

MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
 Department of Electrical Engineering and Computer Science  
 6.001—Structure and Interpretation of Computer Programs  
 Spring Semester, 2000

**Quiz I**

**Closed Book – one sheet of notes**

Please write clear and concise answers to the questions in the spaces provided in this booklet. You may use scratch paper, but the spaces we provide are the **only** places we will look at when grading.

Your name (0 points):

“Your name” (2 points):

**Question 1: (14 points)**

For each of the following expressions or sequences of expressions, state what value is returned as the result of evaluating the final expression in each set. Each part is independent—in other words, for each part, state what value would be returned if that expression or sequence of expressions were the only expressions evaluated in a newly initialized scheme system.

If the evaluation results in an error, please state in general terms what kind of error (e.g. you might write “error: wrong type of argument to procedure”). If the evaluation returns a procedure, simply write the word **procedure**.

1. (+ 2 (\* 3 (+ 1 3)))

2. ((lambda (arg) arg) (lambda () 5))

3. ((lambda (arg) (arg)) (lambda () 5))

4. ((lambda (a \* b) (\* a b)) 5 + 7)

5. ((if (> 1 2) 1 2) #t)

```
6. (define (foo x) (+ x 1))
   (define (bar y) (y 5))
   ((bar foo) 2)
```

```
error: can't apply object 6
```

```
7. (define (foo x) (+ x 1))
   (define (bar y) (lambda (z) (y z)))
   ((bar foo) 2)
```

```
3
```

### Question 2: (22 points)

Suppose you are asked to create a small database system to handle vote counting for the Republican primaries. We are going to store information as a record, that includes information about the voting district as well as a candidate and the number of votes in that district for that candidate. Here are a pair of constructors:

```
;Type: Symbol, VoteData --> Pair<Symbol, VoteData>
(define (make-record district vote-data)
  (cons district vote-data))

;Type: Symbol, number --> Pair<Symbol, number>
(define (make-votes name how-many)
  (list name how-many))
```

So for example, we might have

```
(define test (make-record 'townhall
  (make-votes 'mccain 500)))
```

#### Part a:

We need selectors `record-district` and `record-data` to complete the abstraction for a district record. For example:

```
(record-district test)
;Value: townhall
```

Define `record-district`

```
(define record-district car)
```

Define `record-data`

```
(define record-data cdr)
```

#### Part b:

Define selectors `votes-name` and `votes-how-many` to complete the abstraction for a data record. For example:

```
(votes-how-many (record-data test))
;Value: 500
```

Define `votes-name`

```
(define votes-name car)
```

Define `votes-how-many`

```
(define votes-how-many cadr)
```

**Part c:** Now suppose that we are provided with a list of entries (called `results`) of the following form:

```
((<district-1> (<votes-1> <votes-2> <votes-3>))
 (<district-2> (<votes-4> <votes-5>))
 .....
 )
```

In other words, `results` is a list, each of whose elements is a list. Each element is a list of two parts, a district name and a list of “votes” abstractions. We want to convert this list in two stages.

In the first stage, we want to convert to a list of entries, where each entry is a list of records obtained by attaching the district name to each vote abstraction.

For example, if the first element of `results` were

```
(townhall ((mccain 500) (bush 340) (keyes 10)))
```

then in our new list, this would become

```
((townhall mccain 500) (townhall bush 340) (townhall keyes 10))
```

– assuming of course that we use the constructors defined in this question!

Complete the following expression to accomplish this:

```
(define new-results
  (map (lambda (elt)
        <INSERT-A>
        results))
```

What should be provided for `<INSERT-A>`?

```
(map (lambda (x) (make-record (car elt) x)) (cadr elt))
```

**Part d:**

In the second stage, we want to convert this list of lists into a single list. In other words, if `new-results` has the form

```
((<record-1> <record-2> <record-3>)
 (<record-4> <record-5>)
 ...
)
```

then we want to convert this to

```
(<record-1>
 <record-2>
 <record-3>
 <record-4>
 <record-5>
 ...
)
```

Complete the following expression by providing code for <INSERT-B> and <INSERT-C>:

```
(accumulate <INSERT-B> <INSERT-C> new-results)
```

```
append
```

```
<INSERT-C>
```

```
'() or nil
```

### Question 3: (18 points)

Suppose we are given a list of integers, each of which lies between -10000 and 10000. We want to compute the minimum and maximum value in the list. Consider the following code:

```
;Type: LIST<number> -->Pair<number, Pair<number, nil>>
(define (max&min lst)
  (if (null? lst)
      (list -10000 10000)
      (let ((rest-m&m (max&min (cdr lst))))
        (list (max (car lst) (car rest-m&m))
              <INSERT>))))
```

#### Part a:

Write the code that goes in place of <INSERT> so that the procedure `max&min` correctly computes the maximum and minimum values in a list of number.

```
(min (car lst) (cadr rest-m&m))
```

#### Part b:

The method of computing the extremes of a list used by the code on this page is:

Choice A: a recursive process

Choice B: an iterative process

Choice C: cannot tell based on the information given

Choice:  A

**Part c:** Another way of computing the extremes of a list is to use the following procedures to compute the maximum and minimum separately:

```
(define (my-max lst) ; type: list<number> -> number
  (max-help lst -10000))

(define (max-help lst todate)
  (if (null? lst) todate
      (max-help (cdr lst) (max (car lst) todate))))

(define (my-min lst) ; type: list<number> -> number
  (min-help lst 10000))

(define (min-help lst todate)
  (if (null? lst) todate
      (min (car lst) (min-help (cdr lst) todate))))

(define (my-max&min lst)
  (list (my-max lst) (my-min lst)))
```

We are interested in the order of growth of the processes generated by these procedures. For order of growth in time, we are interested in the number of max or min operations, and for order of growth in space, we are interested in the number of deferred operations.

For each procedure below, circle the order of growth:

my-max	space	<b>CONSTANT</b>	linear	exponential	logarithmic
	time	constant	<b>LINEAR</b>	exponential	logarithmic
my-min	space	constant	<b>LINEAR</b>	exponential	logarithmic
	time	constant	<b>LINEAR</b>	exponential	logarithmic
my-max&min	space	constant	<b>LINEAR</b>	exponential	logarithmic
	time	constant	<b>LINEAR</b>	exponential	logarithmic

**Question 4: (20 points)**

We want to create a procedure to compute a histogram of an input list (which we will assume is a list of symbols). This means that your procedure should return a list of how many times each symbol occurs in the input list. For example:

```
(compute-histogram '(foo bar bar foo baz foo bat))
;Value: ((foo 3) (bar 2) (baz 1) (bat 1))
```

indicating that the symbol `foo` occurred 3 times in the input, the symbol `bar` occurred twice, and so on.

For this question you may find it helpful to use `map`, `filter` or `accumulate`, copies of which are on the last page of the quiz.

**Part a:**

To begin, we will create a procedure that counts the number of times a particular symbol occurs in a list, e.g., `(count 'foo '(foo bar bar foo baz foo bat))` should return the value 3:

```
(define (count x L)
  (cond ((null? L) 0)
        ((<INSERT-A> x (car L))
         <INSERT-B>)
        (else
         <INSERT-C>)))
```

Write the code for `<INSERT-A>` needed to complete this definition.

eq?

Write the code for <INSERT-B> needed to complete this definition.

(+ 1 (count x (cdr L)))

Write the code for <INSERT-C> needed to complete this definition.

(count x (cdr L))

**Part b:** Suppose instead we use the following procedure

```
(define (count x L)
  (accumulate + 0
    (map <INSERT-D>
      L)))
```

Write the code for <INSERT-D> needed to complete this definition.

(lambda (elt) (if (eq? x elt) 1 0))

**Part c:**

Now, we want to use `count` to compute the complete histogram. Below is a template for the procedure:

```
(define (compute-histogram L)
  (cond ((null? L) '())
        (else (cons <INSERT-E>
                    (compute-histogram
                     (filter <INSERT-F> <INSERT-G>))))))
```

Write the code for <INSERT-E> needed to complete this definition.

(list (car L) (count (car L) L))

Write the code for <INSERT-F> needed to complete this definition.

(lambda (x) (not (eq? x (car L))))

Write the code for <INSERT-G> needed to complete this definition.

(cdr L)

**Question 5: (15 points)**

For each of the following expressions, draw the associated box-and-pointer diagram.

**Part a:** `(cons 'a (list (list 'b) (list 'c)))`

See your TA for details

**Part b:** `(list 'a '(b c))`

See your TA for details

**Part c:**

```
(let ((temp (list 1 2)))
  (list temp temp))
```

See your TA for details

### Question 6: (9 points)

The procedure `list-ref` returns an element of a list, e.g. `(list-ref '(1 2 3 4) 2)` returns the value 3. It would be nice to have a similar feature for trees. Write a procedure `tree-ref` that takes as input a tree (a list of lists) and a list of integers, and returns the element specified by selecting the appropriate branch from the list of integers. For example:

```
(define tree '((1 2) 3)
              ((5 6) 4)
              7))
```

```
(tree-ref tree '(2))
;Value: 7
```

```
(tree-ref tree '(1 0))
;Value: (5 6)
```

Thus, we are using zero-based indexing, and we are applying the indices in top-down order. Note that `(tree-ref tree '())` would return the value of `tree`.

Complete the following definition:

```
(define (tree-ref tree indices)
  (cond ((null? indices)
        <INSERT-A>)
        (else
         (tree-ref <INSERT-B> <INSERT-C>))))
```

Complete <INSERT-A>

Complete <INSERT-B>

Complete <INSERT-C>