

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
 Department of Electrical Engineering and Computer Science
 6.001—Structure and Interpretation of Computer Programs
 Spring 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 this 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	9:00	26-314	Jovan Popovic	Toh Ne Win
2	10:00	34-301	Konrad Tollmar	Matt Peters
3	10:00	26-314	Jovan Popovic	Limor Fried
4	10:00	34-304	Robert Miller	Adnan Dosani
5	11:00	34-301	Konrad Tollmar	Matt Peters Limor Fried Alex Andersson
6	2:00	26-204	Robert Miller	Todd Atkins Aaron Strauss
7	12:00	34-301	Michael Collins	Alex Andersson
8	1:00	34-303	Tony Eng	Aaron Strauss
9	1:00	34-30	Michael Collins	Samitha Samaranayake
10	2:00	34-303	Tony Eng	Todd Atkins
11	11:00	26-314	Fredo Durand	Dave Feinberg
12	12:00	26-314	Fredo Durand	Aaron Strauss Dave Feinberg Adnan Dosani Samitha Samaranayake

Part 1: (16 points)

Memoization is a clever idea that allows us to save on computation. It works by keeping track of evaluation of a procedure on a specific argument, and simply returns the remembered value if the procedure has already been run on that argument. It does not work for all procedures, but can be handy when it does apply. Here is an implementation:

```
(define (memoize proc)
  (let ((return '()))
    (lambda (arg)
      (if (assq arg return)
          (cadr (assq arg return))
          (let ((new-val (proc arg)))
              (set! return new-val)
              return))))))
```

Suppose that after evaluating this definition, we evaluate the following two expressions, in the order shown:

```
(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 P5, identify, if possible, the environment pointer of the procedure (i.e. one of GE, E1, ..., E5). If the appropriate environment is not shown, write “not shown”.

Question 1. To what does the environment pointer of P1 point?

Question 2. To what does the environment pointer of P2 point?

Question 3. To what does the environment pointer of P3 point?

Question 4. To what does the environment pointer of P4 point?

Question 5. To what does the environment pointer of P5 point?

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 6. What is the enclosing environment for E1?

Question 7. What is the enclosing environment for E2?

Question 8. What is the enclosing environment for E3?

Question 9. What is the enclosing environment for E4?

Question 10. What is the enclosing environment for E5?

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

- one of the procedure objects, P1, ..., P5,
- 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, ..., E5, GE,
- nothing

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

Question 11. To what is the variable `memoize` in the global environment bound?

Question 12. To what is the variable `x` in E2 bound?

Question 13. What variable is bound to the procedure object P3? Give both the name and the environment, if applicable.

Question 14. What variable is bound to the procedure object P4? Give both the name and the environment, if applicable.

Part 2 (24 points):

Suppose we are given a list of symbols, and we want to find patterns within that list. You are to write a procedure called `subsequences` that takes as argument two lists of symbols, and returns the number of times the first sequence appears within the second one. If one of the symbols in the first sequence is the special character `*`, then that symbol matches any other symbol.

For example:

```
(define *test* '(a b c a b a b a b c b c b c a b c))

(subsequences '(a) *test*)
;Value: 5

(subsequences '(a b c) *test*)
;Value: 3

(subsequences '(a b c) '())
;Value: 0

(subsequences '(a *) *test*)
;Value: 5

(subsequences '(c *) *test*)
;Value: 4
```

Here is an initial template for the procedure. The idea behind the internal procedure is that it keeps track of where you are in terms of matching the pattern to a spot in the text and where in the list of text words you started this particular match. Thus, if the pattern completely matches, you can increment your count by 1, otherwise start matching the pattern again, starting at the next point in the sequence.

```
(define (subsequences pattern text)
  (define (match pt txt start)
    (cond ((null? pt)
           (+ 1 (match CODE1)))
          ((null? txt) 0)
          ((eq? (car pt) '*)
           (match CODE2))
          ((eq? (car pt) (car txt))
           (match CODE3))
          (else (match CODE4))))
  (match pattern text text))
```

Your job is to supply the arguments for each of the calls to `match`.

Question 15. What arguments should be supplied in the spot marked CODE1, to match the pattern against the remaining portion of the sequence?

Question 16. What arguments should be supplied in the spot marked CODE2, to match the rest of the pattern against the remaining portion of the sequence?

Question 17. What arguments should be supplied in the spot marked CODE3, to continue the match of the pattern against the sequence?

Question 18. What arguments should be supplied in the spot marked CODE4?

Part 3 (22 points)

A **queue** is a common data structure – it has the property that elements are stored in a linear fashion, but can only be added at the end, and removed from the front. For example, if `make-q` creates an empty queue, `enqueue` is used to add elements to the queue, and `dequeue` is used to remove elements from the queue (and return the element), then we might have:

```
(define q (make-q))
;Value: ‘‘q --> (queue empty empty)’’
```

```
(enqueue q 5)
;Value: (queue (5) (5))
```

```
(enqueue q 2)
;Value: (queue (5 2) (2))
```

```
(enqueue q 3)
;Value: (queue (5 2 3) (3))
```

```
(dequeue q)
;Value: 5
```

```
q
;Value: (queue (2 3) (3))
```

The basic idea is that a queue is represented as a list of three elements, the symbol `queue` and pointers to the head (or beginning) and tail (or end) of the queue. We represent this as a data abstraction, as defined below. The actual elements of the queue are stored as a list, so that the queue head pointer would point to the first cons pair of the list, and the queue tail pointer would point to the last cons pair of the list.

Here is a definition of the initial constructor:

```
(define (make-q)
  (list 'queue 'empty 'empty))
```

And to get the head and tail of the queue, we use

```
(define (queue-head q)
  (if (and (pair? q) (eq? (car q) 'queue)) ; have a tagged structure
      (cadr q)))
```

```
(define (queue-tail q)
  (if (and (pair? q) (eq? (car q) 'queue)) ; have a tagged structure
      (caddr q)))
```

An empty queue is found using

```
(define (empty-queue? q)
  (if (and (pair? q) (eq? (car q) 'queue)) ; have a tagged structure
      (eq? (cadr q) 'empty)))
```

In answer the questions below, be sure to use the right data abstractions – head and tail operations for the queue data structure, list operations for the actual list of elements.

Here is a template for adding elements to the queue:

```
(define (enqueue q elt)
  (let ((new (list elt))) ;; portion of list that will contain addition
    (cond ((empty-queue? q)
           (change-head q new)
           (change-tail q new))
          (else
           INSERT1
           (change-tail q new))))))
```

Question 19. `change-head` should adjust the head pointer of the queue to point to a new cons pair. Write this procedure.

Question 20. `change-tail` should adjust the tail pointer of the queue to point to a new cons pair. Write this procedure.

Question 21. What should be used in place of `INSERT1`, in order to complete the adjustment of the queue structure?

To remove an element from the front of the queue, we use the following procedure. Remember that the queue must be maintained in the same order, and the value at the front of the queue must be returned:

```
(define (dequeue q)
  (if (empty-queue? q)
      (error "empty queue")
      (if (eq? (queue-head q) (queue-tail q))
          INSERT2
          INSERT3)))
```

Question 22. What should be used in place of `INSERT2`?

Question 23. What should be used in place of `INSERT3`?

Part 4: (18 points)

A **cycle** or **ring** is an interesting data structure. It consists of a linear sequence of elements, but one that is circular. One can think of this as a strange version of a list in which the last pair of the list points back to the first pair of the list. For this question, assume that a ring consists of at least 2 elements.

We can create a ring by mutating a list, and we choose to represent a ring as a message passing object.

Assume that `last` of a non-empty list returns the last pair in the list. Here is a template for creating a ring.

```
(define (make-ring lst)
  (let ((pointer lst)
        (last-pointer lst))
    (lambda (msg)
      (case msg
        ((CONNECT) (lambda (self) (set-cdr! (last lst) lst)))
        ((VALUE) (lambda (self) (car pointer)))
        ((RIGHT)  INSERT4)
        ((LEFT)  INSERT5)
        ((UPDATE) (lambda (self) (set! last-pointer pointer)
                       'done))
        ((ROTATE) (lambda (self dirn)
                     (ask self dirn)
                     (ask self 'update))))))))
```

And here is an example of using a ring:

```
(define test '(1 2 3 4 5 6 7))

(define test-ring (make-ring test))

(ask test-ring 'connect)

(ask test-ring 'value)
;Value: 1

(ask test-ring 'rotate 'right)
;Value: done

(ask test-ring 'value)
;Value: 2

(ask test-ring 'rotate 'right)
;Value: done

(ask test-ring 'value)
;Value: 3
```

```
(ask test-ring 'rotate 'left)
;Value: done
```

```
(ask test-ring 'value)
;Value: 2
```

To make this work, you need to complete the procedure by providing methods for rotating to the right and left. The idea of the ring is that `pointer` points to the current “start” of the ring, hence when asked for a value, it returns the value in that pair. To rotate to the right, we need to update the pointer to point to the next cell in the structure. To rotate to the left, we need to keep rotating to the right until we reach the cons pair immediately before the one we started at. To accomplish this, we will use `last-pointer` to keep track of the starting point. Thus, as you can see, the method for `rotate` resets the `last-pointer` after we are done.

Question 24. What should be used in place of `INSERT4`?

Question 25. What should be used in place of `INSERT5`?

Question 26. Write the procedure `last`. Assume that it is only applied to non-empty lists.

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 Crate is a hard-sided wooden crate. Its weight varies with its contents, but its volume is always the same.
- a Bag 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)
                    (accumulate + 0
                               (map (lambda (thing) (ask thing 'WEIGHT))
                                     things))))
        ((VOLUME) (lambda (self)
                    (accumulate + 0
                               (map (lambda (thing) (ask thing 'VOLUME))
                                     things))))
        (else (no-method))))))

(define (make-crate 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-bag 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 armoire (make-thing 'Old-Armoire 200 16))
(define bear (make-thing 'Teddy-Bear 1 0.5))
(define crate1 (make-crate 'Crate1 40 20))
(define bag1 (make-bag 'Bag1))

```

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

Question 27. (ask bag1 'WEIGHT)

Question 28. (ask crate1 'WEIGHT)

Question 29. (ask crate1 'ADD-THING armoire)

Question 30. (ask crate1 'WEIGHT)

Question 31. (ask crate1 'VOLUME)

Question 32. (ask crate1 'DENSITY)

Question 33. Rewrite Thing's DENSITY method so that it computes the correct density for crate1:

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

Question 34. (ask bag1 'ADD-THING bear)

Question 35. (ask bag1 'WEIGHT)

Question 36. (ask bag1 'VOLUME)

Question 37. Rewrite the Bag class so that it returns the correct weight and volume for bag1.

```

(define (make-plastic-bag 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)))

  (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