

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

Quiz II

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: (18 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**.

1.

`'(+ 1 2 3)`

`(+ 1 2 3)`

2.

```
(define x '(a))
(define y '(b (c)))
(set-cdr! y (cadr y))
(list x y)
```

`((a) (b c))`

3.

```
(define x '(a))
(define y '(b (c)))
(set! (cdr y) (cadr y))
(list x y)
```

error, not a symbol

4.

```
(define a 3)
(define foo
  (let ((a a))
    (lambda (x) (* x a))))
(set! a 5)
(foo 10)
```

30

5.

```
(define a 3)
(define foo
  (lambda (x)
    (let ((a a))
      (* x a))))
(set! a 5)
(foo 10)
```

50

6.

```
(define a 3)
(define b 'a)
(list 'a a 'b b)
```

(a 3 b a)

Question 2: (23 points)

Consider the following simple message passing system:

```
(define (make-container)
  (let ((contents '()))
    (lambda (msg)
      (case msg
        ((add) (lambda (thing) (set! contents (cons thing contents))))
        ((remove) (lambda (thing) (set! contents (delete! thing contents)))))))))
;; assume that delete! is some appropriately defined procedure, which removes
;; thing from contents, and returns a new contents

(define (make-safe combo)
  (let ((inner (make-container))
        (state 'unlocked))
    (lambda (msg)
      (case msg
        ((unlock) (lambda (input)
                     (if (eq? state 'unlocked)
                         'already-unlocked
                         (if (eq? input combo)
                             (set! state 'unlocked)
                             'wrong-combo))))
        ((lock) (lambda (input)
                   (if (eq? state 'locked)
                       'already-locked
                       (if (eq? input combo)
                           (set! state 'locked)
                           'wrong-combo))))
        ((remove) (lambda (thing) (if (eq? state 'unlocked)
                                       ((inner 'remove) thing)
                                       'safe-is-locked)))
        ((add) (lambda (thing) (if (eq? state 'unlocked)
                                    ((inner 'add) thing)
                                    'safe-is-locked)))))))
```

Suppose we evaluate the following sequence of expressions.

```
(define mine (make-safe 'eric))
((mine 'add) 'money)
```

We want 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 the labels used on OUR diagram!!**

First, for each procedure object, P1 through P4, identify, if possible, the environment pointer of the procedure (i.e. one of GE, E1, ..., E8). If the appropriate environment is not shown, write “not shown”.

7. To what does the environment pointer of P1 point?

E3

8. To what does the environment pointer of P2 point?

GE

9. To what does the environment pointer of P3 point?

GE

10. To what does the environment pointer of P4 point?

E5

Second, for each environment frame, indicate which environment is the enclosing environment for that frame. If the environment has no enclosing environment, write “none”. If the environment has an enclosing environment, but it is not one of those shown in the diagram, write “not shown”

11. What is the enclosing environment for E1?

E3

12. What is the enclosing environment for E2?

GE

13. What is the enclosing environment for E3?

E4

14. What is the enclosing environment for E4?

GE

15. What is the enclosing environment for E5?

E2

16. What is the enclosing environment for E7?

E5

17. What is the enclosing environment for GE?

none

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 (say what number)

- a symbol (say what specific symbol)
- a list of numbers or symbols (which you can simply write as list structure, but be specific),
- an environment, E1, ..., E6, GE.

Do this in terms of the values associated with these variables after **all** the expressions have been evaluated.

18. What is the variable `mine` in the global environment bound to?

P1

19. What is the variable `make-safe` in the global environment bound to?

P3

20. What is the variable `make-container` in the global environment bound to?

P2

21. What is the variable `inner` in environment E3 bound to?

P4

22. What is the variable `combo` in environment E4 bound to?

eric

23. What is the variable `contents` in environment E5 bound to?

(money)

Question 3: (24 points)

We want to add a new kind of expression to the meta-circular evaluator, called an `until`. Its syntax is as follows:

```
(until test  exp1 exp2 ... expn)
```

and its behavior is to first evaluate the `test` expression. If the value is true, the `until` expression returns the symbol `done`. Otherwise it evaluates each of the expressions `exp1`, `exp2`, ... `expn` in turn, then repeats this entire process.

First, we want to add this kind of expression to the evaluator by using a syntactic transformation. Thus, we have added the following clause to `eval`:

```
((until? exp)
 (eval (until->if exp) env))
```

Assume that the following abstractions are provided:

```
(define (until? exp) (tagged-list? exp 'until))
(define (until-test exp) (cadr exp))
(define (until-body exp) (caddr exp))
```

24. Using these abstractions, write a definition for `until->if`:

```
(define (until->if exp)
  (list 'if
        (until-test exp)
        '(quote done)
        (append (cons 'begin
                      (until-body exp))
                (list exp))))
```

25. Now, suppose instead that we add the following clause to `eval`.

```
((until? exp) (eval-until exp env))
```

Complete the definition of `eval-until`. Do this without using syntactic transformations. You may find it convenient to define a helper procedure to do this, though this is not necessary.

```
(define (eval-until exp env)
  (if (eval (until-test exp) env)
      'done
      (begin (eval-sequence (until-body exp) env)
              (eval exp env))))
```

Question 4 (15 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 tagged list of entries, where each entry is a pair of the argument and the number of calls. Thus:

```
history = Pair<symbol, 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.

26. What expression should be used in place of <blob0> (see above definition)? Do this in such a manner that any variable pointing to `entry` will still point to the same list structure after evaluation of `(update-entry! entry)`

```
(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.

```

(define (make-history proc)
  (let ((history (list 'history)))
    (define (increment-history m lst)
      (cond ((null? lst)
              <blob1>)
            ((= m (entry-argument (car lst)))
              <blob2>)
            (else
              <blob3>)))
      (define (me m)
        (cond ((number? m)
                (increment-history m (cdr history))
              (proc m))
              ((eq? m 'reset)
                (set! history (list 'history))
                history)
              ((eq? m 'show)
                history)))
          me))

(define my-cube (make-history (lambda (x) (* x x x))))
;Value: 'my-cube --> #[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)) ;; NOTE THE ORDER OF THIS LIST

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

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

```

Based on this observed behavior, you should be able to complete the missing blobs:

- 27.** What expression should be used in place of <blob1>?
(set-cdr! history (cons (make-entry m 1) (cdr history)))
- 28.** What expression should be used in place of <blob2>?
(update-entry (car lst))
- 29.** What expression should be used in place of <blob3>?
(increment-history m (cdr lst))

Question 5 (points):**

***** At the end of the quiz you will find the code for our object oriented system. Also, we have included a scratch sheet on which you may want to trace out environment diagrams for this question. *****

The following object oriented class hierarchy loosely copies politicians' tendencies to "mis-speak".

```
(define (make-A thing)
  (lambda (msg) (case msg
                  ((MENTION) (lambda (self) thing))
                  ((DESCRIBE) (lambda (self)
                                (newline)
                                (display "type-A, have thing ")
                                (display thing)
                                'A-done))
                  ((EXAGGERATE) (lambda (self)
                                   (newline)
                                   (ask self 'DESCRIBE)
                                   'A-really-done))
                  (else (error "type-A: unknown message" msg))))))

(define (make-B thing)
  (let ((A (make-A 'frog)))
    (lambda (msg) (case msg
                    ((MENTION) (lambda (self) thing))
                    ((DESCRIBE) (lambda (self)
                                    (newline)
                                    (display "type-B, have thing ")
                                    (display (ask self 'MENTION))
                                    'B-done))
                    ((EXAGGERATE) (lambda (self)
                                       (ask a 'EXAGGERATE)
                                       'B-really-done))
                    (else (get-method msg A))))))

(define (make-C thing)
  (let ((A (make-A 'bird))
        (B (make-B thing)))
    (lambda (msg) (case msg
                    ((MENTION) (find-method 'MENTION A B))
                    (else (find-method msg B A))))))

(define (make-D thing)
  (let ((A (make-A 'lizard))
        (C (make-C thing)))
    (lambda (msg) (case msg
                    ((MENTION) (find-method 'MENTION A C))
                    ((DESCRIBE) (lambda (self) (delegate-to-all (list C A) self 'DESCRIBE)))
                    ((EXAGGERATE) (lambda (self) (delegate-to-all (list C A) self EXAGGERATE)))
                    (else (find-method msg A C))))))
```

Assume we make the above definitions, and then evaluate

```
(define D (make-D 'paper))
```

What gets printed (either by the "display" procedure or by the read-eval-print loop) for each of the following expressions? (You may find it helpful to sketch out an environment model.)

30.

```
(ask D 'MENTION)
```

;Value: lizard

31.

```
(ask D 'DESCRIBE)
```

type-B, have thing lizard

type-A, have thing lizard

;Value: (b-done a-done)

32.

```
(ask D 'EXAGGERATE)
```

type-A, have thing frog

type-B, have thing lizard

type-A, have thing lizard

;Value: (b-really-done a-really-done)

```

; Our familiar OOP system

(define (delegate to from message . args)
  (apply-method to from message args))

(define (apply-method in-object for-object message args)
  (let ((method (get-method message in-object)))
    (cond ((method? method)
           (apply method for-object args))
          ((eq? in-object for-object)
           (error "No method for" message 'in (ask in-object 'NAME)))
          (else (error "Can't delegate" message
                       "from" (ask for-object 'NAME)
                       "to" (ask in-object 'NAME))))))

(define (delegate-to-all to-list from message . args)
  (map (lambda (who) (apply-method who from message args))
       to-list))

(define (ask object message . args)
  (apply-method object object message args))

(define (get-method message object) ; single-inheritance
  (object message))

(define (find-method message . objects) ; multiple-inheritance
  (define (try objects)
    (if (null? objects)
        (no-method)
        (let ((method (get-method message (car objects))))
          (if (not (eq? method (no-method)))
              method
              (try (cdr objects))))))
    (try objects))

(define (method? x)
  (cond ((procedure? x) #T)
        ((eq? x (no-method)) #F)
        (else (error "Object returned this non-message:" x))))

(define no-method
  (let ((tag (list 'NO-METHOD)))
    (lambda () tag)))

```