Question	Possible Points	Points Awarded
0 - Name	1	
1 - Lists	9	
2 - OOP	8	
3 - Scheme-1	6	
4 - DDP / Message Passing	6	
5 - Trees	10	

QUESTION 0 (1 points)
Fill out the following information correctly:

Your name:

Your SID:

Your TA's name and/or your section number:

Your login:

The neighbor to your right's login:

The neighbor to your left's login:

Question 1 (lists) (9 points) What will Scheme ***print*** in response to the following expressions? If error, write "error". Also, **draw** the corresponding box-and-pointer diagram.

```
(let ((x (list 1 2)))
 (cons x (list (cdr x))))
```

```
(append (list 1 2) (list (cons 3 4)))
```

```
(filter pair? (list '() 1 (cons 2 3) (list 4 5)))
```

Question 2 (OOP) (8 points)
Suppose we simulate Berkeley's schools and colleges using OOP. We then define classes for professors, departments, and classes.

```
(define-class (department name)
   ...)
(define-class (class name department)
   ...)
(define-class (professor name department favorite-class)
   ...)
```

For each of the following, state whether it would best be implemented as which of the following, **and** state which class it corresponds to.

- object / instance of a class
- instance variable
- instantiation variable
- class variable
- child class

a) Professor Kahan

b) A list of all classes in a given department

c) Seminars

d) A professor's favorite saying

e) Assistant professors

```
(define-class (department name)
 (instance-vars (all-classes '())
                (all-professors '()))
 (method (add-professor professor)
   (set! all-professors (cons professor all-professors)))
 (method (add-class class)
   (set! all-classes (cons class all-classes))) )
```

Define the professor class so that every time a professor is created, the professor's department is updated accordingly.

```
(define-class (professor name department favorite-class)
```

Question 3 (Scheme-1) (6 points)

The following is the original code for `eval-1` and `apply-1`.

```
(define (eval-1 exp)
  (cond ((constant? exp) exp)
        ((symbol? exp) (eval exp)) ; use underlying Scheme's EVAL
        ((quote-exp? exp) (cadr exp))
        ((if-exp? exp)
         (if (eval-1 (cadr exp))
             (eval-1 (caddr exp))
             (eval-1 (cadddr exp))))
        ((lambda-exp? exp) exp)
        ((pair? exp) (apply-1 (eval-1 (car exp))      ; eval the operator
                              (map eval-1 (cdr exp))))
        (else (error "bad expr: " exp))))

(define (apply-1 proc args)
  (cond ((procedure? proc) ; use underlying Scheme's APPLY
         (apply proc args))
        ((lambda-exp? proc)
         (eval-1 (substitute (caddr proc)   ; the body
                             (cadr proc)    ; the formal parameters
                             args           ; the actual arguments
                             '()))     ; bound-vars, see below
        (else (error "bad proc: " proc))))
```

Suppose we wrote another version of Scheme evaluator called Scheme-1.1. It is the same as Scheme-1 except a part in `eval-1` as follows.

```
(define (eval-1 exp)
  (cond ...
        ...
        ((pair? exp) (apply-1 (eval-1 (car exp))      ; eval the operator
                              (map EVAL (cdr exp))))  ; USE STk EVAL INSTEAD
        ...
```

Give an example of expression that evaluates to different results under Scheme-1 and Scheme-1.1. Any kinds of errors for both versions are considered same results. An error in one version and no error in the other version are considered different results.

Expression:

Result in Scheme-1:

Result in Scheme-1.1:

Question 4 (Data-directed programming and Message Passing)(6 points)
Consider the following game (the details are not important, so don't try and memorize them or dwell on them):

board: 8 by 8 grid (like a chess or checker board)

pieces:

- yellow: moves horizontally and vertically many spaces; eats other pieces one space away horizontally or vertically
- blue: moves diagonally many spaces; eats pieces two spaces away diagonally
- green: has to jump over another piece to move, and can only move horizontally and vertically; eats other pieces one space away diagonally

We have implemented part of the game using data-directed programming:

		piece type		
		yellow	*blue*	*green*
operation	*move*	yellow-move	blue-move	green-move
	eat	yellow-eat	blue-eat	green-eat

Suppose we implemented the above code using message passing instead of data-directed programming.

a) What is represented with a dispatch function? (circle one) each piece type/each operation

b) Select one of the dispatch functions you would need to implement the code above using message passing and implement it. You may use the entries in the table above in your code.

Question 5 (trees) (10 points)
Recall the Tree ADT (abstract data type) defined in lecture (shown below). Write a procedure `max-leaf-datum` that takes a Tree of numbers as argument and returns the greatest leaf datum in the tree. You may use `max`, which takes two number arguments and returns the maximum of the two arguments.

```
(define make-tree ...)
(define datum ...)
(define children ...)

(define (leaf? node)
  (null? (children node)) )
```

```
(define (max-leaf-datum tree)
```

```
)
```