>List Length</th>
<th>r1: Calls to <code>myappend</code></th>
<th>r1: Total calls</th>
<th>r2: Calls to <code>kernel</code></th>
<th>r2: Total calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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))))

a. 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.

b. 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.

c. 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.

d. 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))))

a. Does the change seem to have an effect? If so, what is that effect?

b. 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))))))

a. Add this procedure to your file for this lab.

b. 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.

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

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 \text{iota} procedure from a previous reading. Given a positive integer, \(n\), as a parameter, \text{iota} creates a list of all the values between 0 and \(n-1\). Here are two common definitions of \text{iota}, one that uses a helper and one that does not.

\[
\text{(define$ iota1}
\text{ (lambda (n)
\text{  (if (zero? n)
\text{    null
\text{    (myappend (iota1 (- n 1)) (list (- n 1))))))
\text{ (define$ iota2}
\text{ (letrec ((kernel (lambda (n)
\text{    (if (zero? n)
\text{      null
\text{      (cons (- n 1) (kernel (- n 1)))))
\text{    (lambda (n)
\text{      (reverse-2 (kernel n))))))

5
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.