MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1999

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 if you need, but the spaces we provide are the **only** places we will look at when grading.

Your name: SOLUTION

Circle your *Section Number* and your *Tutor* below:

Section	Time	Location	Rec. Instructor	Tutors
1	10:00	13-4101	Randy Davis	Jim Waldrop
2	11:00	13-4101	Randy Davis	Kyle Ingols
3	10:00	36-155	Daniel Jackson	Jim Waldrop/Mike Phillips
4	11:00	36-156	Daniel Jackson	Robbin Chapman
5	11:00	26-204	Tony Eng	Robbin Chapman/Kyle Ingols
6	12:00	26-204	Tony Eng	Aileen Tang
7	1:00	26-204	Leslie Kaelbling	Mieszko Lis
8	2:00	26-210	Leslie Kaelbling	Mike Phillips
9	12:00	36-153	Tomas Lozano-Perez	Aileen Tang/Mieszko Lis

Please do not write below this line:

Problem	Value	Grade	Grader
1	12		
2	8		
3	6		
4	10		
5	14		
6	20		
Total	70		

Question 1: (12 points – 2 for each answer box)

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 of this question 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. For example, you might write "error: wrong type of argument to procedure." If the evaluation returns a procedure, simply write the word `procedure`.

1.
```
(define (x val) 10)
x
```

procedure

2.
```
((if test 10 20) #t)
```

error: unbound variable test

3.
```
((lambda (x) x) (lambda () 3))
```

procedure

4.
```
((lambda (x y) (+ x y)) (+ 1 3))
```

error: wrong number of arguments to procedure

5.
```
(define x 10)
(let ((x 7))
  (let ((y 3))
    (+ x y)))
```

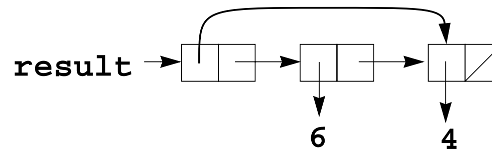
10

6.
```
(define x 10)
(let ((x 7)
      (y (lambda (y) (+ x y))))
  (y 3))
```

13

Question 2: (8 points)

Consider the following box and pointer diagram:



Part a: (2 points)

What does the box and pointer object shown above (the value of `result`) print as?

```
Choice A: ((4) (6 4))

Choice B: ((4) 6 4)

Choice C: (4 (6 4))

Choice D: ((4) 6 (4))

Choice E: ((4) 6 . 4)
```

Which choice corresponds to the printed representation of `result`? | B |

Part b: (6 points)

The object bound to `result` could be created by one or more of the following expressions.

```
Choice F: (list 4 (list 6 4))

Choice G: (cons (list 4) (list 6 4))

Choice H: (let ((y (list 4)))
            (list y (list 6 y)))

Choice I: (let ((z (list 4)))
            (cons z (cons 6 z)))

Choice J: (let ((x (list 6 4)))
            (cons (cdr x) x))

Choice K: (let ((w (list 6 4)))
            (list (cdr w) w))
```

Which choice or choices would return an object with the same box and pointer structure as `result` above? List all choices that produce the `result` structure. | I, J |

Question 3: (6 points – 2 for each answer box)

Consider the following code:

```
(define (transform lst)
  (map (lambda (x)
         (let ((two-x (* 2 x))
               (three-x (* 3 x)))
           (+ two-x three-x)))
       lst))
```

Part a:

What is the value printed when you evaluate `(transform (list 1 2 3))`?

(5 10 15)

Part b:

How many lambda expressions are there in these six lines of code (including lambda expressions that are hidden by syntactic sugar).

3

Part c:

How many times is the `lambda` expression starting on the second line, `(lambda (x) ...)`, *evaluated* (as opposed to *applied*) when you evaluate `(transform (list 1 2 3))`?

1

Question 4: (10 points – 2 for each answer box)

The arithmetic mean of numbers $x_1, x_2, \ldots, x_n$ is $((x_1 + x_2 + \ldots + x_n)/n)$.

Consider the following code:

```
; type: list<number> -> number
(define (mean-1 lst)
  (if (null? lst) 0
      (let ((result (mean-helper lst)))
        (/ (car result) (cadr result)))))

; type: list<number> -> pair<number, pair<number, null>>
(define (mean-helper lst)
  (if (null? lst)
      (list 0 0)
      (let ((info-from-rest (mean-helper (cdr lst))))
        (list (+ (car lst) (car info-from-rest))
              <EXP A>
              ))))
```

Part a:

Write the code that goes in place of `<EXP A>` so that the procedure `mean-1` correctly computes the arithmetic mean of a list of numbers.

```
(+ 1 (cadr info-from-rest))
```

Part b:

The method of computing the arithmetic mean used by the code on this page is:

`Choice A`: an iterative algorithm

`Choice B`: a recursive algorithm

`Choice C`: cannot determine from the information given

Choice: `B`

Part c: An alternate way of computing the mean is to use the following procedures to compute the list sum and length separately:

```
(define (sum lst)                 ; type: list<number> -> number
  (define (help lst sofar)
    (if (null? lst) sofar
        (help (cdr lst) (+ (car lst) sofar))))
  (help lst 0))

(define (length lst)              ; type: list<anytype> -> number
  (define (help lst sofar)
    (if (null? lst) 0
        (+ 1 (help (cdr lst) sofar))))
  (help lst 0))

(define (mean-2 lst)
  (if (null? lst) 0
      (/ (sum lst) (length lst))))
```

If n is the length of the list, we are interested in the order of growth of the processes generated by these procedures.

`Choice A:` a **linear iterative** process, i.e. requires an order of growth in time (the number of addition operations) that is linear in n for large n, and an order of growth in storage space (the stack up of deferred operations) that is constant for large n.

`Choice B:` a **linear recursive** process, i.e. requires an order of growth in time that is linear in n for large n, and an order of growth in storage space that is also linear in n for large n.

`Choice C:` a **recursive** process that requires order of growth greater than linear in time or more than linear in space for large n.

`Choice D:` an **iterative** process that requires greater than linear time order of growth in n but constant space for large n.

`Choice E:` a **constant space and time** process which requires computation that does not grow in proportion to n.

`Choice F:` cannot determine from the information given.

1. What is the order of growth for the process resulting from the `sum` procedure?

   Choice: `A`

2. What is the order of growth for the process resulting from the `length` procedure?

   Choice: `B`

3. What is the order of growth for the process resulting from the `mean-2` procedure?

   Choice: `B`

Question 5: (14 points – two for each answer box)

In this problem, we will be using the list higher order procedures `map`, `filter`, and `accumulate`. You can find the code for these procedures on the last page of this quiz if you need them. Note: the various parts of this question are independent; you should be able to work each part even without the correct solution to previous parts.

Part a:

Alyssa P. Hacker is working to define a procedure `member?` that tells us if a number is a member (an element of) a given list. The procedure takes a number and a list of numbers, and returns `#t` if the number is in the list and `#f` if not. E.g.

```
(member? 1 (list 1 2 3)) ==> #t
(member? 1 (list 2 3)) ==> #f
(member? 1 nil) ==> #f
```

Complete the definition of Alyssa's procedure below to implement `member?`. Your answer should not use any higher order procedures.

```
(define (member? x L)
  (cond ((null? L) #f)
        ((= x (car L)) #t)
        (else   <EXP1>   )))
```

<EXP1>: `(member? x (cdr L))`

Part b:

Ben Bitdiddle thinks about his `member?` procedure a little differently, and seeks to implement it using the higher order list procedures. Complete his definition below. (Note: if you have not seen the `(or ...)` special form before, it is described on the last page of this handout.)

```
(define (member? x L)
  (accumulate (lambda (x y) (or x y))
              #f
              (map (lambda (item) (= item x))
                   <EXP2>)))
```

<EXP2>: `L`

Part c:

Carl N. Kons happens upon Ben's code, and complains that mapping will force the `=` comparison to be performed on every item, even if `x` appears very early in the list. Carl suggests a more subtle use of the `accumulate` procedure, together with the special form `or`, that attempts to avoid this:

```
(define (member? x L)
  (accumulate (lambda (elt in-rest)
                 (or (= elt x) in-rest))
               #f
               L))
```

In terms of the number of times the `=` procedure is called, is Carl's `member?` procedure more efficient than Ben's (i.e. will Carl's procedure sometimes avoid calling `=` on some items in the list)? Answer **Yes** or **No**.

No

Part d:

Assuming that we have a working `member?` procedure that complies with the specification given in part a, we want to implement the `intersection` procedure. This procedure takes as arguments two lists of numbers, `L1` and `L2`, and returns a list of those numbers in `L1` that *also* appear in list `L2`. (As a side note, `intersection` also outputs numbers in the same order as they appeared in the first list). E.g.

```
(intersection (list 1 2 3) (list 2 3 4)) ==> (2 3)
(intersection (list 1 2 3) (list 4 5 6)) ==> nil
```

Ben writes `intersection` again using available higher order list procedures. Complete his definition:

```
(define (intersection L1 L2)
  (filter (lambda (elt) <EXP3>) L1))
```

`<EXP3>`: `(member? elt L2)`

Part e:

Ben and Alyssa are asked to write a procedure `running-sum` that, given a list of numbers, produces a new list that contains, as each element in the new list, the sum of all corresponding elements up to that point in the original list. E.g.

```
(running-sum (list 1 3 9 2))
==> (1 4 13 15)
; That is, 4 = 1+3, 13 = 1+3+9, and 15 = 1+3+9+2

(running-sum (list 2 2 2 2 2))
==> (2 4 6 8 10)
```

Ben tries to write `running-sum` using the list higher order procedures, but gives up. Alyssa decides that `running-sum` is most easily written directly rather than using `map`, `filter`, or `accumulate`. Complete her definition below.

```
(define (running-sum L)
  (define (run L sofar)
    (if (null? L)
        <EXP4>
        (let ((sum (+ (car L) sofar)))
          (cons <EXP5> <EXP6>))))
  (run L 0))
```

<EXP4>: `nil`

<EXP5>: `sum`

<EXP6>: `(run (cdr L) sum)`

Question 6: (20 points – two for each answer box)

Alyssa P. Hacker is a TA for 6.001. One of the problem sets involves drawing pictures on the screen, so she writes a data abstraction for RGB colors and gives it to the students. Here is a small part of the data abstraction that she wrote.

```
; RGB colors are triples (r,g,b), where r is the amount of the red
; component in the color, g is the amount of the green component, and
; b is the amount of the blue component.  Each of the three amounts is
; an integer between 0 and 255, inclusive.

; To ensure that color values have components that are valid amounts,
; the procedure make-color uses the procedure restrict on each of its
; arguments.

(define make-color
  (lambda (red-amt green-amt blue-amt)
    (list (restrict red-amt)
          (restrict green-amt)
          (restrict blue-amt))))

(define red-val   (lambda (color) (car color)))
(define green-val (lambda (color) (cadr color)))
(define blue-val  (lambda (color) (caddr color)))

; Restrict converts from an arbitrary number to a valid color amount.
; This is an internal procedure that should not be used by clients
; of the data abstraction.

(define (restrict x)
  (cond ((< x 0)   0)
        ((> x 255) 255)
        (else      (round-to-integer x))))  ; round x to the nearest integer

; [remainder of the data abstraction not shown here]
```

Alyssa also writes a graphics library, one procedure of which is

```
; type: point, point, color -> undef
(define paint-rectangle-on-screen
  (lambda (lower-left-corner upper-right-corner color-to-paint) ...))
```

Part a:

For each of the procedures listed below, answer

- what is the type of this procedure?
- what is it?

Choice A: a constructor of the data abstraction
Choice B: an accessor of the data abstraction
Choice C: an operation of the data abstraction
Choice D: none of the above

1. `make-color:`

   What is its type? `number, number, number -> Color`

   What is it? Choice: `A`

2. `green-val:`

   What is its type? `Color -> number`

   What is it? Choice: `B`

3. `restrict:`

   What is its type? `number -> number`

   What is it? Choice: `D`

Part b:

Louis Reasoner, a student in 6.001, is a "user" of the above color data abstraction. At one place in his problem set solution, he needs a procedure which computes a color that is half as bright as another color. He looks up in a reference book how to do this, and it says that reducing the amount of all components (red, green, and blue) of a color by half will reduce its apparent brightness by half.

Louis implements this computation by writing the following operation layered on the color data abstraction above.

```
; type: color -> color
(define half-as-bright (lambda (color) (map (lambda (val) (/ val 2)) color)))
```

Louis tests this procedure once by painting two large rectangles, A and B, side by side on the computer screen. Rectangle A uses the color produced by evaluating `(make-color 100 150 200)`. Rectangle B uses the color produced by evaluating `(half-as-bright (make-color 100 150 200))`. Rectangle B appears half as bright as rectangle A. He concludes that the routine works and submits it as part of his problem set.

What is printed if Louis evaluates `(half-as-bright (make-color 100 150 200))`

`(50 75 100)`

Part c:

Alyssa grades Louis' problem set and marks off points for `half-as-bright`. Possible reasons for this are given below. For each statement below, answer whether the statement is true or false (more than one of the following may be true):

1. `Half-as-bright` violates the abstraction barrier.

   `True`

2. Although the test case works now, if it is run again in the future (after Alyssa gives the class an improved version of her data abstraction for colors) rectangle B might not appear to be half as bright as rectangle A on the monitor.

   `True`

3. Although the test case works now, in the future (after Alyssa gives the class an improved version of her data abstraction for colors) evaluating `(half-as-bright (make-color 100 150 200))` might give an error and halt.

   `True`

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)))))
```

Or Special Form

The `or` special form is a special form because it has unusual rules for the order of evaluation of its arguments. Specifically, argument expressions are evaluated in order, from left to right, until the first non-false value is encountered, at which point that value will be returned. Any remaining argument expressions are not evaluated. If none of the arguments evaluates to a non-false value, then the `or` expression returns `#f`.

For example:

```
(or (= 2 2) (< 2 1)) ==> #t

(or (= 2 2) (> 2 1)) ==> #t

(or (< 5 0) (* 6 3) (= 5 6)) ==> 18

(or (< 5 0) (= 5 6)) ==> #f

(or 1 2 3 4) ==> 1

(or #f 2 3 4) ==> 2
```

BACKGROUND INFORMATION—THIS PAGE CONTAINS NO QUESTIONS