MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 2006

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. Also note that while there may be a lot of reading to do on a problem, there is relatively little code to write, so please take the time to read each problem carefully.

NAME:

Section Time: Tutor's Name:

PART	Value	Grade	Grader
1	23		
2	18		
3	20		
4	20		
5	19		
Total	100		

For your reference, the TAs are:

- Arnab Bhattacharyya
- Austin Clements
- Estanislao Fidelholtz
- Harold Fox
- Sumudu Watugala
- Tom Wilson
- Tom Yeh

Part 1: (23 points)

For computations that do not involve mutation, it can sometimes be efficient to remember previous computations of a procedure, for example, storing the value obtained by applying a procedure to a particular argument. This technique, called **memoization**, allows us to avoid redoing the same computation at subsequent stages.

Here is some code for memoizing a procedure of one numeric argument, together with an example use:

```
(define (memoize proc)
  (let ((hist '()))
    (lambda (arg)
      (let ((prev (its-in arg hist)))
        (if prev
            prev
            (let ((new (proc arg)))
              (set! hist (cons (list arg new) hist))
              new))))))

(define (its-in arg hist)
  (cond ((null? hist) #f)
        ((equal? arg (caar hist))
         (cadar hist))
        (else (its-in arg (cdr hist)))))

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

(my-sq 5)
```

Attached at the end of the exam is a partially completed environment diagram corresponding to this set of evaluations. Complete that diagram, and then use the labels on the parts of the environment diagram to answer the following questions.

Question 1: For each of the following frames, indicate the enclosing environment, choosing one of **GE, E1, E2, E3, E4, E5, E6, E7** or **none** or **not shown**.

Frame:	Enclosing Environment
GE	
E1	
E2	
E3	
E4	
E5	
E6	
E7	

Question 2: For each of the following procedure objects, indicate the enclosing environment, choosing one of **GE, E1, E2, E3, E4, E5, E6, E7** or **none** or **not shown**.

Procedure:	Enclosing Environment
P1	
P2	
P3	
P4	

Question 3: For each of the following variable names, indicate the value to which it is bound at the end of the evaluation of the expressions, choosing one of **GE, E1, E2, E3, E4, E5, E6, E7** or **P1, P2, P3, P4** or a symbol, a number, a boolean value, a list or the empty list.

Variable:	Environment	Value
memoize	GE	
its-in	GE	
my-sq	GE	
proc	E1	
new	E2	
arg	E3	
prev	E4	
arg	E5	
hist	E5	
hist	E6	
x	E7	

Part 2: (18 points)

Consider the following two procedures for processing trees:

```
(define (tree-map proc tree)
  (cond ((leaf? tree) (proc tree))
        (else (map (lambda (x) (tree-map proc x)) tree))))

(define (leaf? x) (not (list? x)))

(define (tree-fold leaf-op combine init tree)
  (if (leaf? tree)
      (leaf-op tree)
      (fold-right combine init
                  (map (lambda (x) (tree-fold leaf-op combine init x))
                       tree))))

(define (map proc lst)
  (if (null? lst)
      '()
      (cons (proc (car lst)) (map proc (cdr lst)))))

(define (fold-right proc init lst)
  (if (null? lst)
      init
      (proc (car lst) (fold-right proc init (cdr lst)))))
```

As an example, here is a test tree:

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

Question 4: Write an expression using `tree-map` that will double the values of all the leaves of a tree (you may assume that the leaves only hold numerical values), e.g.,

```
>(tree-map ... test)
(2 (4 (6 (8) 10) 12) 14)
```

Question 5: Write an expression using `tree-map` that makes a copy of a tree, e.g.,

```
>(tree-map ... test)
(1 (2 (3 (4) 5) 6) 7)
```

Question 6: Write an expression using `tree-fold` that makes a copy of a tree, e.g.,

```
>(tree-fold ... ... ... test)
(1 (2 (3 (4) 5) 6) 7)
```

Question 7: Write an expression using `tree-fold` that will flatten a tree, e.g.,

```
>(tree-fold ... ... ... test)
(1 2 3 4 5 6 7)
```

Question 8: Write an expression using `tree-fold` that counts the number of leaves in a tree, e.g.,

```
>(tree-fold ... ... ... test)
7
```

Question 9: Write an expression using `tree-fold` that returns `#t` if all of the leaves of the tree are odd (you may assume that `odd?` is defined, and that all the values are numbers), e.g.,

```
>(tree-fold ... ...  ... test)
#f
```

Part 3: (20 points)

Earlier in the term, we saw a set of procedures for processing lists, in particular, `map`, `filter`, `append`, and `reverse`. Now that we have introduced mutation, it is possible to create versions of these procedures that alter the list structure of their argument, rather than create a new version of the list structure. For each of the questions below, provide the code fragments necessary to complete these mutating versions of list procedures.

Question 10:

The procedure `map!` should mutate its list of arguments, replacing each value with the result of applying the supplied procedure. Here is a template:

```
(define (map! f lst)
  (define (lter l)
    (cond ((null? l) lst)
          (else ANSWER-10)))
  (iter lst))
```

To complete this, you need to provide an expression or expressions for ANSWER-10

Question 11:

The procedure **append!** should create a single list from two lists, by mutation. It's behavior is shown below:

```
>(define x (list 1 2 3))
>(define y (list 4 5 6))

>(append! '() x)
(1 2 3)

>(append! x y)
(1 2 3 4 5 6)

>x
(1 2 3 4 5 6)

>y
(4 5 6)
```

Here is a partially completed procedure

```
(define (append! a b)
  (define (iter first-lst second-lst)
    ANSWER-11)
  (cond ((null? a) b)
        (else (iter a b)
              a)))
```

To complete this, you need to provide an expression for ANSWER-11

Question 12:

The procedure `reverse!` should reverse the elements of its argument "in place", that is by mutating the existing list structure, rather than creating a new copy. It should do this by working down the list, keeping track of the last cons cell and the current cons cell, and mutating the structure so that the current cell points to the last one (i.e. reversing the order) while being careful to keep track of the rest of the list. When you are done, the original list may be destroyed (see example below):

```
>(define x (list 1 2 3 4 5))

>(reverse! x)
(5 4 3 2 1)

>x
(1)
```

Here is a partially completed procedure

```
(define (reverse! lst)
  (define (iter last current)
     (if (null? current)
         last
         ANSWER-12))
  (iter '() lst))
```

To complete this, you need to provide an expression for ANSWER-12

Part 4: (20 points)

A **queue** is a data structure, in which elements can only be added at the end of the queue, and elements can only be removed from the front of the queue (kind of like the line for LSC movies, assuming people behave courteously!). Traditionally, a queue is a tagged data structure where the elements are stored in a list, and the queue provides pointers to the front and end of the list. We are going to build a different implementation of a queue, in which the elements are represented by objects with state, including internal state variables pointing to the next element in the queue.

Here is part of the framework for such an implementation of a queue.

```
(define (make-empty-queue)
  (list 'queue #f #f))    ; list of a tag, a pointer to the front of
                          ; the queue, and a pointer to the end of the queue

(define (queue? q)
  (tagged-list q 'queue))

(define (make-entry element)
  (let ((next #f))
    (lambda (msg)
      (cond ((eq? msg 'value) element)
            ((eq? msg 'next) next)
            ((eq? msg 'set-next!)
             (lambda (val) (set! next val)))))))

(define (insert-queue element q)
  ;; procedure to add entry to end of queue
  (let ((current-rear (caddr q))
        (new (make-entry element)))
    (cond ((not current-rear)     ;;the queue is currently empty
           ANSWER-13
           ANSWER-14)
          (else
           ANSWER-15
           ANSWER-16))))

(define (delete-queue q)
  ;; procedure to remove entry from front of queue,
  ;; and return value of that entry
  (if (not (cadr q))
      (error "empty queue")
      ANSWER-17))
```

Question 13: What expression(s) should be used for ANSWER-13 to ensure that the front of the queue points to the added element?

Question 14: What expression(s) should be used for ANSWER-14 to ensure that the rear of the queue points to the added element?

Question 15: What expression(s) should be used for ANSWER-15 to ensure that the element currently at the rear of the queue includes information pointing to the new element as the next element?

Question 16: What expression(s) should be used for ANSWER-16 to ensure that the rear of the queue correctly points to the newly added element?

Question 17: When we delete an element from a non-empty queue, we need to ensure that the front of the queue points to the next element after the current front of the queue, and that we return the value held in the current front element of the queue. What expression(s) should be used for ANSWER-17 to ensure this?

Part 5: (19 points)

A stack is a data structure for a sequence of elements, with the property that elements can only be "pushed" onto the top of the stack, and "popped" off the top of the stack.

Here is a partial implementation of a stack, using a tagged data structure:

```
(define (make-empty-stack)
  (list 'stack '()))  ;; a stack is a pair: a tag, and a list of elements

(define (tagged-list? element tag)
  (if (pair? element)
      (eq? (car element) tag)))

(define (stack? st)
  (tagged-list? st 'stack))

(define (push element stack)
  (if (stack? stack)
      (set-car! (cdr stack) (cons element (cdr stack)))))
```

Question 18: The procedure pop should take as an argument a stack, and should alter the stack to remove the first element, and return that value. If the stack is empty, it should return an error.

```
(define (pop stack)
  (if (stack? stack)
      ANSWER-18
      (error "not a stack")))
```

Write the expression(s) needed for ANSWER-18.

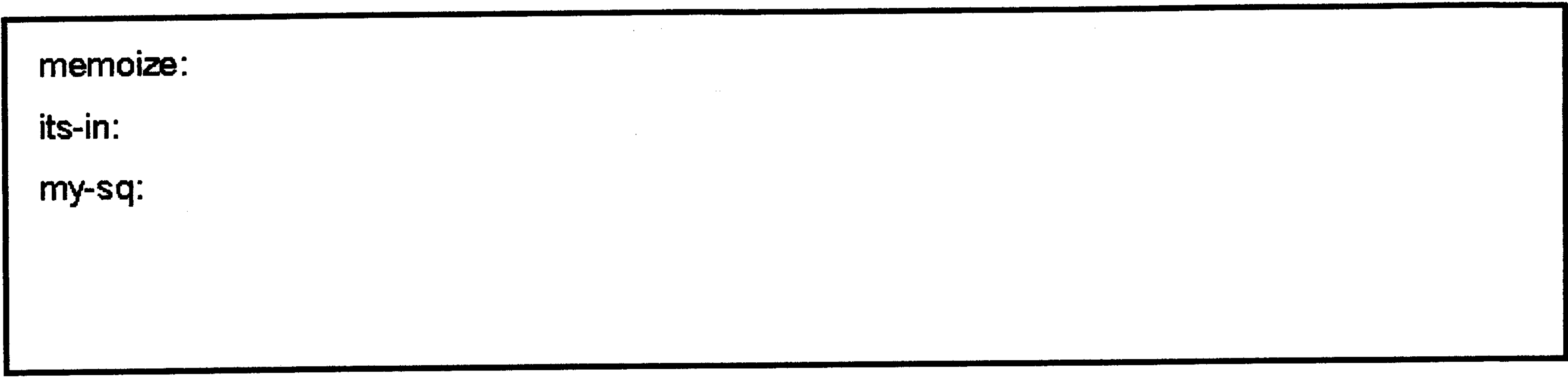
Question 19: We can create a different version of a stack, by requiring that the elements of the stack are ordered, based on some comparison predicate. For example:

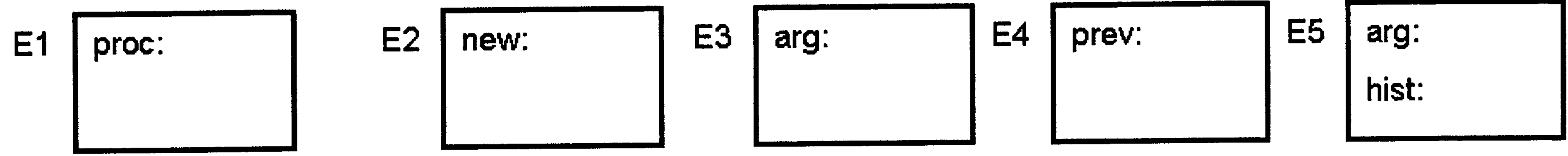
```
> (define test (make-empty-stack))
> (ordered-push '(1 a) test (lambda (x y) (< (car x) (car y))))
> (ordered-push '(0 b) test (lambda (x y) (< (car x) (car y))))

> test
(stack (0 b) (1 a))

> (pop test)
(0 b)
```

Here is a template for the procedure `ordered-push`:

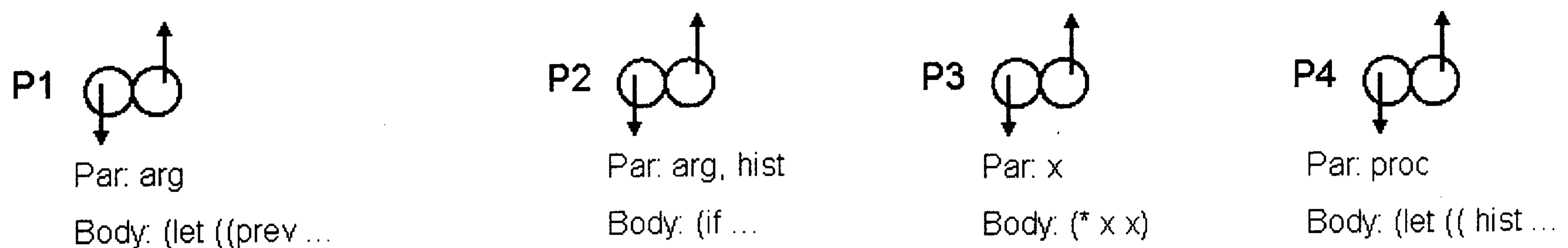
```
(define (ordered-push element stack less?)
  (if (stack? stack)
      (set-car! (cdr stack) (insert-ordered less? element (cdr stack)))))
```

Write the procedure `insert-ordered`:


GE
memoize:
its-in:
my-sq:


E1
proc:
E2
new:
E3
arg:
E4
prev:
E5
arg:
hist:

E6
hist:
E7
x:


P1
Par: arg
Body: (let ((prev ...
P2
Par: arg, hist
Body: (if ...
P3
Par: x
Body: (* x x)
P4
Par: proc
Body: (let ((hist ...