

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

Quiz II

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 (we've attached some sheets at the end of the exam), but the spaces we provide are the **only** places we will look at when grading.

Your name:

Question 1: (14 points)

For each of the following sets of expressions, evaluate the expressions in order (assuming nothing about prior evaluations in the environment) and state the value of the final expression. 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**. In the evaluation returns a value that might be different depending on which Scheme system you use to evaluate the expressions, write the words **implementation dependent**.

1. (+ 1 '(+ 2 3))

2. (+ x (/ 3 0))

3. (let ((x '(1 (2 3))))
 (cond ((eq? (cdr x) '(2 3))
 'one)
 ((eq? x '(1 (2 3)))
 'two)
 ((eq? x x)
 'three)
 (else 'none))))

```
4. (define f '(a))
   (define g '(b c))
   (set-cdr! (cdr g) f)
   (list f g)
```

```
((a) (b c a))
```

```
5. (define f '(a b c))
   (set! (cdr f) '(d))
   f
```

```
Error: not a symbol
```

```
6. (define x 2)
   (define test
     (let ((x x))
       (lambda (y) (* y x))))
   (set! x 5)
   (test 10)
```

```
20
```

```
7. (define x 2)
   (define test
     (lambda (y)
       (let ((x x))
         (* y x))))
   (set! x 5)
   (test 10)
```

```
50
```

Question 2: (14 points)

Bert has decided to create a small system to help compute his taxes. This system will just keep track of simple things like income, deductions, amount paid, and amount owed. Initially he chooses to represent the tax information in a table structure, with the following interface:

```
make-table: void -> Table<anytype, anytype> ;creates a new table
table-put!: Table<k,v>, k, v -> undef        ;inserts a key and value into the table
table-get: Table<k,v>, k -> (v | null)      ;returns value associated with a key, if any
```

In this table, he can keep information by associating a value with a name (e.g. INCOME and 20000).

He decides to implement the table using two lists – one for the keys and one for the values. The two lists are synchronized, so that corresponding parts of the `values` list match the same parts of the `keys` list.

```
(define bert-tag 'bert)
(define (make-bert-table) (cons bert-tag (cons '() '())))

(define (keys tbl) (cadr tbl))
(define (values tbl) (caddr tbl))

(define (table-get tbl key)
  (define (search keys values)
    (cond ((null? keys) #f)
          ((eq? (car keys) key) (car values))
          (else (search (cdr keys) (cdr values)))))
  (search (keys tbl) (values tbl)))
```

Initially, Bert decides not to worry about whether there are multiple copies of keys and associated values in his list, relying on his `get` method to find the most recent key-value pairing put into the table. Here is his `put` procedure:

```
(define (table-put! tbl key val)
  (set-car! (cdr tbl) (cons key (keys tbl)))
  (set-cdr! (cdr tbl) (cons val (values tbl)))
  'done)
```

Part a: Upon reflection, Bert decides he would be better off replacing an entry if there already exists one in the table. Thus, there will be at most one entry in the `keys` list for each symbol. He writes an outline for a more careful procedure.

Here is the correct code:

```
(define (table-put-careful! tbl key val)
  (define (loop keys values)
    (cond ((null? keys)
           (TABLE-PUT! TBL KEY VAL))
          ((eq? (car keys) key)
           (SET-CAR! VALUES VAL))
          (else (loop (cdr keys) (cdr vals)))))
  (loop (keys tbl) (values tbl)))
```

Part b: Ernie has also been working on a system to help compute taxes, but chooses to use association lists. Here is his implementation (– looks like Ernie has been going to 6.001 lectures).

```
(define ernietag 'ernie)
(define (make-table-ernie) (cons ernietag nil))
(define ernie-alist cdr)
(define (ernie-get tbl key)
  (find-assoc key (ernie-alist tbl)))
(define (ernie-put! tbl key val)
  (set-cdr! tbl (add-assoc key val (ernie-alist tbl))))
```

Ernie also decides to implement association lists (which are manipulated by `find-assoc` and `add-assoc` using:

```
association = Pair<key, Pair<value, null>>
association_list = List<association>
```

Since Bert and Ernie want to file joint returns, they need to convert between tables. Here is a procedure to convert an “ernie” table to a “bert” table.

Below is the correct code:

```
(define (convert-ernie-to-bert ernie-tbl)
  (let ((bert-tbl (make-bert-table))
        (alist (ernie-alist ernie-tbl)))
    (let ((new-keys (MAP CAR ALIST))
          (new-vals (MAP CADR ALIST)))
      (SET-CDR! BERT-TBL (CONS NEW-KEYS NEW-VALS))))))
```

Question 3 (17 points):

We want to add to our language the ability to keep track of how often a procedure is used with particular arguments. That is, we would like to have procedures that remember how many times they were called with each different value of their argument. For simplicity, we will restrict ourselves to procedures of one integer argument. We will keep a *history* of the calls of the procedure as a list of entries, where each entry is a pair of the argument and the number of calls. Thus:

```
history = List<entry>
entry = Pair<integer, integer>
```

In particular, we have an abstraction for an entry:

```
(define (make-entry arg)
  (cons arg 1) ;; create an entry when argument is first used
)

(define entry-argument car)

(define entry-number-of-calls cdr)

(define (update-entry! entry)
  <blob0>
)
```

Updating an entry means that we have called the procedure with an argument again (we will assume that it has already been called at least once) and we need to record this fact in our history. Fill in the missing blob:

```
<blob0> (set-cdr! entry (+ 1 (cdr entry)))
```

Below is a template for creating such procedures. Note that we want to separate out the process of updating the history from the process of using the actual procedure. Thus `increment-history` should update an entry for an argument if there is an entry already in the history, and otherwise should add a new entry to the history. The application of the “historical” procedure itself should be handled by the `me` procedure below.

Below is the correct code with the blobs filled in:

```
(define (make-history proc)
  (let ((history (list 'history)))
    (define (increment-history m lst)
      (cond ((null? lst)
             (SET-CDR! HISTORY (CONS (MAKE-ENTRY M 1) (CDR HISTORY))))
            ((= m (entry-argument (car lst)))
             (UPDATE-ENTRY (CAR LST)))
            (else
             (INCREMENT-HISTORY M (CDR LST)))))
    (define (me m)
      (cond ((number? m)
             (INCREMENT-HISTORY M (CDR HISTORY))
             (PROC M))
            ((eq? m 'reset)
             (SET! HISTORY (LIST 'HISTORY)))
            ((eq? m 'show)
             HISTORY)))
    me))
```

```
(define my-cube (make-history (lambda (x) (* x x))))
;Value: ‘my-square --> #[compound-procedure 20 me]’
```

```
(my-cube 3)
;Value: 27
```

```
(my-cube 3)
;Value: 27
```

```
(my-cube 5)
;Value: 125
```

```
(my-cube 'show)
;Value: (history (5 . 1) (3 . 2))
```

```
(my-cube 'reset)
;Value: (history)
```

```
(my-cube 'show)
;Value: (history)
```

Question 4: (17 points)

Sometimes a procedure may have to do a lot of computation to determine the value associated with applying it to an argument. As a consequence, it is sometimes convenient to create a “memoized” version of the procedure. Under this approach, the procedure keeps track of what arguments it has been called on, and what value was returned for that call. Thus, if the procedure is called a second time with the same argument, it can simply return the previously computed value. Below is a procedure for memoization:

```
(define memoize
  (lambda (proc)
    (let ((val-1st '()))
      (lambda (arg)
        (let ((lookup (member arg val-1st)))
          (if lookup
              (cadr lookup)
              (let ((new-val (proc arg)))
                (set! val-1st (cons (list arg new-val) val-1st))
                new-val)))))))

(define (member arg lst)
  (cond ((null? lst) #f)
        ((= arg (caar lst)) (car lst))
        (else (member arg (cdr lst)))))
```

Now suppose we evaluate the following expressions:

```
(define my-square (memoize (lambda (x) (* x x))))

(my-square 4)
```

Now we are going to ask you to draw the environment diagrams generated by these expressions, using diagram fragments we will provide. At the end of the exam is a tear off sheet, with a partially completed environment diagram. In this diagram, we have marked procedure objects as P1, P2, etc., environments as E1, E2, etc. You should EITHER complete this diagram directly on these fragments, OR you should do your own environment diagram on a separate sheet, then copy the labels for the fragments. In either case, answer the questions below based on this diagram.

First, for each procedure object, P1 through P7, identify, if possible, the environment pointer of the procedure, by circling the appropriate answer. If the appropriate environment is not shown, mark that spot in the table.

Procedure	Environment Pointer									
P1	E1	E2	E3	E4	E5	E6	E7	<input checked="" type="checkbox"/> GE	Not Shown	
P2	E1	E2	E3	E4	E5	E6	E7	<input checked="" type="checkbox"/> GE	Not Shown	
P3	E1	E2	E3	<input checked="" type="checkbox"/> E4	E5	E6	E7	GE	Not Shown	
P4	E1	E2	E3	E4	E5	E6	E7	<input checked="" type="checkbox"/> GE	Not Shown	
P5	E1	E2	E3	E4	E5	<input checked="" type="checkbox"/> E6	E7	GE	Not Shown	
P6	<input checked="" type="checkbox"/> E1	E2	E3	E4	E5	E6	E7	GE	Not Shown	
P7	E1	E2	<input checked="" type="checkbox"/> E3	E4	E5	E6	E7	GE	Not Shown	

Second, for each environment frame, indicate which environment is the enclosing environment for that frame, by circling the appropriate answer. If the appropriate environment is not shown, mark that spot in the table.

Frame	Enclosing Environment									
E1	E1	E2	E3	<input checked="" type="checkbox"/> E4	E5	E6	E7	GE	None	Not Shown
E2	E1	E2	<input checked="" type="checkbox"/> E3	E4	E5	E6	E7	GE	None	Not Shown
E3	<input checked="" type="checkbox"/> E1	E2	E3	E4	E5	E6	E7	GE	None	Not Shown
E4	E1	E2	E3	E4	E5	<input checked="" type="checkbox"/> E6	E7	GE	None	Not Shown
E5	E1	E2	E3	E4	E5	E6	E7	<input checked="" type="checkbox"/> GE	None	Not Shown
E6	E1	E2	E3	E4	E5	E6	E7	<input checked="" type="checkbox"/> GE	None	Not Shown
E7	E1	E2	E3	E4	E5	E6	E7	<input checked="" type="checkbox"/> GE	None	Not Shown
GE	E1	E2	E3	E4	E5	E6	E7	GE	<input checked="" type="checkbox"/> None	Not Shown

Finally, for each of the following variables, indicate to what it is bound in the diagram (possibilities include one of the procedure objects, P1, ..., P6, a number, a symbol, a list of numbers or symbols (which you can simply write as list structure), an environment, E1, ..., E6, GE. Do this in terms of the values associated with these variables after **all** the expressions have been evaluated.

Frame	Variable	Bound value
E1	arg	4
E2	new-val	16
E3	lookup	nil or #f
E4	val-lst	((4 16))
E5	x	4
E6	proc	P2
GE	memoize	P4
GE	my-square	P3

Question 5 (15 points)

One of the techniques we have explored during the term is the idea of capturing local state within a procedure. Suppose we want to create a set of accumulators, that is, procedures which begin with an initial value, and after each application of the procedure to a single argument, returns the sum of the values to which it has been applied. For example:

```
(define a (make-accumulator 5))
;Value: a

(a 10)
;Value: 15

(a 10)
;Value: 25

(define b (make-accumulator 10))
;Value: b

(b 10)
;Value: 20
```

For each of the following definitions, indicate whether this definition will support the behavior illustrated above. If your answer is no, indicate why it fails.

Part a:

```
(define make-accumulator
  (lambda (init)
    (let ((ttl init))
      (lambda (arg)
        (set! ttl (+ ttl arg))
        ttl))))
```

Does this work correctly? YES

Part b:

```
(define make-accumulator
  (let ((ttl 0))
    (lambda (init)
      (set! ttl init)
      (lambda (arg)
        (set! ttl (+ ttl arg))
        ttl))))
```

Does this work correctly? NO

If not, explain why not.

The state variable `ttl` is shared by all objects constructed with this accumulator

Part c:

```
(define (make-changer)
  (lambda (arg)
    (set! ttl (+ ttl arg))))

(define make-accumulator
  (lambda (init)
    (let ((ttl init)
          (make-changer))))
```

Does this work correctly? NO

If not, explain why not.

The procedure created by `make-changer` points to the wrong environment

Question 6 (23 points)

Attached are a set of procedures from an Object Oriented System similar to the one that you used in Project 2. The procedure `ask` gets a method from an object (perhaps with inheritance) and applies it to that object. The procedure `delegate` gets a method from one object and applies it to a separate object. `Delegate-to-all` delegates a message to all members of a list of objects.

Part A: First, complete the definition of the `Appliance` class. An electrical appliance can be switched to the state `'ON` or to `'OFF`, and has a fuse which can be made to blow via the message `'BLOW-FUSE`. One can also ask if the appliance is currently `ON?`, which should return a boolean.

```
;; Example desired behavior
(define my-toaster (make-appliance))
(ask my-toaster 'ON?) ==> #f
(ask my-toaster 'SWITCH 'ON)
(ask my-toaster 'ON?) ==> #t
(ask my-toaster 'BLOW-FUSE)
  Bang! Fuse Blown!
(ask my-toaster 'ON?) ==> #f
```

Below is the completed code:

```

(define (make-appliance)
  (let ((switch-state 'OFF)
        (fuse-blown #f))
    (lambda (message)
      (case message
        ((SWITCH)
         (LAMBDA (SELF ON-OFF)
          (IF (NOT FUSE-BLOWN)
              (SET! SWITCH-STATE ON-OFF))))
        ((ON?)
         (LAMBDA (SELF)
          (EQ? SWITCH-STATE 'ON)))
        ((BLOW-FUSE)
         (LAMBDA (SELF)
          (COND (FUSE-BLOWN (DISPLAY ‘‘FUSE ALREADY BLOWN’))
                (ELSE (SET! FUSE-BLOWN #T)
                       (SET! SWITCH-STATE 'OFF)
                       (DISPLAY ‘‘BANG! FUSE BLOWN!’))))))
        (else (no-method))))))

```

Part B: Now we implement some appliances. A **Blender** is an appliance that also has a speed setting. A **TV** is an appliance that also has a channel setting. Complete the definitions of these classes:

```

;; Desired behavior
(define my-blender (make-blender))
(ask my-blender 'SET-SPEED 5)
(ask my-blender 'SPEED) ==> 5
(ask my-blender 'ON?) ==> #F

(define (make-blender)
  (let ((speed 0)
        (appliance (make-appliance)))
    (lambda (message)
      (case message
        ((SET-SPEED)
         (LAMBDA (SELF NEW-SPEED)
          (SET! SPEED NEW-SPEED)))
        ((SPEED) ...) ; some method for returning the speed
        (else (get-method message appliance))))))

(define (make-TV)
  (let ((channel 0)
        (appliance (make-appliance)))
    (lambda (message)
      (case message
        ((SET-CHANNEL)
         (LAMBDA (SELF NEW-CHAN)
          (SET! CHANNEL NEW-CHAN)))

```

```
((CHANNEL) ...) ; some method for returning the channel
  (else (get-method message appliance))))))
```

Part C: Ronco, Inc. has noticed that a lot of people watch TV while in the kitchen, and has decided to market the “BL-TV” – which is both a blender AND a TV! Only one problem: whenever the speed of the blender goes above 3, and the TV part and blender part are both on, the fuses in both parts blow. Implement this behavior in the new class “BL-TV” which inherits from the blender and TV classes.

In the space below, create an implementation of a constructor for BL-TVs. The resulting objects should support three messages: `blow-if-overload`, `switch`, and `set-speed`, as well as inheriting or delegating to methods from appropriate internal objects.

```
(define (make-BL-TV)
  (let ((BL (make-blender))
        (TV (make-tv)))
    (lambda (message)
      (case message
        ((BLOW-IF-OVERLOAD) ;; internal method for overload
         (LAMBDA (SELF)
           (IF (AND (DELEGATE BL SELF 'ON?)
                    (DELEGATE TV SELF 'ON?)
                    (> (DELEGATE BL SELF 'SPEED) 3))
              (DELEGATE-TO-ALL (LIST BL TV) SELF 'BLOW-FUSE))))
        ((SWITCH)
         (LAMBDA (SELF ON-OFF)
           (DELEGATE-TO-ALL (LIST BL TV) SELF 'SWITCH ON-OFF)
           (ASK SELF 'BLOW-IF-OVERLOAD)))
        ((SET-SPEED)
         (LAMBDA (SELF NEW-SP)
           (DELEGATE BL SELF 'SET-SPEED NEW-SP)
           (ASK SELF 'BLOW-IF-OVERLOAD)))
        (else (get-method message TV BL))))))
```

BACKGROUND INFORMATION—THIS PAGE CONTAINS NO QUESTIONS

```
; An OOP system

(define (delegate to from message . args)
  (let ((method (get-method message to)))
    (if (method? method)
        (apply method from args)
        (error "No method" message))))

(define (ask object message . args)
  (apply delegate object object message args))

(define (get-method message preferred . others)
  (define (loop objs)
    (let ((method (get-method-from-object
                  message (car objs)))
          (rest (cdr objs)))
      (if (or (method? method) (null? rest))
          method
          (loop rest))))
  (loop (cons preferred others)))

(define (get-method-from-object message object)
  (object message))

(define (no-method) '(NO-METHOD))

(define (delegate-to-all to-list from message . args)
  (foreach
   (lambda (to-whom)
     (apply delegate to-whom from message args))
   to-list))

(define (foreach proc lst)
  (cond ((null? lst) nil)
        (else (proc (car lst))
                (foreach proc (cdr lst)))))
```

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