

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 2003

Quiz II

Closed Book – two sheets of notes

Separately, we have distributed an answer sheet. You may use the space on the exam booklet for whatever temporary work you find useful, but you **MUST** enter your answers into the answer sheet. **Only the answer sheet will be graded.** Each problem that requires an answer has been numbered. Place your answer at the corresponding number in the answer sheet.

Note that any procedures or code fragments that you write will be judged not only on correct function, but also on clarity and good programming practice.

The answer sheet asks for your section number and tutor. For your reference, here is the table of section numbers.

Section	Time	Location	Rec. Instructor	Tutors
1	10:00	26-328	Joel Moses	Bryan Russell
2	10:00	36-155	Regina Barzilay	Ben Vandiver
3	11:00	26-328	Joel Moses	Dave Feinberg
4	12:00	36-144	Regina Barzilay	
5	1:00	26-204	Randy Davis	Limor Fried
6	2:00	26-204	Randy Davis	Vivienne Lee

Part 1: (20 points)

Here are four different attempts to create a procedure with state:

```
;Version 1
(define object
  (let ((init 0))
    (lambda (new)
      (let ((temp init))
        (set! init new)
        temp))))
```

```
;Version 2
(define object
  (lambda (new)
    (let ((init new))
      (let ((temp init))
        (set! init new)
        temp))))
```

```
;Version 3
(define object
  (lambda (new)
    (let ((init 0))
      (set! init new)
      init)))
```

```
;Version 4
(define object
  (let ((init 0))
    (lambda (new)
      (set! init new)
      init)))
```

For each version of the code, we further evaluate

```
(object 1)
(object 2)
```

On the attached sheet, we show some possible environment diagrams.

Question 1. Assume version 1 of the code for `object`. Which environment diagram (from A to F) corresponds to the state of the environment after evaluating:

```
(object 1)
(object 2)
```

Question 2. Assume version 2 of the code for `object`. Which environment diagram (from A to F) corresponds to the state of the environment after evaluating:

(object 1)
(object 2)

Question 3. Assume version 3 of the code for `object`. Which environment diagram (from A to F) corresponds to the state of the environment after evaluating:

(object 1)
(object 2)

Question 4. Assume version 4 of the code for `object`. Which environment diagram (from A to F) corresponds to the state of the environment after evaluating:

(object 1)
(object 2)

Here are a set of possible outputs (that is, values returned)

Choice	Output after (object 1)	Output after (object 2)
A	error	error
B	1	error
C	0	0
D	1	1
E	0	1
F	1	2

Question 5. Assume version 1 of the code for `object`. Select from choices A to F, the output from evaluating:

(object 1)
(object 2)

Question 6. Assume version 2 of the code for `object`. Select from choices A to F, the output from evaluating:

(object 1)
(object 2)

Question 7. Assume version 3 of the code for `object`. Select from choices A to F, the output from evaluating:

(object 1)
(object 2)

Question 8. Assume version 4 of the code for `object`. Select from choices A to F, the output from evaluating:

(object 1)
(object 2)

Part 2: (20 points)

Earlier in lecture, we saw an example of a queue as a data structure. Remember that a queue had the following properties:

- a queue is a linear sequence of elements, characterized by a head and a tail,
- new elements are inserted only at the tail of the queue,
- elements are deleted only from the head of the queue.

Suppose we decide to build a queue data abstraction, where we assume that the internal representation of the sequence of elements is a list, and where the queue itself has the following data abstraction:

```
(define (make-q)
  (let ((front nil)
        (back nil))
    (cons front back)))

(define queue-head car) ;; access head of queue
(define queue-tail cdr)
(define set-queue-head! set-car!) ;; change head of queue
(define set-queue-tail! set-cdr!)

(define (queue-head-value q) (car (queue-head q)))
;;used to examine value at head of a queue, applied to queue directly, see below

(define (queue-tail-value q) (car (queue-tail q)))

(define (insert elt q) ; insert element at tail of queue
  (let ((new (list elt)))
    (cond ((null? (queue-head q))
           (set-queue-head! q new)
           (set-queue-tail! q new))
          (else (set-cdr! (queue-tail q) new)
                 (set-queue-tail! q new)))))

(define (delete q) ; remove element at head of queue, and return value
  (cond ((null? (queue-head q))
         (error "nothing in queue"))
        ((eq? (queue-head q) (queue-tail q))
         (let ((return (queue-head-value q)))
           (set-queue-head! q nil)
           (set-queue-tail! q nil)
           return))
        (else
         (let ((return (queue-tail-value q)))
           (set-queue-head! q (cdr (queue-head q)))
```

```
        return))))

;; for example
(define my-q (make-q))

(insert 1 my-q)

(insert 5 my-q)

(insert 2 my-q)

(queue-head-value my-q)
;Value: 1

(queue-tail-value my-q)
;Value: 2

(delete my-q)
;Value: 1

(queue-head-value my-q)
;Value: 5
```

Now, let's change things so that we have an **ordered queue**. This means that elements are still deleted from the front of the queue, but when an element is inserted into the queue, it is done so as to keep all the elements of the queue in ascending order. For simplicity, assume we only insert integers into the queue.

For example:

```
(define my-q (make-q))

(ordered-insert 1 my-q)

(queue-head-value my-q)
;Value: 1

(ordered-insert 5 my-q)

(queue-tail-value my-q)
;Value: 5

(ordered-insert 2 my-q)

(queue-tail-value my-q)
;Value: 5

(queue-head-value my-q)
;Value: 1
```

Here is a template for code to do ordered insertion (everything else should stay the same):

```
(define (ordered-insert elt q)
  (let ((new (list elt)))
    (cond ((null? (queue-head q))
           (set-queue-head! q new)
           (set-queue-tail! q new))
          ((< elt (value (queue-head q)))
           ANSWER-9)
          ((>= elt (value (queue-tail q)))
           ANSWER-10)
          (else (insert-in-place new (queue-head q))))))
```

Question 9. What code should be used for ANSWER-9?

Question 10. What code should be used for ANSWER-10?

The procedure `insert-in-place` should be used to adjust the internal sequence of elements of the queue, without mutating the head or tail of the queue.

Question 11. Write the procedure `insert-in-place`.

Part 3: (16 points)

Suppose we want to make a very simple clock, with the behavior shown below:

```
(define c1 (clock))

(c1)
;Value: tick

(c1)
;Value: tock

(c1)
;Value: tick

(c1)
;Value: tock

(define c2 (clock))

(c2)
;Value: tick

(c1) ;; note that this is calling C1 again
;Value: tick

(c2) ;; note that this is calling C2 again
;Value: tock
```

Thus this object simply alternates between a “tick” and a “tock”, on each call to the procedure.

Question 12. Write a definition for `clock` that achieves this behavior. You may not use any variables in the global environment, other than the definitions invoked in the example above.

Part 4: (24 points)

A **graph** is composed of a set of **nodes**, each of which contains an entry and a set of neighbors:

```
(define (node entry nbrs)
  (lambda (msg)
    (case msg
      ((entry) entry)
      ((nbrs) nbrs)
      ((add-nbr)
       (lambda (who) (set! nbrs (cons who nbrs)))))))

(define n1 (node 'ab '()))
(define n2 (node 'cd '()))
(define n3 (node 'ef '()))
(define n4 (node 'gh '()))

((n1 'add-nbr) n2)
((n1 'add-nbr) n3)
((n2 'add-nbr) n1)
((n2 'add-nbr) n4)
```

If we want to find a node with a particular entry or value, we can search through the graph, starting at some designated start point.

```
(define (find value start)
  (search value (list start) '()))

(define (search value todo seen)
  (cond ((null? todo) 'not-found)
        ((memq (car todo) seen) ;; have already examined next node
         ANSWER-13)
        (ANSWER-14
         (car todo)) ; return the actual node
        (else ANSWER-15&16)))
```

The idea is to use **todo** to keep a list of nodes to be examined, and to use **seen** to keep track of those nodes which it has already examined (so we don't loop forever inside the graph). The procedure **memq** returns a true value if the first argument is contained in the second argument (which is a list).

Question 13. What code should be used to handle the search when the next node in the **todo** list has already been examined? (ANSWER13)

Question 14. What code should be used to test if the next node in the **todo** list has an entry that matches the value being sought? (ANSWER14)

Question 15. What code should be used to continue the search, with the property that the procedure should execute a **breadth first** search? (ANSWER15&16)

Question 16. What code should be used to continue the search, with the property that the procedure should execute a **depth first** search? (ANSWER15&16)

Part 5: (20 points)

This problem explores a small object-oriented world, consisting of Things and Containers. The desired behavior of the classes is as follows:

- a Thing is an object with a name, a weight, and a volume.
- a Container is a set of things.
- a Box is a hard-sided wooden structure. Its weight varies with its contents, but its volume is always the same.
- a Tote is a flexible cloth bag. Both its weight and its volume depend on its contents.

The code defining the classes is shown below. Beware! This code has several mistakes in it, which you will be asked to fix.

```
(define (make-thing name weight volume)
  (lambda (msg)
    (case msg
      ((NAME) (lambda (self) name))
      ((WEIGHT) (lambda (self) weight))
      ((VOLUME) (lambda (self) volume))
      ((DENSITY) (lambda (self) (/ weight volume)))
      (else (no-method))))))

(define (make-container)
  (let ((things '()))
    (lambda (msg)
      (case msg
        ((THINGS) (lambda (self) things))
        ((ADD-THING) (lambda (self thing)
                       (set! things (cons thing things))
                       (map (lambda (thing) (ask thing 'NAME)) things)))
        ((WEIGHT) (lambda (self)
                    (fold-right + 0
                                (map (lambda (thing) (ask thing 'WEIGHT))
                                     things))))
        ((VOLUME) (lambda (self)
                    (fold-right + 0
                                (map (lambda (thing) (ask thing 'VOLUME))
                                     things))))
        (else (no-method))))))

(define (make-Box name weight-when-empty volume)
  (let ((thing-part (make-thing name weight-when-empty volume))
        (container-part (make-container)))
    (lambda (msg)
      (case msg
        ((WEIGHT) (lambda (self)
```

```

      (+ (delegate thing-part self 'WEIGHT)
         (delegate container-part self 'WEIGHT))))
      (else (find-method msg thing-part container-part))))))

(define (make-tote name)
  (let ((thing-part (make-thing name 0 0))
        (container-part (make-container)))
    (lambda (msg)
      (case msg
        ((VOLUME) (delegate container-part self 'VOLUME))
        (else (find-method msg thing-part container-part))))))

```

Assume the following definitions have been evaluated:

```

(define chest (make-thing 'Old-Dusty-Chest 200 16))
(define textbook (make-thing 'SICP 1 0.5))
(define box1 (make-box 'Box1 40 20))
(define tote1 (make-tote 'Tote1))

```

What is the value of each of the following expressions, assuming they are evaluated in the order shown? (Write *unspec* for unspecified, *error* for error, or *procedure* for a procedure value.)

Question 17. (ask tote1 'WEIGHT)

Question 18. (ask Box1 'WEIGHT)

Question 19. (ask Box1 'ADD-THING chest)

Question 20. (ask Box1 'WEIGHT)

Question 21. (ask Box1 'VOLUME)

Question 22. (ask Box1 'DENSITY)

Question 23. Rewrite Thing's DENSITY method so that it computes the correct density for Box1:

```
((DENSITY) your-code-goes-here)
```

Question 24. (ask tote1 'ADD-THING textbook)

Question 25. (ask tote1 'WEIGHT)

Question 26. (ask tote1 'VOLUME)

Question 27. Rewrite the Tote class so that it returns the correct weight and volume for tote1.

```

(define (make-tote name)
  (let ((thing-part (make-thing name 0 0))
        (container-part (make-container)))
    (lambda (msg)
      your-code-goes-here)))

```

BACKGROUND INFORMATION—THIS PAGE CONTAINS NO QUESTIONS

```

; Our familiar OOP system

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