Question	Possible Points	Points Earned
1 - Information	1	
2 - Lists	6	
3 - Environment Diagram	8	
4 - Metacircular Evaluator	9	
5 - Concurrency	7	
6 - Vectors	9	
Total	40	

1 Information

Fill out the following information correctly:

Name:

Your SID:

Login: cs61a-

Your TA's name:

Write your name and login at the top of each page.

2 Lists (6 Points)

For the following expressions, write what Scheme will print, and draw the box and pointer diagram for the result. Make sure to label what pairs correspond to what variables (i.e. show what x,y,v,z and a point to, etc.!) Hint: It might be easier to draw the box-and-pointer diagrams first!

```
(let ((x (list 1 2 3))
      (y (list 4 5 6)))
  (set-cdr! (cddr y) (cdr x))
  (set-car! (cdddr y) y)
  y)
```

```
(define v (list 'a 'b 'c 'd))
(define z (cons v v))
(set-cdr! (cdar z) '())
z
```

```
(define a (list (list 0) (list 5) (list 10) (list 15)))
(set-cdr! (caddr a) (cdddr a))
(set-car! a 30)
a
```

3 Environment Diagrams (8 Points)

We have seen that `set!` changes the binding of variables. Suppose that we want to allow undoing the change (for only one step) with the following code.

```
(define (set!-start val)
  (define old #f)
  (let ((current val))
    (lambda (msg . arg)
      (cond ((eq? msg 'value) current)
            ((eq? msg 'new)
             (set! old current)
             (set! current (car arg)))
            ((eq? msg 'undo)
             (set! current old)
             (set! old #f))
            (else (error "invalid message " msg))))))
```

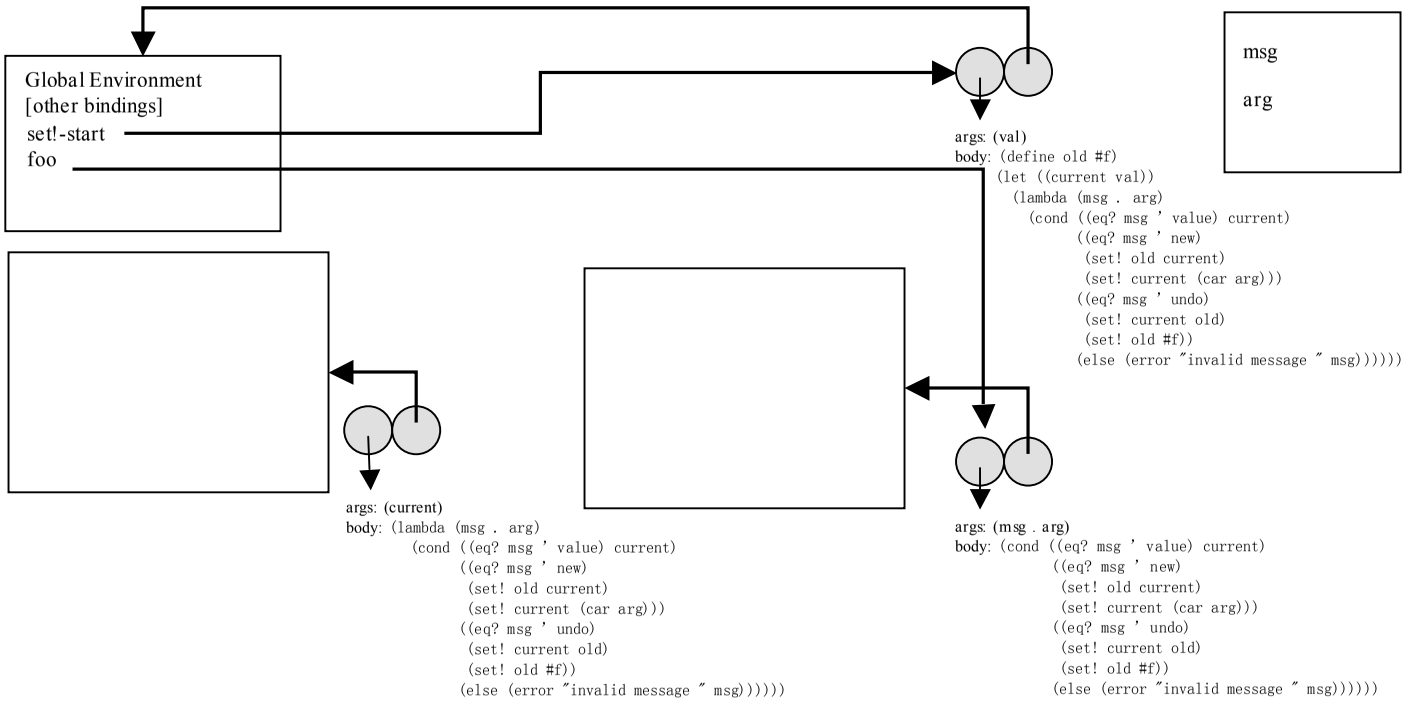
The variables are essentially procedures in this case. We act on a variable by passing a message, for example:

```
> (define foo (set!-start 5))
foo
> (foo 'value)
5
> (foo 'new 10)
okay
> (foo 'value)
10
> (foo 'undo)
okay
> (foo 'value)
5
```

Below is an incomplete environment diagram for when the definition of procedure `set!-start` and the following code are entered in a fresh environment. (Note that the example above is *not* evaluated.)

```
(define foo (set!-start 5))
(foo 'new 10)
```

Complete the diagram by adding arrows to enclosing frames and variable bindings in each frame.


Global Environment
[other bindings]
set!-start
foo
msg
arg
args: (val)
body: (define old #f)
(let ((current val))
(lambda (msg . arg)
(cond ((eq? msg ' value) current)
((eq? msg ' new)
(set! old current)
(set! current (car arg)))
((eq? msg ' undo)
(set! current old)
(set! old #f))
(else (error "invalid message " msg))))))
args: (current)
body: (lambda (msg . arg)
(cond ((eq? msg ' value) current)
((eq? msg ' new)
(set! old current)
(set! current (car arg)))
((eq? msg ' undo)
(set! current old)
(set! old #f))
(else (error "invalid message " msg))))))
args: (msg . arg)
body: (cond ((eq? msg ' value) current)
((eq? msg ' new)
(set! old current)
(set! current (car arg)))
((eq? msg ' undo)
(set! current old)
(set! old #f))
(else (error "invalid message " msg))))))

4 Metacircular Evaluator (9 points)

Suppose we change the application case in `mc-eval` as follows

Old code:

```
(define (mc-eval exp env)
  (cond ...
        ((application? exp)
         (mc-apply (mc-eval (operator exp) env)
                   (list-of-values (operands exp) env)))
   ...))
```

New code:

```
(define (mc-eval exp env)
  (cond ...
        ((application? exp)
           (let ((procedure (mc-eval (operator exp) env)))
              (mc-apply procedure
                        (list-of-values (operands exp) (procedure-environment procedure)))))
   ...))
```

Below is mc-apply, no changes have been made.

```
(define (mc-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else ... )))
```

For each set of expressions in the table below, fill in what the value for the last expression would be with the old `mc-eval` code, and the new `mc-eval` code. If there would be an error, write ERROR. Assume let has been implemented properly and that the metacircular evaluator is restarted each time.

Expression	Old `mc-eval`	New `mc-eval`

`(define x 7)`
`(define (foo x)`
`  (lambda (y) (+ x y)))`
`((foo 10) x)`		

`(define (foo x)`
`  (lambda (y) (+ x y)))`
`((foo 10) x)`		

`(define (foo x)`
`  (lambda (y) (+ x y)))`
`(let ((z 7))`
`  ((foo 10) z))`		

5 Concurrency (7 Points)

Our friend Ben Bitdiddle complained about the inefficiency in the BART system. (In case you're new to Berkeley, BART is a subway system in the San Francisco Bay area.) He believes that we could have saved millions on the construction of the train tracks by building a single track (instead of two tracks) between some stations. Take for example the tracks between the Downtown Berkeley station and the Ashby station. He snuck into the tunnel and observed that no Ashby-Berkeley train ever passes a Berkeley-Ashby train, so they can actually just share a single track.

Iwanna Passe doubts if Ben's observation about the trains is correct. Perhaps there are a few trains passing between Ashby and Berkeley. We surely don't want them running into each other. She proposes that we need some kind of signaling before using the track. Assume that each train is equipped with a computer that connects to a server. The server maintains a state variable

$$\texttt{track-in-use} = \begin{cases} \texttt{\#f} & \text{no trains on the track} \\ \texttt{\#t} & \text{a train is on the track} \end{cases}$$

Each train checks the state of the track with the server before departing the station. If `track-in-use` appears to be `#f`, the train will ask the server to change it to `#t` and move the train to the next station. If `track-in-use` is not `#f`, it will wait at the station and keep checking. Once the train arrives at the next station, it will tell the server to set `track-in-use` back to `#f`.

To quickly illustrate the idea to her prospective employers, Iwanna abstracts out the networking component and assumes that trains are parallel processes sharing the variable `track-in-use`. Here is her demo code.

```
(define track-in-use #f)

(define (check-and-go)
  (if (ask-server 'is-track-available?)
      (begin (ask-server 'occupy-track)
             (run-to-next-station) ; this takes a few minutes
             (ask-server 'done-with-track))
      (check-and-go)))             ; keep checking until the track is available

(define (ask-server cmd . data)
  (cond ((eq? cmd 'is-track-available?) (not track-in-use))
        ((eq? cmd 'occupy-track) (set! track-in-use #t))
        ((eq? cmd 'done-with-track) (set! track-in-use #f))
        (else (error "server doesn't understand " cmd))))

(define A->B-train (lambda () (check-and-go)))
(define B->A-train (lambda () (check-and-go)))
(parallel-execute A->B-train B->A-train)
```

A) Iwanna shows her codes to Prof. May B. Right, who immediately notices a disaster. Explain what the disaster is and explain how it can occur.

B) Prof. Right asks her graduate students to help Iwanna. The first student, Louis Reasoner, suggests the following fix. The differences from Iwanna's original code are in capital letters.

```
(define track-in-use #f)
(DEFINE PROTECTOR (MAKE-SERIALIZER))

(define (check-and-go)
  (if (ask-server 'is-track-available?)
      (begin (ask-server 'occupy-track)
             (run-to-next-station) ; this takes a few minutes
             (ask-server 'done-with-track))
      (check-and-go)))             ; keep checking until the track is available

(define (ask-server cmd . data)
  (cond ((eq? cmd 'is-track-available?) (not track-in-use))
        ((eq? cmd 'occupy-track) (set! track-in-use #t))
        ((eq? cmd 'done-with-track) (set! track-in-use #f))
        (else (error "server doesn't understand " cmd))))

(define A->B-train (PROTECTOR (lambda () (check-and-go))))
(define B->A-train (PROTECTOR (lambda () (check-and-go))))
(parallel-execute A->B-train B->A-train)
```

Is Louis' fix correct and efficient? Explain concisely.

Nothing below this line will be graded.

C) Another grad student, Lem E. Tweakit, proposes the following fix. The differences from Iwanna's original code are in capital letters.

```
(define track-in-use #f)
(DEFINE PROTECTOR (MAKE-MUTEX))

(define (check-and-go)
  (if (ask-server 'is-track-available?)
      (begin (ask-server 'occupy-track)
             (run-to-next-station) ; this takes a few minutes
             (ask-server 'done-with-track))
      (check-and-go)))             ; keep checking until the track is available

(define (ask-server cmd . data)
  (PROTECTOR 'ACQUIRE)
  (LET ((RESULT (cond ((eq? cmd 'is-track-available?) (not track-in-use))
                      ((eq? cmd 'occupy-track) (set! track-in-use #t))
                      ((eq? cmd 'done-with-track) (set! track-in-use #f))
                      (else (PROTECTOR 'RELEASE)
                            (error "server doesn't understand " cmd)))))
    (PROTECTOR 'RELEASE)
    RESULT))

(define A->B-train (lambda () (check-and-go)))
(define B->A-train (lambda () (check-and-go)))
(parallel-execute A->B-train B->A-train)
```

Is Lem's fix correct and efficient? Explain concisely.

Nothing below this line will be graded.

6 Vectors (9 points)

Write a function, `scalar-mult!` that takes a matrix, represented as a vector of vectors, and an integer, and multiplies each element in the matrix by the integer. You may assume the vector of vectors forms a proper matrix (all of the inner vectors are of the same length). You may use the vector procedures below. Any other vector procedure you would like to use, you should implement. You may not use `list->vector` and `vector->list`.

```
(vector-ref vec index)
(vector-set! vec index value)
(vector-length vec)
```

For example:

$$\text{mat} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} =$$

```
(define v1 (vector 1 2 3 4))
(define v2 (vector 5 6 7 8))
(define v3 (vector 9 10 11 12))
(define mat (vector v1 v2 v3))
```

$$\texttt{(scalar-mult! mat 2)} = \begin{bmatrix} 2 & 4 & 6 & 8 \\ 10 & 12 & 14 & 16 \\ 18 & 20 & 22 & 24 \end{bmatrix}$$

`mat` before

```
#(#(1 2 3 4) #(5 6 7 8) #(9 10 11 12))
```

`mat` after calling `(scalar-mult! mat 2)`

```
#(#(2 4 6 8) #(10 12 14 16) #(18 20 22 24))
```

```
(define (scalar-mult! mat scalar)
```