

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

**Quiz I**

**Closed Book – one sheet of notes**

Here is a sample quiz with the solutions included, for your reference. Note that in many cases there may be multiple acceptable answers.

**Question 1: (16 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 built-in procedure, simply write **primitive**. If the evaluation returns a user-created procedure, simply write **compound**.

As well, for all non-error values, indicate the type of the returned value.

`(+ 1 (* 2 (/ 3 4)))`

**5/2; NUMBER**

+

**PRIMITIVE; NUMBER, NUMBER ...  $\mapsto$  NUMBER**

```
(define foo (lambda (x) (* x x)))
foo
```

**COMPOUND; NUMBER  $\mapsto$  NUMBER**

```
((lambda (arg) arg) (lambda () 5))
```

**COMPOUND; VOID  $\mapsto$  NUMBER**

```
((lambda (arg) (arg)) (lambda () 'arg))
```

**ARG; SYMBOL**

```
(define sq (lambda (x) (* x x)))
((lambda (op) (op 5)) 'sq)
```

**ERROR: NOT A PROCEDURE**

```
(define (foo x) (* x 2))
(define (bar x) (lambda (y) (x y)))
((bar foo) 3)
```

**6; NUMBER**

```
(define x 5)
(let ((x 3)
      (y (* x x)))
  (let ((x 1))
    (list x y)))
```

**(1 25); PAIR<1, PAIR<25, NIL >>****Question 2: (8 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 (defined as a list whose elements are either integers or other  $n$ -argument trees) and a list of integers, and returns the element specified by selecting the appropriate branch from the list of integers. 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`.

For example:

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

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

(tree-ref tree '(1 0))
;Value: 4

(tree-ref tree '(0 0 1))
;Value: 2

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

Here is the correct code (parts 9, 10, 11).

```
(define (tree-ref tree indices)
  (cond ((null? indices)
        TREE
        (else
         (tree-ref (LIST-REF TREE (CAR INDICES)) (CDR INDICES))))))
```

**Question 3: (10 points)**

On the attached sheet, you will find several labelled box-and-pointer diagrams. For each of the questions below, indicate which diagram corresponds to the value returned by the expression by listing the number of that diagram, or indicate **none**, if the correct box-and-pointer diagram is not among those shown on the attached sheet.

```
(list 'a '(b c))
```

**answer: 3**

```
(cons 'a (list (list 'b) '(c)))
```

**answer: 4**

```
(let ((temp (list 'a 'b 'c)))
  (list temp temp))
```

**answer: 6**

```
(let ((temp (list 'b 'c)))
  (list (cons 'a temp) temp))
```

**answer: 5**

```
(list (list 'a 'b 'c) (list 'b 'c))
```

**answer: 9; 8 was given half credit****Question 4: (12 points)**

You have been hired by one of the political parties to help answer email questions from voters. The idea is that a message, represented as a list of symbols, such as:

```
(define test-message '(could you please tell me your position on tuition and on education))
```

would be processed by your system as follows:

- detect keywords in the message, by comparing each symbol to a list of special symbols,
- look up the standard response (also represented as a list of symbols) for each keyword from a database of platform positions,
- assemble the responses together into a single list.

Given a **message** as a list of symbols, and a list of keywords, **keys**, the following procedure should return a list of the keywords that are contained in the message:

```
(define (find-key-words message keys)
  (filter <INSERT A> message))
```

For example,

```
(find-key-words test-message '(abortion education tuition taxes health social-security))
;Value: (tuition education)
```

17. What expression should be used in place of <INSERT A>? You may use the procedure **contains-key?**, which is defined as:

```
(define (contains-key? elt lst)
  (cond ((null? lst) #f)
        ((eq? elt (car lst)) #t)
        (else (contains-key? elt (cdr lst)))))
```

Answer: (LAMBDA (WORD) (CONTAINS-KEY? WORD KEYS))

For a given keyword, **key**, and a representation of associations of keys and responses, **platform**, assume that the procedure **lookup** can be used to extract the appropriate position from the database. We will assume that a **platform** is a list of elements, each of which is a data abstraction, consisting of a pairing of a **key** and a **position**. The appropriate data selectors for an element of the platform are **get-key** and **get-position**. Thus, (**lookup** **key** **platform**) will return a position, as a list of symbols, that is the desired response for the keyword, **key**. For example, we might have:

```
(lookup 'education my-platform)
;Value: (I believe that education is very important)
```

```
(lookup 'tuition my-platform)
;Value: (colleges should not charge students tuition, rather it should be free to anyone)
```

Using these abstractions, plus **map**, **filter** and/or **accumulate** (which are defined at the end of the quiz), complete the definition below to generate a **single** list, that merges together the responses to all the keys found in a message.

Here is a completed version of the code, (parts 18-20)

```
(define (generate-response message platform)
  (let ((possible-keys (MAP GET-KEY PLATFORM)))
    (let ((cues (find-key-words message possible-keys)))
      (accumulate APPEND NIL
        (map (lambda (cue) (lookup cue platform))
          cues))))))
```

For example,

```
(generate-response test-message my-platform)
;Value:(colleges should not charge students tuition, rather it should be free to anyone
I believe that education is very important)
```

### Question 5: (20 points)

Suppose we are given a list of integers, and we want to sort the list into decreasing order. One way to do this is to find the biggest element in the list, put that at the front of the new list, and then sort the remainder of the list, and so on. Here is a sorting function based on this:

```
(define (sort lst)
  (if (null? lst)
      '()
      (let ((next (max&rest (car lst) '() (cdr lst))))
        (cons (car next)
              (sort (cdr next))))))
```

To complete this procedure, we need to implement `max&rest`. The inputs are the current value of the biggest element, the set of elements (other than the biggest) already examined, and the elements yet to be examined. The output should be a list with the largest element at the front, and the remaining elements in arbitrary order afterwards.

Here is a completed version of the code (parts 21, 22, 23):

```
(define (max&rest best rest todo)
  ;; best is largest so far, rest is other things already looked at,
  ;; todo is list of things still to examine
  (if (null? todo)
      (cons best rest)
      (max&rest (MAX BEST (CAR TODO))
                (CONS (MIN BEST (CAR TODO)) REST)
                (CDR TODO))))
```

**24.** The method of computing the largest value and the rest of the list used by `max&rest` is:

**Choice B: an iterative process**

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 primitive or built-in operations, and for order

of growth in space, we are interested in the number of deferred operations (not the number of pairs generated).

For each procedure below, choose from one of the following options:

Option	Order of growth
A	$\Theta(1)$
B	$\Theta(n)$
C	$\Theta(2^n)$
D	$\Theta(\log n)$
E	$\Theta(n^2)$

**25.** Order of growth in space of `max&rest`.  $\Theta(1)$

**26.** Order of growth in time of `max&rest`.  $\Theta(n)$

**27.** Order of growth in space of `sort`.  $\Theta(n)$

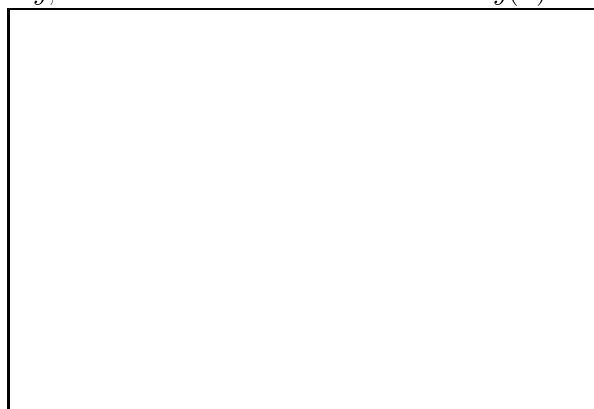
**28.** Order of growth in time of `sort`.  $\Theta(n^2)$

**Question 6: (24 points)**

A standard way to represent a line is with a slope and an intercept, that is a line (generally) can be represented in the form

$$y(x) = ax + b$$

where the constants  $a$  and  $b$  define the slope and the intercept. Thus, as we vary  $x$ , we sweep out the corresponding values of  $y$ , and a combination of an  $x$  and a  $y(x)$  defines a point on the line.



**29.** Assuming that `make-vector` is a constructor for a data abstraction, with associated selectors `x-coord` and `y-coord`, where for example,

```
(define make-vector list)
(define x-coord car)
(define y-coord cadr)
```

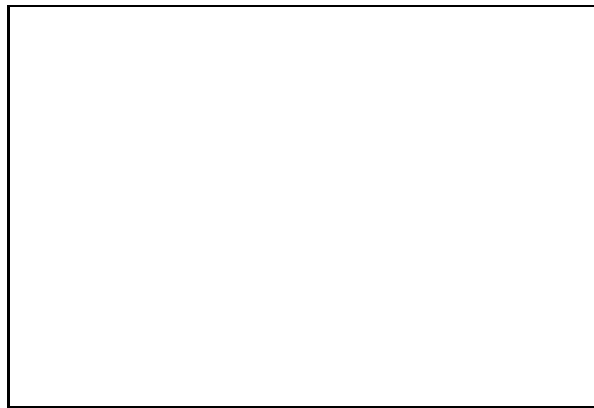
complete the definition below so that

```
(define myline (make-line 10 20))
(define example-point (make-vector 30 (myline 30)))
```

would create an example line, and an example point on that line.

```
(define (make-line a b)
  (LAMBDA (X) (+ (* A X ) B))
  )
```

A second way to represent a line is with a tangent vector  $\mathbf{t}$  (that is a vector that points along the line) and an offset  $\mathbf{o}$  (that is, a point on the line). Then, a point on the line is given by picking an  $\alpha$  and computing  $\alpha\mathbf{t} + \mathbf{o}$  where the  $+$  denotes vector addition.



Assume that the operation `add-vectors` takes two vectors as input, and adds them together component-wise, and that the operation `scale-vector` takes a number and a vector, and produces a new vector with each component multiplied by the number.

**30.** Implement a procedure, `line-to-tangent` that takes a line (as computed with `make-line`) and produces a vector along the tangent of the line. This can be done by selecting two points along the line, and creating the vector between them by subtracting one vector from the other.

```
(define (line-to-tangent line)
  (LET ((PT1 (MAKE-VECTOR 0 (LINE 0)))
        (PT2 (MAKE-VECTOR 1 (LINE 1))))
    (ADD-VECTORS PT1 (SCALE-VECTOR -1 PT2))))
```

**31.** Using `vector-add`, `vector-scale`, `line-to-tangent`, `make-vector`, create a new procedure that converts a line to the tangent form. You may pick an arbitrary point on the line to be the offset. Thus, for example:

```
(define mytangentline (tangent myline))

(mytangentline 0)
;Value: (0 20)
```

```
(mytangentline 10)
;Value: (-10 -80)

(define (tangent line)
  (let ((tng (line-to-tangent line))
        (base (make-vector 0 (line 0))))
    (LET ((TNG (LINE-TO-TANGENT LINE))
          (BASE (MAKE-VECTOR 0 (LINE 0))))
      (LAMBDA (ALPHA)
        (VECTOR-ADD (VECTOR-SCALE ALPHA TNG) BASE)))
    )
```

**Question 7: (12 points)**

Remember the definition of derivative, given by

```
(define dx 0.0001)

(define (deriv f)
  (lambda (x) (/ (- (f (+ x dx))
                    (f x))
                 dx)))
```

Generalize this to create a procedure that takes  $n$ th derivatives, (parts 32 and 33):

```
(define (n-deriv f n)
  (if (= n 0)
      F
      (DERIV (N-DERIV F (- N 1)))))
```

Another way of accomplishing this is to use:

```
(define (repeat f n)
  (if (= n 0)
      <INSERT C>
      <INSERT D>))
```

so that, for example, we have

```
(define mytest (repeat deriv 2))
((mytest square) 3)
;Value: 1.9999999878

((mytest square) 5)
;Value: 1.9999998102
```

Below are possible choices for the two insertions. List all that will correctly complete this definition, or write **none** if none of them will correctly complete this definition.

**Option A.**

INSERT C: x

INSERT D: (lambda (x) (f ((repeat f (- n 1)) x)))

**Option B.**

INSERT C: x

INSERT D: (f ((repeat f (- n 1)) x))

**Option C.**

INSERT C: (lambda (x) x)

INSERT D: (lambda (x) (f ((repeat f (- n 1)) x)))

**Option D.**

INSERT C: (lambda (x) x)

INSERT D: (lambda (x) ((repeat f (- n 1)) (f x)))

**Option E.**

INSERT C: (lambda (x) x)

INSERT D: (f (repeat f (- n 1)))

**Option F.**

INSERT C: (lambda (x) x)

INSERT D: (lambda (x) ((f (repeat f (- n 1))) x))

correct answers are: C, D

**BACKGROUND INFORMATION—THIS PAGE CONTAINS NO QUESTIONS**

**List Procedures**

```
(define (filter pred lst)
  (cond ((null? lst) nil)
        ((pred (car lst))
         (cons (car lst) (filter pred (cdr lst))))
        (else (filter pred (cdr lst)))))
```

```
(define (map op lst)
  (if (null? lst)
      nil
      (cons (op (car lst))
            (map op (cdr lst)))))
```

```
(define (accumulate op init lst)
  (if (null? lst)
      init
      (op (car lst)
          (accumulate op init (cdr lst)))))
```

**BACKGROUND INFORMATION—THIS PAGE CONTAINS NO QUESTIONS**