MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 2005

Quiz II

Closed Book – two sheets of notes

Throughout this quiz, we have set aside space in which you should write your answers. Please try to put all of your answers in the designated spaces, as we will look only in this spaces when grading.

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.

NAME:

Section Number: Tutor's Name:

PART	Value	Grade	Grader
1	25		
2	30		
3	15		
4	30		
Total	100		

Part 1: (25 points)

We are going to explore a new kind of data structure, called a **cycle**. You can think of this as a kind of circular list: indeed we are going to represent a cycle as a loop of `cons` cells, where the `car` of each cell points to a value, and the `cdr` of each cell points to the next element in the circular sequence. To create a cycle from a list, we can use:

```
(define (list->cycle lst)
  (set-cdr! (last lst) lst)
  lst)

(define (last lst)
  (if (null? lst)
      (error "not long enough")
      (if (null? (cdr lst))
          lst
          (last (cdr lst)))))
```

For example:

```
(define test-cycle (list->cycle '(a b c g)))
```

To see what this structure looks like, you might find it convenient to draw a box-and-pointer diagram in the space below.

Associated with a cycle, we have several operations:

- `head` will return the value stored at the cell in the cycle pointed to by its argument, e.g. `(head test-cycle)` will return the value `a`;
- `rotate-left` will rotate the cycle one element to the left, e.g., `(head (rotate-left test-cycle))` will return the value `b`; (you can think of a cycle as a circular list in which the elements are arranged in clockwise sequence, the head is at the top, and "left" means rotating the sequence counter-clockwise);
- `rotate-right` will rotate the cycle one element to the right, in analogy to `rotate-left`, e.g., `(head (rotate-right test-cycle))` will return the value `g`.

Defining `head` is easy:

```
(define head car)
```

Question 1: Write the procedure `rotate-left`. Be sure that it returns a pointer to the right cell in the sequence. You may assume that the argument is a non-empty cycle.

Question 2: Write the procedure `rotate-right`. Be sure that it returns a pointer to the right cell in the sequence. You may assume that the argument is a non-empty cycle.

Now we want to add a new element to a cycle. This means we want to insert a new `cons` cell, with the supplied argument as its `car`, before the location in the cycle currently pointed to by the supplied argument. We are going to treat this as a mutator (that is, it changes an existing data structure), so we do not want to rely on the value returned by the procedure. Hence, by convention, let's have the procedure return the symbol `done`. Here is an example behavior:

```
(define test-cycle (list->cycle '(a b c g)))
;Value: test-cycle

(head test-cycle)
;Value: a

(head (rotate-left (rotate-left test-cycle)))
;Value: c

(insert-cycle! (rotate-left (rotate-left test-cycle)) 'x))
;Value: done

(head test-cycle)
;Value: a

(head (rotate-left (rotate-left test-cycle)))
;Value: x

(head (rotate-left (rotate-left (rotate-left test-cycle))))
;Value: c
```

Question 3: Write the procedure `insert-cycle!`. You may assume that the argument is a non-empty cycle. You should use `rotate-left` and/or `rotate-right` where appropriate.

Finally, suppose we want to remove an element from the cycle. Our goal is to remove from the cycle the cell to which the supplied argument points, as well as any pointers into the cycle from removed cells. Note that if we remove the cell to which a global variable refers, we may break our data structure, but we will assume that it is the responsibility of the user to manage pointers into the cycle.

Question 4: Provide a definition for the procedure `delete-cycle!`. Use `rotate-right, rotate-left, head` and `set-cdr!` but no other list operations. Also be sure that you change the `cdr` of the cycle cell that contained the deleted value to the empty list.

Part 2: (30 points)

We have introduced the concept of mutation (e.g., `set!`) into our language, which allows us to change the bindings of variables. Sometimes it would be nice to be able to undo one of these mutations, which means we would need some way of remembering what the previous value was. Consider the following code:

```
(define (set!-start val)
  (define old '())
  (let ((current val))
    (lambda (action . new)
      (cond ((eq? action 'value) current)
            ((eq? action 'new)
             (set! old (cons current old))
             (set! current (car new))
             current)
            ((eq? action 'reset)
             (set! current (car old))
             (set! old (cdr old))
             current)
            (else (error "I don't know how to do" action))))))

(define (set!-careful var val)
   (var 'new val))

(define (set!-undo var)
   (var 'reset))
```

Under this new scheme, a variable is actually a procedure, so to get its value, we just need to send it a message, for example:

```
(define foo (set!-start 5))

(foo 'value)
;Value: 5

(set!-careful foo 10)

(foo 'value)
;Value: 10

(set!-careful foo 15)

(foo 'value)
;Value: 15

(set!-undo foo)

(foo 'value)
;Value: 10
```

Now, suppose we evaluate the following expressions in a fresh environment.

```
(define foo (set!-start 5))

(set!-careful foo 10)
```

Attached is a partial environment diagram associated with this evaluation. You will notice that the frames are labeled E1, etc., and the procedure objects are labeled P1, etc. You should complete this diagram to reflect the state of the environment after evaluation of both expressions in order, and then use the results to answer the following questions.

Question 5: For each of the following environments, indicate the value of its enclosing environment, using one of **GE, E1, E2, E3, E4** or **none** or **not shown**.

	Enclosing environment
E1	
E2	
E3	
E4	

Question 6: For each of the following variables, indicate the value to which it is bound, using a number, a symbol, a list of numbers or symbols, or one of **P1, P2, P3, P4, P5**.

Variable	Environment	Value to which bound
set!-start	GE	
set!-careful	GE	
val	E2	
foo	GE	
new	E4	
action	E4	
var	E3	
val	E3	
current	E1	

Question 7: Where is the variable `old` bound (what environment), and what is its value?

location:

value:

Part 3 (15 points)

Here is a definition of a higher order procedure for performing computations on trees (where trees are represented as a list, whose elements are either leaves (i.e., values such as numbers or symbols) or other trees). You may assume that `leaf?` is a predicate that returns true for any object that is not a tree:

```
(define (process-tree tr leaf-op combine init)
  (if (null? tr)
      init
      (if (leaf? tr)
          (leaf-op tr)
          (combine (process-tree (car tr) leaf-op combine init)
                   (process-tree (cdr tr) leaf-op combine init)))))
```

Below are a set of descriptions of operations on trees:

A: returns a new copy of the input tree

B: returns a pointer to the original input tree

C: sums the values of the leaves of the input tree

D: sums the values of the even-valued leaves of the input tree

E: sums the values of the odd-valued leaves of the input tree

F: reverses the top level of the input tree

G: deep reverses the input tree (i.e. reverses each level of the tree)

H: counts the numbers of leaves in the input tree

I: returns the value of the first leaf of the input tree

J: flattens the input tree (i.e., returns a single list of all the elements of the tree in order)

K: none of the above

For each expression below, identify the description in the above list that best matches its behavior.

Question 8:

```
(process-tree tr (lambda (x) x) cons '())
```

Question 9:

```
(process-tree tr x + 0)
```

Question 10:

```
(process-tree tr (lambda (x) 1) + 0)
```

Question 11:

```
(process-tree tr (lambda (x) x) (lambda (x y) (append y (list x))) '())
```

Question 12:

```
(process-tree tr list append '())
```

Part 4 (30 points)

This problem explores a small object-oriented world, consisting of Documents, Folders and Cabinets. The properties of the classes (defined by the code below) are as follows:

- a Document is an object with a name, and a number of sheets.
- a Folder is a collection of documents.
- a Cabinet is a structure that can hold Folders. It behaves as if it were a giant folder.

```
(define (create-document name sheets)
  (create-instance document name sheets))

(define (document self name sheets)
  (let ((root-part (root-object self)))
    (make-handler
     'document
     (make-methods
      'NAME    (lambda () name)
      'INSTALL (lambda () 'installed)
      'SHEETS (lambda () sheets))
     root-part)))

(define (folder self name)
  (let ((root-part (root-object self))
        (contents '()))
    (make-handler
      'folder
      (make-methods
        'NAME (lambda () name)
        'CONTENTS (lambda () contents)
        'ADD-THING (lambda (thing)
                     (set! contents (cons thing contents))
                     (map (lambda (thing) (ask thing 'NAME)) contents))
        'DOCUMENTS (lambda ()
                    (fold-right + 0
                                (map (lambda (doc) 1) contents)))
        )
      root-part)))

(define (create-folder name)
  (create-instance folder name))

(define (cabinet self name)
  (let ((root-part (root-object self))
        (folder-part (folder self name)))
    (make-handler
      'cabinet
      (make-methods
        'NAME (lambda () name)
        'CONTENTS (lambda () (ask folder-part 'contents))
        'ADD-THING (lambda (thing)
                     (ask folder-part 'add-thing thing))
        'SHEETS (lambda () 0)
        )
      folder-part root-part)))

(define (create-cabinet name)
  (create-instance cabinet name))
```

Assume the following definitions have been evaluated:

```
(define doc1 (create-document 'doc1 10))

(define doc2 (create-document 'doc2 100))

(define folder1 (create-folder 'folder1))

(ask folder1 'add-thing doc1)
(ask folder1 'add-thing doc2)

(define cab (create-cabinet 'cab))

(ask cab 'add-thing folder1)
```

What is the value of each of the following expressions, assuming they are evaluated in the order shown? (Write *unspec* for unspecified, *no-method* for an error because there is no method, *error* for some other error, or *procedure* for a procedure value.)

Question 13. (ask folder1 'DOCUMENTS)

Question 14. (ask folder1 'SHEETS)

Question 15. Add an explicit SHEETS method to Folders so that these objects will now return the total number of sheets in the documents it contains.

With that change, suppose we again evaluate:

```
(define doc1 (create-document 'doc1 10))

(define doc2 (create-document 'doc2 100))

(define folder1 (create-folder 'folder1))

(ask folder1 'add-thing doc1)
(ask folder1 'add-thing doc2)

(define cab (create-cabinet 'cab))

(ask cab 'add-thing folder1)
```

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 16. (ask folder1 'SHEETS)

Question 17. (ask cab 'SHEETS)

Suppose we remove the SHEETS method from the class definition for a cabinet.

With that change, suppose we again evaluate:

```
(define doc1 (create-document 'doc1 10))

(define doc2 (create-document 'doc2 100))

(define folder1 (create-folder 'folder1))

(ask folder1 'add-thing doc1)
(ask folder1 'add-thing doc2)

(define cab (create-cabinet 'cab))

(ask cab 'add-thing folder1)
```

What is the value of the following expression? (Write *unspec* for unspecified, *error* for error, or *procedure* for a procedure value.)

Question 18. (ask cab 'SHEETS)

Question 19:

We're now going to define a new object class `aged-cabinet`. Your job will be to fill in the code for CODE-1 below:

```
(define (create-aged-cabinet name)
        (create-instance aged-cabinet name))

(define (aged-cabinet self name)
        CODE-1)
```

The aged-cabinet class should have the cabinet class as its only superclass. It has a single additional internal state variable, `age`, which is initially set to 0. `age` will keep track of how many times a thing has been added to the cabinet; if `age` is greater than 4 then the cabinet will break.

More precisely, every time ADD-THING is called the following happens: if age is greater than 4, then 'broken is the return value, and nothing is added to the cabinet. If the age is less than or equal to 4, then the thing is added to the aged-cabinet in the usual way.

Write your code here:

Next, assume we have the following definition of the `located-object` class. This class has a pair of (x,y) coordinates as its location.

```
(define (create-located-object x y)
  (create-instance located-object x y))

(define (located-object self x y)
  (let ((root-part (root-object self)))
    (make-handler
     'located-object
     (make-methods
      'GET-X  (lambda () x)
      'GET-Y  (lambda () y)
      'SET-X! (lambda (new) (set! x new))
      'SET-Y! (lambda (new) (set! y new))
      'SET-X-Y! (lambda (newx newy) (set! x newx) (set! y newy)))
     root-part)))
```

Question 20: Now, given the definitions of `cabinet` and `located-object`, create a new class `located-cabinet` that has both `cabinet` and `located-object` as its superclasses. The new class should inherit first from `cabinet`, second from `located-object`. It should have no additional methods.

```
(define (create-located-cabinet name x y)
       (create-instance located-cabinet name x y))

(define (located-cabinet self name x y)
        CODE-2)
```

Question 21: Now add some code so that the `located-cabinet` has the following behavior whenever the x-coordinate of its location is changed:

```
(define cab (located-cabinet 'g450 10 10))

(ask cab 'set-x! 20) =>
``My location has changed.''
```

So the object displays "My location has changed." whenever `set-x!` is used.

Write code for CODE-3 that achieves this behavior (CODE-3 is part of the make-methods call within the definition of located-cabinet)

```
(define (located-cabinet self name x y)
     ...
     (make-methods 'SET-X! CODE-3)
     ...)
```

Question 22: Suppose we now do the following:

```
(define cab (located-cabinet 'g450 10 10))

(ask cab 'set-x-y! 20 30)
```

In this case, is "My location has changed." displayed?

If not, how would you change the definition of the 'set-x-y! method within `located-object` to achieve this behavior?

BACKGROUND INFORMATION—THIS PAGE CONTAINS NO QUESTIONS

```
;; Instance

; instance is a structure which holds the "self" of a normal
; object instance.  It passes all messages along to the handler
; procedure that it contains.
;

(define (make-instance)
  (list 'instance #f))

(define (instance? x)
  (and (pair? x) (eq? (car x) 'instance)))

(define (instance-handler instance) (cadr instance))

(define (set-instance-handler! instance handler)
  (set-car! (cdr instance) handler))

(define (create-instance maker . args)
  (let* ((instance (make-instance))
         (handler (apply maker instance args)))
    (set-instance-handler! instance handler)
    (if (method? (get-method 'INSTALL instance))
        (ask instance 'INSTALL))
    instance))

;;------------------------------------------------------------
;; Handler
; handler is a procedure which responds to messages with methods
; it automatically implements the TYPE and METHODS methods.

(define (make-handler typename methods . super-parts)
  (cond ((not (symbol? typename))    ;check for possible programmer errors
         (error "bad typename" typename))
        ((not (method-list? methods))
         (error "bad method list" methods))
        ((and super-parts (not (filter handler? super-parts)))
         (error "bad part list" super-parts))
        (else
         (named-lambda (handler message)
           (case message
             ((TYPE)
              (lambda () (type-extend typename super-parts)))
             ((METHODS)
              (lambda ()
                (append (method-names methods)
                        (append-map (lambda (x) (ask x 'METHODS))
                                    super-parts))))
             (else
              (let ((entry (method-lookup message methods)))
                (if entry
                    (cadr entry)
                    (find-method-from-handler-list message super-parts)))))))))

(define (handler? x)
  (and (compound-procedure? x)
       (eq? 'handler (lambda-name (procedure-lambda x)))))

(define (->handler x)
  (cond ((instance? x)
         (instance-handler x))
        ((handler? x)
         x)
        (else
         (error "I don't know how to make a handler from" x))))

; builds a list of method (name,proc) pairs suitable as input to make-handler

(define (make-methods . args)
  (define (helper lst result)
    (cond ((null? lst) result)

          ; error catching
          ((null? (cdr lst))
           (error "unmatched method (name,proc) pair"))
          ((not (symbol? (car lst)))
           (if (procedure? (car lst))
               (pp (car lst)))
           (error "invalid method name" (car lst)))
          ((not (procedure? (cadr lst)))
           (error "invalid method procedure" (cadr lst)))

          (else
           (helper (cddr lst) (cons (list (car lst) (cadr lst)) result)))))
  (cons 'methods (reverse (helper args '()))))

(define (method-list? methods)
  (and (pair? methods) (eq? (car methods) 'methods)))

(define (empty-method-list? methods)
  (null? (cdr methods)))

(define (method-lookup message methods)
  (assq message (cdr methods)))

(define (method-names methods)
  (map car (cdr methods)))

;;------------------------------------------------------------
;; Root Object
```

```
; Root object.  It contains the IS-A method.
; All classes should inherit (directly or indirectly) from root.
;
(define (make-root-object self)
  (make-handler
   'root
   (make-methods
    'IS-A
    (lambda (type)
      (memq type (ask self 'TYPE))))))

;;------------------------------------------------------------
;; Object Interface

; ask
;
; We "ask" an object to invoke a named method on some arguments.
;
(define (ask object message . args)
  ;; See your Scheme manual to explain `. args' usage
  ;; which enables an arbitrary number of args to ask.
  (let ((method (get-method message object)))
    (cond ((method? method)
           (apply method args))
          (else
           (error "No method for" message 'in
                  (safe-ask 'UNNAMED-OBJECT
                            object 'NAME))))))

; Safe (doesn't generate errors) method of invoking methods
; on objects.  If the object doesn't have the method,
; simply returns the default-value.  safe-ask should only
; be used in extraordinary circumstances (like error handling).
;
(define (safe-ask default-value obj msg . args)
  (let ((method (get-method msg obj)))
    (if (method? method)
        (apply ask obj msg args)
        default-value)))

;;--------------------
;; Method Interface
;;
;; Objects have methods to handle messages.

; Gets the indicated method from the object or objects.
; This procedure can take one or more objects as
; arguments, and will return the first method it finds
; based on the order of the objects.
;
```

```
(define (get-method message . objects)
  (find-method-from-handler-list message (map ->handler objects)))

(define (find-method-from-handler-list message objects)
  (if (null? objects)
      (no-method)
      (let ((method ((car objects) message)))
        (if (not (eq? method (no-method)))
            method
            (find-method-from-handler-list message (cdr 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)))

; used in make-handler to build the TYPE method for each handler
;
(define (type-extend type parents)
  (cons type
        (remove-duplicates
         (append-map (lambda (parent) (ask parent 'TYPE))
                     parents))))
```

BACKGROUND INFORMATION—THIS PAGE CONTAINS NO QUESTIONS