MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 2002

Quiz II – Sample solutions

Below are example solutions for each of the questions. These are not the only possible answers, but they are the most common ones.

Part 1: (18 points)

Consider the following procedure:

```
(define (last-call proc)
  (let ((old '*not-used*))
    (lambda (arg)
      (let ((temporary old))
        (set! old (proc arg))
        temporary))))
```

For example, we might have:

```
(define weird (last-call (lambda (x) (* x x))))

(weird 5)
;Value: *not-used*

(weird 7)
;Value: 25
```

Suppose we actually evaluate the above sequence of expressions in the order shown:

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 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".

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

GE

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

GE

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

E7

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

E6

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 5. What is the enclosing environment for E1?

E2

Question 6. What is the enclosing environment for E2?

E6

Question 7. What is the enclosing environment for E3?

E6

Question 8. What is the enclosing environment for E4?

E3

Question 9. What is the enclosing environment for E5?

GE

Question 10. What is the enclosing environment for E6?

E7

Question 11. What is the enclosing environment for E7?

GE

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, ..., E7, GE,

- nothing

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

Question 12. To what is the variable `last-call` in the global environment bound?

P2

Question 13. To what is the variable `proc` in E7 bound?

P1

Question 14. To what is the variable `old` in E6 bound?

49

Question 15. What variable is bound to the procedure object P4? Give both the name and the environment, if applicable.

weird, bound in GE

Question 16. What variable is bound to the procedure object P5? Give both the name and the environment, if applicable.

nothing

Part 2 (18 points):

At the end of the quiz you will find the code for our object oriented system.

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

```
(define (make-A state)
  (lambda (msg)
    (case msg
      ((GETSTATE) (lambda (self) state))
      ((ALTER) (lambda (self newstate)
                 (set! state newstate)
                 (ask self 'GETSTATE)))
      ((REPORT) (lambda (self)
                  (list (ask self 'GETSTATE))))
      (else (no-method)))))

(define (make-B state)
  (let ((my-A (make-A state)))
    (lambda (msg)
      (case msg
        ((ALTER) (lambda (self newstate)
                   (set! state newstate)
                   (ask self 'GETSTATE)))
        ((REPORT)
         (lambda (self)
```

```
           (cons state (ask my-A 'REPORT))))
        (else (get-method msg my-A))))))

(define (make-C state)
  (let ((my-A (make-A state))
        (my-B (make-B state)))
    (lambda (msg)
      (case msg
        ((GETSTATE) (lambda (self) (ask my-A 'GETSTATE)))
        ((ALTER) (lambda (self newstate)
                   (ask my-A 'ALTER newstate)))
        ((REPORT) (lambda (self)
                   (delegate my-B self 'REPORT)))
        (else (get-method msg my-A))))))
```

Assume we make the above definitions, and then evaluate

```
(define alex (make-A 'alpha))
(define bruce (make-B 'beta))
(define chris (make-C 'gamma))
```

What gets returned as value 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 17.

```
(ask alex 'REPORT)
```

(alpha)

Question 18.

```
(ask alex 'ALTER 'alef)
```

alef

Question 19.

```
(ask alex 'REPORT)
```

(alef)

Question 20.

```
(ask bruce 'REPORT)
```

(beta beta)

Question 21.

```
(ask bruce 'ALTER 'betel)
```

beta

Question 22.

```
(ask bruce 'REPORT)
```

(betel beta)

Question 23.

```
(ask chris 'REPORT)
```

(gamma gamma)

Question 24.

```
(ask chris 'ALTER 'gum)
```

gum

Question 25.

```
(ask chris 'REPORT)
```

(gamma gamma)

Part 3 (20 points)

We are going to add a new special form to our meta-circular evaluator (a copy of which is attached at the end of the quiz). The form we are going to add is an `unless` expression, such as

```
(define test '(1 2 3))

(unless (null? test)
        (display (car test))
        (set! test (cdr test)))
```

The idea is that an `unless` expression consists of two parts. The first sub-expression is a `test` clause, the remaining sub-expressions are the body. The evaluation of an `unless` is to first evaluate the `test`. If it is true, then evaluation of the entire `unless` is terminated, with an unspecified return value. Otherwise, the body of the `unless` is evaluated and the process repeats.

To the meta-circular evaluator, we add the following clause

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

where

```
(define (unless? exp) (tagged-list? exp 'unless))

(define (unless-test exp) (cadr exp))

(define (unless-body exp) (cddr exp))
```

Question 26. Write the procedure `eval-unless` by completing the following defintion. Be sure to use the appropriate data abstractions.

```
(define (eval-unless exp env)
  (IF (NOT (M-EVAL (UNLESS-TEST EXP) ENV))
      (BEGIN (EVAL-SEQUENCE (UNLESS-BODY EXP) ENV)
             (EVAL-UNLESS EXP ENV))))
```

Question 27. Suppose **instead** that we add the following clause to the evaluator:

```
((unless? exp) (m-eval (unless->if exp) env))
```

Complete the syntactic transformation

```
(define (unless->if exp)
 (LIST 'IF
       (LIST 'NOT (UNLESS-TEST EXP))
       (CONS 'BEGIN
             (APPEND (UNLESS-BODY EXP)
                     (LIST EXP)))))
```

so that this will correctly evaluate `unless` expressions.

Part 4: (22 points)

We want to construct a data abstraction called a **queue**. A queue consists of an ordered sequence of items, which we will represent using a list. Users of queues see a data abstraction, with a queue consisting of two components, a **head** and a **tail**. One can read the element at the head of the queue, but cannot directly access other elements in the queue. Attempting to read an empty queue is an error.

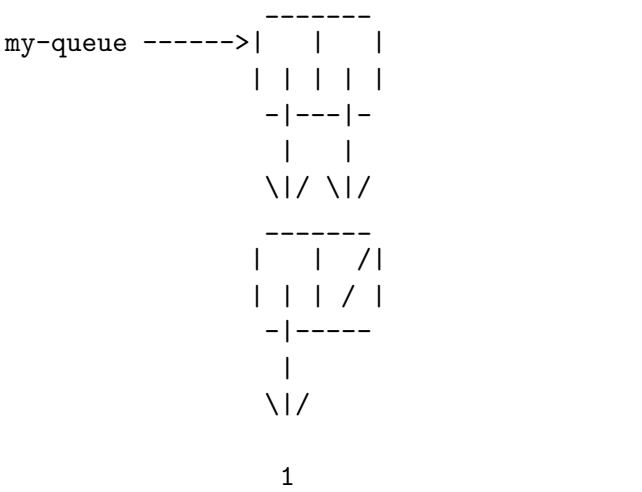
One can only add new elements to a queue at the **tail** of the queue, and one can only remove elements from the **head** of the queue.

For example, here is some code to create queues, and some interactions with a queue.

```
(define (make-queue elt)
  (let ((temp (list elt)))
    (cons temp temp)))

(define (head q) (car q))
```

```
(define change-head! set-car!)

(define (tail q) (cdr q))

(define change-tail! set-cdr!)

(define (read q) (car (head q)))

(define (add-queue elt q)
  (let ((new-part (list elt)))
    (cond ((null? (tail q))
           INSERT-1)
          (else INSERT-2
                INSERT-3)))
  q)

(define (delete-queue q)
  (cond ((eq? (head q) (tail q))
         (change-head! q #f)
         (change-tail! q #f))
        (else INSERT-4))
  q)

;; HERE IS AN EXAMPLE QUEUE

(define my-queue (make-queue 1))
;Value: ((1) 1)

(read my-queue)
;Value: 1
```

Question 28: Draw a box and pointer diagram for the value of `my-queue`.



Now, consider the following actions on a queue:

```
(add-queue 2 my-queue)
;Value: ((1 2) 2)

(read my-q)
;Value: 1
;; NOTE: the same value is still at the head

(add-queue 3 my-queue)
;Value: ((1 2 3) 3)

(delete-queue my-queue)
;Value: ((2 3) 3)

(read my-q)
;Value: 2
;; NOTE: the value under the head has now changed

(delete-queue my-queue)
;Value: ((3) 3)

(delete-queue my-queue)
;Value: (#f)
;;NOTE: we are now left with an empty queue
```

Question 29: Provide code for `INSERT-1`, which should create a queue with one element.

Question 30: Provide code for `INSERT-2`, so that the internal representation of the elements of the queue is modified correctly.

Question 31: Provide code for `INSERT-3`, so that the external representation of the queue is modified correctly.

```
(define (add-queue elt q)
  (let ((new-part (list elt)))
    (cond ((null? (tail q))
           (CHANGE-HEAD! Q NEW-PART)
           (CHANGE-TAIL! Q NEW-PART))
          (else (SET-CDR! (TAIL Q) NEW-PART)
                (CHANGE-TAIL! Q (CDR (TAIL Q))))))
  q)
```

Question 32: Provide code for `INSERT-4`.

```
(define (delete-queue q)
```

```
  (cond ((eq? (head q) (tail q))
         (change-head! q #f)
         (change-tail! q #f))
        (else (CHANGE-HEAD! Q (CDR (HEAD Q)))))
  q)
```

Now suppose we decide to change the underlying representation of a queue, to use a list rather than a pair to represent the head and tail of the queue.

```
(define (make-queue elt)
  (let ((temp (list elt)))
    (list temp temp)))   ;; NOTE THE CHANGE HERE
```

Question 33: What is the minimal set of definitions that must change for the system to still operate correctly? Provide the correct new definitions.

```
(DEFINE (TAIL Q) (CADR Q))

(DEFINE (CHANGE-TAIL! Q NEW)
   (SET-CAR! (CDR Q) NEW))
```

Part 5 (22 points): Attached at the end of the quiz is a copy of the meta-circular evaluator. As it presently stands, this version of the evaluator cannot deal with procedures that use the "dot" notation for their arguments, such as:

```
(define foo
   (lambda (x . y)
     (if (> x 0) (length y))))

(foo 1 2 3 4)
;Value: 3

(foo 1)
;Value: 0

(foo 2 5)
;Value: 1
```

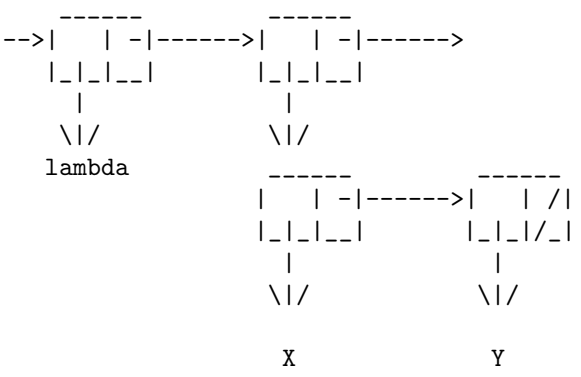
Remember that the idea behind the "dot" notation was to allow a procedure to be called with variable numbers of arguments. In this example, `foo` must be called with at least one argument, and the formal parameter `x` would be bound to that value.

However, `foo` could also be called with 2 or more arguments, and the formal parameter `y` would then be bound to a list of the values of all the subsequent arguments. Thus in

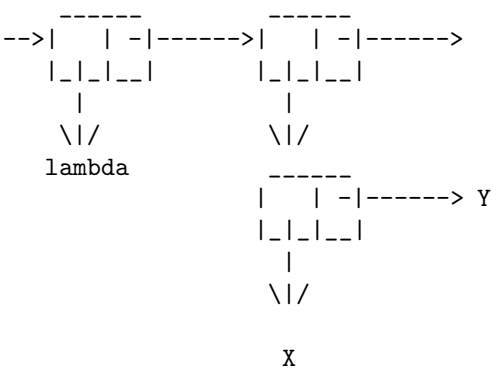
```
(foo 1 2 3 4)
```

the formal parameter `x` would be bound to the value `1` and the formal parameter `y` would be bound to the list `(2 3 4)`.

When we evaluate a `lambda` expression, it will create a representation for a procedure (see `make-procedure` in the evaluator code). In particular, when the reader converts the input expression such as `(lambda (x y) ...)` into an internal representation for the evaluator, it provides list structure like this:



whereas if the reader is given input such as `(lambda (x . y) ...)`, this is converted into list structure like this:



Thus when we evaluate a `lambda` expression whose parameter list is of the ordinary sort (e.g., `(x y)` or `(x y z a)`), the internal representation of this parameter list is an ordinary list (i.e., a sequence of `cons` cells whose `cdr`-pointers end in nil).

However, when the parameter list includes a "dot" (such as `(x . y)` or `(x y z . a)`), the representation for this parameter list will not be an ordinary list. It will differ in one small but important detail: the `cdr` of the last cons cell in the sequence will point to the last parameter (rather than to nil).

To change our evaluator to handle "dot" notation,, we just need to modify `extend-environment`:

```
(define (extend-environment vars vals base-env)
  (let ((processed-lists (find-dot vars vals)))
    (cons (make-frame (car processed-lists)
                      (cadr processed-lists))
          base-env)))
```

The `find-dot` procedure walks down the sequence of variable names and associated values, checking to see whether `vars` is an ordinary list (i.e., no "dot") or whether it is a sequence of cons cells that terminates by pointing at a variable rather than at nil (i.e., it had a dot). When we find that the `cdr` of the last cons cell points to a variable name rather than to `nil`, we want to
(a) add that variable to the end of a **list** of variables, and
(b) add the remaining values as a list, which should be added to the end of the list of all values.
(Notice that we still add the variable name to a `list` of variables, but handle the remaining values differently.)

Finally, `find-dot` will return a list of two lists – one for the variables, and one for the values.

Here is a template for this:

```
(define (find-dot vars vals)
  (define (help vars-seen vars-todo vals-seen vals-todo)
    (cond ((null? vars-to-do)     ; no more variables to examine
           (if (null? vals-to-do) ; if no more values either
               INSERT-5           ; then return collection of vars and vals
               (error ``wrong number args'' vars vals))) ; else report error
          ((null? (cdr vars-to-do)) ; we're at the last variable and
                                    ; there was no dot
           (if (null? vals-to-do)   ;   if no more values...
               (error ``wrong number args'' vars vals)     ; ...report error
               (if (not (null? (cdr vals-to-do)))  ; else if >1 more value
                   (error ``wrong number args'' vars vals) ; that's an error too
                   INSERT-6)))                             ; otherwise, do this
          ((not (pair? (cdr vars-to-do)))     ; sequence of vars doesn't end in nil
           (if (null? vals-to-do)             ; if no values left...
               (error ``wrong number args'' vars vals)  ;  ... report error
               INSERT-7))                               ;  ... else do this
          (else  ; otherwise collect next var and associated val, and continue
                 (help (cons (car vars-to-do) vars-seen)
                       (cdr vars-to-do)
                       (cons (car vals-to-do) vals-seen)
                       (cdr vals-to-do)))))
  (help '() vars '() vals))
```

Question 34:

What code should be used for **INSERT-5**?

```
(LIST (REVERSE VARS-SEEN) (REVERSE VALS-SEEN))
```

Question 35:

What code should be used for **INSERT-6**?

```
(LIST (REVERSE (CONS (CAR VARS-TO-DO) VARS-SEEN))
      (REVERSE (CONS (CAR VALS-TO-DO) VALS-SEEN)))
```

Question 36:

What code should be used for **INSERT-7**?

```
(LIST (REVERSE (CONS (CDR VARS-TO-DO)
                     (CONS (CAR VARS-TO-DO) VARS-SEEN)))
      (REVERSE (CONS (CDR VALS-TO-DO)
                     (CONS (CAR VALS-TO-DO) VALS-SEEN)))
```