

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
 Department of Electrical Engineering and Computer Science
 6.001—Structure and Interpretation of Computer Programs
 Spring Semester, 2002

Quiz II – Solutions

Listed below are example answers for each of the questions. There are alternative answers in some cases, these are simply intended to demonstrate the most common answer.

Part 1: (20 points)

We want to construct a data abstraction called a **doubly-linked list**. This abstraction acts in part like a list – one can extract the first element of the list, or one can move to the next element in the list. Different from normal lists, however, in a doubly-linked list one can return (in constant time) to the previous element in the list. We will build this data structure using message passing objects as the elements (note that these are not our standard OOPS objects). Below are COMPLETED procedure for creating elements, and for putting an element at the front of a list:

```
(define (make-unit value)
  (let ((next #f) ; pointer to next element in list
        (previous #f)) ; pointer to previous element in list
    (define me
      (lambda (msg)
        (cond
          ((eq? msg 'what) VALUE) ; return value of this element
          ((eq? msg 'set-next!)
           (LAMBDA (ARG) (SET! NEXT ARG))) ; change next element, return unspecified
          ((eq? msg 'set-previous!)
           (LAMBDA (ARG) (SET! PREVIOUS ARG))) ; change previous element, return unspecified
          ((eq? msg 'next) next) ; return next element
          ((eq? msg 'previous) previous) ; return previous element
          (else (error "Can't handle this message" msg))))
      me))

(define (add-to-list val front-of-list)
  (let ((new (make-unit val)) ; make element
        ((new 'set-next!) front-of-list) ; put at front of list
        ((front-of-list 'set-previous!) new) ; create double link
        new)) ; return element, front of list
```

Now suppose we want to add the ability to insert an element in the middle of a doubly-linked list, while preserving the linked nature of the list. We will assume that `insert` (see below) is used strictly for side-effect and does not need to return the beginning of the list. We will also assume that n is less than the length of `dlist`. Here is a COMPLETED procedure for accomplishing this.

```
(define (insert val dlist n)
```

```
(if (= n 0)
    (let ((new (make-unit val)))
        ((NEW 'SET-NEXT!) DLIST)
        ((NEW 'SET-PREVIOUS!) (DLIST PREVIOUS))
        ((NEW 'NEXT) 'SET-PREVIOUS!) NEW)
        ((NEW 'PREVIOUS) 'SET-NEXT!) NEW)
    )
    (insert val (DLIST 'NEXT) (- n 1))))
```

Part 2: (20 points)

Consider the following procedure:

```
(define (memoize proc)
  (let ((diary '()))
    (lambda (arg)
      (let ((previous (assoc arg diary)))
        (if previous
            (cadr previous)
            (let ((temp (proc arg)))
              (set! diary (cons (list arg temp) diary))
              temp))))))
```

Assume that `assoc` searches its second argument (which we assume is a list of lists), comparing its first argument to the first element of each component of that list, using `equal?`. If it finds a match, it returns that sublist, if not it returns `nil`.

For example:

```
(assoc 5 '((5 25) (6 36)))
;Value: (5 25)
```

```
(assoc 3 '((5 25) (6 36)))
;Value: #f
```

```
(assoc 5 '())
;Value: #f
```

Suppose we now evaluate the following sequence of expressions:

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

(my-sq 5)
```

We want you to draw the environment diagram generated by these expressions, using diagram fragments that we provide. Attached to the exam is a tear-off sheet, with some fragments from an 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 onto your diagram (we actually recommend the latter). **In either case, answer the questions about the environment based on the labels used on OUR diagram!!**

First, for each procedure object, P1 through P6, identify, if possible, the environment pointer of the procedure (i.e. one of GE, E1, ..., E6). If the appropriate environment is not shown, write “not shown”.

Question 6. To what does the environment pointer of P1 point? **GE**

Question 7. To what does the environment pointer of P2 point? **E6**

Question 8. To what does the environment pointer of P3 point? **GE**

Question 9. To what does the environment pointer of P4 point? **E3**

Question 10. To what does the environment pointer of P5 point? **E2**

Question 11. To what does the environment pointer of P6 point? **E4**

Second, for each environment frame, indicate which environment is the enclosing environment for that frame. If the appropriate environment is not shown, write “not shown”. If there is no environment, write “none”.

Question 12. What is the enclosing environment for E1? **E3**

Question 13. What is the enclosing environment for E2? **E6**

Question 14. What is the enclosing environment for E3? **E2**

Question 15. What is the enclosing environment for E4? **GE**

Question 16. What is the enclosing environment for E5? **GE**

Question 17. What is the enclosing environment for E6? **E4**

For each of the following questions, choose from among these possibilities:

- one of the procedure objects, P1, ..., P6,
- a number (say what number)
- a symbol (say what specific symbol)
- a list of numbers or symbols or procedure objects (which you must draw as box-and-pointer notation),
- an environment, E1, ..., E6, GE,
- nothing

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

Question 18. To what is the variable `memoize` in the global environment bound? **P3**

Question 19. To what is the variable `my-sq` in the global environment bound? **P2**

Question 20. To what is the variable `diary` in E6 bound?

the box-and-pointer diagram associated with ((5 25))

Question 21. To what is the variable `arg` in E2 bound? **5**

Question 22. What variable is bound to the procedure object P1? Give both the name and the environment. **to proc, in E4**

Question 23. What variable is bound to the procedure object P5? Give both the name and the environment. **nothing**

Part 3: (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 (for lists, use the printed form, not the box-and-pointer form). Each question is independent—in other words, for each question, 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 sequence of evaluations for a question 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**.

Question 24.

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

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

Question 25.

```
(define x (list 'a 'b))
(define y (list 'c 'd))
(set-car! y 'a)
(set-cdr! y (cdr x))
(eq? x y)
```

```
#f
```

For the following questions, you may find it convenient to draw an environment diagram. Also, note carefully the small changes between these questions.

Question 26.

```
(define x 2)
(define proc
  (let ((x (* x 5))
        (f (lambda (y) (* x y))))
    f))
(proc 3)
```

6

Question 27.

```
(define x 2)
(define proc
  (let ((x (* x 5))
        (let ((f (lambda (y) (* x y))))
          f)))
(proc 3)
```

30**Question 28.**

```
(define x 2)
(define proc
  (let ((x (* x 5))
        (f (lambda (y) (* x y))))
    f))
(set! x 5)
(proc 3)
```

15**Question 29.**

```
(define x 2)
(define proc
  (let ((x (* x 5))
        (let ((f (lambda (y) (* x y))))
          f)))
(set! x 5)
(proc 3)
```

30**Question 30.**

```
(define x 2)
(define proc
  (let ((x (* x 5))
        (f (lambda (y) (* x y))))
    (set! x 5)
    f))
(proc 3)
```

6**Question 31.**

```
(define x 2)
(define proc
  (let ((x (* x 5))
        (let ((f (lambda (y) (* x y))))
          (set! x 5)
          f)))
(proc 3)
```

15**Part 4 (20 points)**

We are going to build a small system for handling bank accounts. Each account has a name and a “balance”.

```
(define (make-account name)
  (list name (make-balance 0)))

(define (get-account-name acc) (car acc))
(define (get-balance acc) (cadr acc))
```

A “balance” is slightly unusual, in that it consists of a current amount, plus any pending actions, such as withdrawals or deposits that have not yet been cleared by the bank. Thus we represent a balance as a log of actions (represented as a list), and an amount:

```
(define (make-balance amt)
  (list '() amt))
```

Question 32. Complete the abstraction for a “balance” by providing the missing code for each of the following procedures:

```
(define (get-balance-actions bal) (CAR BAL))
;; should return the list of pending actions

(define (get-actual-balance bal) (CADR BAL))
;; should return the current amount

(define (set-balance-actions! bal actions) (SET-CAR! BAL ACTIONS))
;; should replace the list of pending actions with the new argument

(define (set-actual-balance! bal amt) (SET-CAR! (CDR BAL) AMT))
;; should replace the current amount with the new argument
```

For representing pending actions of an account, we will use tagged data structures:

```
(define (make-entry tag time amt)
  (list tag time amt))
(define (entry-time entry)
  (cadr entry))
(define (entry-amount entry)
  (caddr entry))

(define (tagged-entry? entry tag)
  (and (pair? entry) (eq? (car entry) tag)))

(define (make-withdrawal time amt)
  (make-entry 'withdrawal time amt))
(define (make-deposit time amt)
  (make-entry 'deposit time amt))
```

Question 33. Complete the data abstraction for a pending withdrawal by completing the definition of the following predicate (we will assume that a similar predicate `deposit?` exists):

```
(define (withdrawal? entry)
  (TAGGED-ENTRY? ENTRY 'WITHDRAWAL))
```

Now we can build an interface to our system, in which whenever we take an action such as withdrawing or depositing, this is added as an action to the “balance” data structure. (Assume that a similar procedure for making deposits also exists, called `deposit`.)

```
(define (withdraw amt acc)
  (set-balance-actions!
   (get-balance acc)
   (cons (make-withdrawal (real-time-clock) amt)
         (get-balance-actions (get-balance acc))))
  'withdrawn)
```

Thus we might have the following actions:

```
(define mine (make-account 'eric))

(deposit 100 mine)

(deposit 100 mine)

(withdraw 50 mine)
```

```
mine
;Value: (eric (((withdrawal 42780 50) (deposit 37950 100) (deposit 33060 100)) 0)
```

Eventually we will need to update the balance, by processing all the pending actions, and changing the actual amount in the balance. We do this with

```
(define (update-acct acc)
  (let ((bal (get-balance acc)))
    (set-actual-balance! bal (get-expected-balance bal))
    (set-balance-actions! bal '())))
```

Question 34 To complete this, we need to implement `get-expected-balance`. This should return the new balance in the account by adding and subtracting pending deposits and withdrawals to the current balance. Provide the missing code.

```
(define (get-expected-balance bal)
  (accumulate
   (LAMBDA (ENTRY AMT)
     (COND ((WITHDRAWAL? ENTRY)
            (- AMT (ENTRY-AMOUNT ENTRY)))
           ((DEPOSIT? ENTRY)
            (+ AMT (ENTRY-AMOUNT ENTRY)))))
   (get-actual-balance bal)
   (get-balance-actions bal)))
```

Part 5 (24 points):

The following object oriented class hierarchy covers different ways objects can be identified. Look over the code carefully before answering the questions.

```
(define (make-A name)
  (lambda (msg)
    (case msg
      ((NAME) (lambda (self) name))
      ((SWITCH) (lambda (self newname)
                  (set! name newname)
                  (ask self 'name)))
      ((WHO) (lambda (self)
              (newline)
              (display "original: ")
              (display name)
              name))
      (else (no-method))))))

(define (make-B name)
  (let ((my-A (make-A name)))
    (lambda (msg)
      (case msg
        ((SWITCH) (lambda (self newname)
                    (set! name newname)
                    (ask self 'name)))
        ((WHO)
         (lambda (self)
           (ask my-A 'who)
           name))
        (else (get-method msg my-A))))))

(define (make-C name)
  (let ((my-A (make-A name)))
    (lambda (msg)
      (case msg
        ((NAME) (lambda (self) name))
        ((SWITCH) (lambda (self newname)
                    (ask my-A 'switch newname)))
        (else (get-method msg my-A))))))
```

Assume we make the above definitions, and then evaluate

```
(define aaron (make-A 'aardvark))  
(define barry (make-B 'barracuda))  
(define cody (make-C 'cougar))
```

What gets printed (either by the "display" procedure and/or by the read-eval-print loop) for each of the following expressions? (Assume that they are evaluated in this order.) You may find it helpful to draw an instance diagram or a class diagram to keep track of the state and structure of the objects created in this example.

Question 35.

```
(ask aaron 'WHO)
```

```
original: aardvark  
;Value: aardvark
```

Question 36.

```
(ask aaron 'switch 'ants)
```

```
;Value: ants
```

Question 37.

```
(ask aaron 'WHO)
```

```
original: ants  
;Value: ants
```

Question 38.

```
(ask barry 'WHO)
```

```
original: barracuda  
;Value: barracuda
```

Question 39.

```
(ask barry 'switch 'bruin)
```

```
;Value: barracuda
```

Question 40.

(ask barry 'WHO)

original: barracuda
;Value: bruin

Question 41.

(ask cody 'WHO)

original: couger
;Value: cougar

Question 42.

(ask cody 'switch 'cat)

;Value: cat

Question 43.

(ask cody 'WHO)

original: cat
;Value: cat

BACKGROUND INFORMATION—THIS PAGE CONTAINS NO QUESTIONS

```

; Our familiar OOP system

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

(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 (safe-ask 'unnamed-object in-object 'NAME)))
          (else (error "Can't delegate" message
                       "from" (safe-ask 'unnamed-object for-object 'NAME)
                       "to" (safe-ask 'unnamed-object in-object 'NAME))))))

(define (safe-ask default-value obj msg . args)
  (let ((method (get-method msg obj)))
    (if (method? method)
        (apply-method obj obj msg args)
        default-value)))

(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)))

```

BACKGROUND INFORMATION—THIS PAGE CONTAINS NO QUESTIONS