MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall 2007

Recitation 10
Quiz 1 Review

Quiz 1

Quiz 1 will be held next wednesday from 7:30pm-9:30pm in 50-340.

Example quizzes from prior years, and recitation notes can be found at:

`http://people.csail.mit.edu/jastr/6001/fall07/`

Queues

A data structure that stores elements in order. Elements are `enqueue`d onto the tail of the queue. Elements are tt dequeued from the head of the queue. Thus, the first element enqueued is also the first element dequeued (FIFO, first-in-first-out). The `head` operation is used to get the element at the head of the queue.

```
(head (enqueue 5 (empty-queue)))
;Value: 5

(define q (enqueue 4 (enqueue 5 (enqueue 6 (empty-queue)))))

(head q)
;Value: 6

(head (dequeue q))
;Value: 5
```

1. Decide on an implementation for queues, then draw a box-and-pointer represention of the value of `q` as defined above.

2. Write `empty-queue`.

   ```
   (define (empty-queue)
   ```

   Order of growth in time? Space?

3. Write `enqueue`; a procedure that returns a new queue with the element added to the tail.

   ```
   (define (enqueue x q)
   ```

   Can you do this with `fold-right`?

   Order of growth in time? Space?

4. Write `dequeue`; a procedure that returns a new queue with the head element removed.

   Order of growth in time? Space?

5. Write `head`; a procedure that returns the value of the head element.

   Order of growth in time? Space?

MinMax

The Following questions come from a 6.001 quiz 1 circa 1995.

For the questions below, assume that time is counted in the number of comparisons ($<$ or $>$ tests) done in the execution of the program on a sequential computer. By space we mean the additional space used by the process to keep track of the computation (We are not counting the space used by the data lists manipulated by the computation.)

1. We were presented with the following program on the quiz handout, probably written by the infamous Louis Reasoner, for computing both the minimum and the maximum of a list of numbers:

```
(define (min-and-max list-of-numbers)
  (cons (min-list list-of-numbers)
        (max-list list-of-numbers)))

(define (min-list list-of-numbers)
  (if (null? (cdr list-of-numbers))
      (car list-of-numbers)
      (if (< (car list-of-numbers)
             (min-list (cdr list-of-numbers)))
          (car list-of-numbers)
          (min-list (cdr list-of-numbers)))))

(define (max-list list-of-numbers)
  (if (null? (cdr list-of-numbers))
      (car list-of-numbers)
      (if (> (car list-of-numbers)
             (max-list (cdr list-of-numbers)))
          (car list-of-numbers)
          (max-list (cdr list-of-numbers)))))
```

It was noticed that this program is an enormously slow way of computing the desired result.

In the worst case, just how inefficient is this program? Circle the true statements about this program in the list given below.

(a) It is not worse than twice as slow as the best program for this problem.
(b) Its order of growth in time is the square of the length of the input list.
(c) Its order of growth in time is exponential in the length of the input list.
(d) The program generates an iterative process.
(e) The program takes no more than a constant amount of space to execute.
(f) The program takes space of order the length of the input list to execute.

2. A staff programmer, Betir Waye, claimed that the program can be improved by the introduction of a LET form in the definition of the procedure that finds the minimum (and symmetrically the one that finds the maximum).

```
(define (min-list list-of-numbers)
  (if (null? (cdr list-of-numbers))
      (car list-of-numbers)
      (let ((m (min-list (cdr list-of-numbers))))
        (if (< (car list-of-numbers) m)
            (car list-of-numbers)
            m))))
```

For the questions below, assume that time is counted in the number of comparisons ($<$ or $>$ tests) done in the execution of the program on a sequential computer.

Just how much better is this program? Circle the true statements about this program in the list given below.

(a) It is worse than Louis's program.

(b) Its order of growth in time is linear in the length of the input list.

(c) The time taken is approximately proportional to the square root of the time taken by Louis's program.

(d) The program generates an iterative process.

(e) The program generates a recursive process.

(f) The program takes no more than a constant amount of space to execute.

(g) The program takes space of order the length of the input list to execute.

3. Evan Betta suggested that we can do even better as follows:

```
(define (min-list list-of-numbers)
  (define (accum rest min)
    (cond ((null? rest) min)
          ((< (car rest) min) (accum (cdr rest) (car rest)))
          (else (accum (cdr rest) min))))
  (accum (cdr list-of-numbers)
         (car list-of-numbers)))
```

For the questions below, assume that time is counted in the number of comparisons ($<$ or $>$ tests) done in the execution of the program on a sequential computer.

Just how much better is this program? Circle the true statements about this program in the list given below.

(a) It is worse than Betir's program.

(b) Its order of growth in time is linear in the length of the input list.

(c) The time taken is proportional to the square root of the time taken by Betir's program.

(d) The program generates an iterative process.

(e) The program generates a recursive process.

(f) The program takes no more than a constant amount of space to execute.

(g) The program takes space of order the length of the input list to execute.

4. After these modifications, the whole program is:

```
(define (min-and-max list-of-numbers)
  (cons (min-list list-of-numbers)
        (max-list list-of-numbers)))

(define (min-list list-of-numbers)
  (define (accum rest min)
    (cond ((null? rest) min)
          ((< (car rest) min) (accum (cdr rest) (car rest)))
          (else (accum (cdr rest) min))))
  (accum (cdr list-of-numbers)
```

```
         (car list-of-numbers)))

(define (max-list list-of-numbers)
  (define (accum rest max)
    (cond ((null? rest) max)
          ((> (car rest) max) (accum (cdr rest) (car rest)))
          (else (accum (cdr rest) max))))
  (accum (cdr list-of-numbers)
         (car list-of-numbers)))
```

The big-wheel programmers Ben Bitdiddle and Alyssa P. Hacker appeared to testify. Alyssa pointed out that because the procedures MAX-LIST and MIN-LIST are so similar, the situation fairly cries out for some abstraction. She quickly sketched a neat plan:

```
(define (min-and-max list-of-numbers)
  (cons ((min-or-max-list <) <blob1>)
        <blob2>))

(define (min-or-max-list comparison)
  (define (accum rest min-or-max)
    <blob3>
    )
  (lambda (list-of-numbers)
    (accum (cdr list-of-numbers)
           (car list-of-numbers))))
```

Unfortunately, Alyssa dropped a slice of pepperoni pizza on her notes. In the spaces provided below, please show us the details that were obliterated with cheese:

<blob1>:

<blob2>:

<blob3>:

5. Always practical, and never to be outdone, Ben proposed the following procedure, which accumulates the minimum and the maximum simultaneously:

```
(define (min-and-max list-of-numbers)
  (if (null? (cdr list-of-numbers))
      (cons (car list-of-numbers)
```

```
          (car list-of-numbers))
      (let ((mmr (min-and-max (cdr list-of-numbers))))
        (cond ((< (car list-of-numbers) (car mmr))
               (cons (car list-of-numbers) (cdr mmr)))
              ((> (car list-of-numbers) (cdr mmr))
               (cons (car mmr) (car list-of-numbers)))
              (else mmr)))))
```

Ben's procedure generates a recursive process. In the space provided below, write an equivalent procedure that generates an iterative process:

6. Alyssa had the last word with the following mysterious procedure:

```
(define (min-and-max list-of-numbers)
  (define (scan rest continue)
    ;; continue = (lambda (min max) ...)
    (if (null? (cdr rest))
        (continue (car rest) (car rest))
        (scan (cdr rest)
              (lambda (min-cdr-rest max-cdr-rest)
                (cond ((< (car rest) min-cdr-rest)
                       (continue (car rest) max-cdr-rest))
                      ((> (car rest) max-cdr-rest)
                       (continue min-cdr-rest (car rest)))
                      (else
                       (continue min-cdr-rest max-cdr-rest)))))))
  (scan list-of-numbers cons))
```

Circle the true statements about this program in the list given below.

(a) For any procedure that takes two numbers F:
`(scan (list 7 3 1 9) F) = (F 1 9)`

(b) `(scan (list 2) +) = 4`

(c) `(scan (list 2 3) +) = 4`

(d) `(scan (list 2 3 5) *) = (scan (list 3 4 5 6 7) +)`

(e) For any list of numbers X and for any procedure that takes two numbers F:

`(scan (scan X list) F) = (scan X F)`