Your name _____

TA's name _____   Discussion section number _____

A random five-digit number: _____

Circle the last two letters of your login (cs61a-<u>xx</u>)

a b c d e f g h i j k l m n o p q r s t u v w x y z 1 2 3

a b c d e f g h i j k l m n o p q r s t u v w x y z

This exam is worth 40 points, or about 13% of your total course grade. It includes two parts. The individual exam (this part) is worth 35 points, and the group exam is worth 5 points. The individual exam contains six substantive questions, plus the following:

Question 0 (1 point): Fill out this front page correctly and correctly copy your random five-digit number to the top of each of the following pages. (This is to make sure the pages of your exam stay together even if the staple comes out.)

This booklet contains ten numbered pages including the cover page. Put all answers on these pages, please; don't hand in stray pieces of paper. This is an open book exam.

When writing procedures, don't put in error checks. Assume that you will be given arguments of the correct type.

Our expectation is that many of you will not complete one or two of these questions. If you find a question especially difficult, leave it for later; start with the ones you find easier.

If you want to use procedures defined in the book or reader as part of your solution to a programming problem, you must cite the page number on which it is defined so we know what you think it does.

READ AND SIGN THIS:

I certify that my answers to this exam are all my own work, and that I have not discussed the exam questions or answers with anyone prior to taking this exam.

If I am taking this exam early, I certify that I shall not discuss the exam questions or answers with anyone until after the scheduled exam time.

_____

0		/1
1-2		/11
3		/3
4		/6
5		/5
6		/9
total		/35

Question 1 (6 points):

What will the Scheme interpreter print in response to each of the following expressions? If any expression results in an error or contains a loop, just write "Error". Also, draw a "box and pointer" diagram for the result of each expression. Hint: It'll be a lot easier if you draw the box and pointer diagram *first*!

```
> (let ((x (list 1 2 3 4)))
    (set-cdr! (cddr x) (car x))
    x)
```

```
> (let ((y (list 1 2 3 4)))
    (set-car! (cddr y) (cddddr y))
    y)
```

Question 1 continues on the next page.

Question 1 continued:

```
> (let ((z (list 1 2 3 4)))
    (set-car! (cddr z) z)
    z)
```

_____

Question 2 (5 points):

Show the first five elements of these two streams.

```
> (define (madness x y)
    (if (even? x)
        (- x y)
        y))
> (define boss (cons-stream 1 (stream-map madness boss truck)))
> (define truck (cons-stream 3 (stream-map + boss truck)))
```

boss   _____   _____   _____   _____   _____

truck   _____   _____   _____   _____   _____

Question 3 (3 points):

Here is a class definition, implemented in ordinary Scheme instead of using `define-class`:

```
(define make-thingo
  (let ((foo 'yakko))
    (lambda (baz)
      (let ((garply 'wakko) (floop (make-bear)))
        (lambda (zot)
          (cond ((eq? zot 'yes)
                 (lambda (xyzzy)
                   (list xyzzy baz)))
                ((eq? zot 'no)
                 (lambda () garply))
                (else (floop zot)))))))))
```

For each symbol in the table below, write the letter of the kind of thing it is in object-oriented terminology. Each letter should be used exactly once.

_____ foo   (A) parent

_____ baz   (B) class variable

_____ garply   (C) method argument

_____ floop   (D) instance variable

_____ zot   (E) instantiation variable

_____ xyzzy   (F) message

4

Question 4 (6 points):

Write a procedure `duplicate-elements!` that takes a list and duplicates its elements, using mutation. You may create new pairs, but every pair in the original list must still be part of the list at the end. The return value of `duplicate-elements!` is unimportant.

Here are some examples of how `duplicate-elements!` should work:

```
> (define a (list))              > (define b (list 1))
> (duplicate-elements! a)        > (duplicate-elements! b)
> a                              > b
()                               (1 1)

> (define c (list 1 2 3))        > (define d (list (list 1 2)))
> (duplicate-elements! c)        > (duplicate-elements! d)
> c                              > d
(1 1 2 2 3 3)                    ((1 2) (1 2))
```

```
(define (duplicate-elements! lst)
```

Question 5 (5 points):

We want to add functions that take any number of arguments to the metacircular evaluator. The syntax for this is based on how Scheme does it: use a single name (without parentheses) instead of a list of names for a procedure's formal parameters. This allows us to use procedures like `new-odds` (left), in addition to `old-odds` (right).

```
; The new way                        ; The existing way
; Look, no parentheses!              ; (which should still work)
> (define new-odds                   > (define old-odds
    (lambda nums                         (lambda (nums)
      (filter odd? nums) ))              (filter odd? nums) ))
> (odds 7 8 4 5 6 7)                  > (odds (list 7 8 4 5 6 7))
(7 5 7)                              (7 5 7)
```

Your code should handle the following kinds of expressions:

```
(lambda (x y z) ...)  ; existing lambda with a fixed number of arguments
(lambda args ...)     ; new lambda with no parentheses
```

Don't worry about handling expressions like this:

```
(lambda (x y . args) ...)  ; real Scheme lets you mix the two this way
```

Here are some relevant procedures from the original Metacircular Evaluator.

```
 1. (define (mc-apply procedure arguments)
 2.   (cond ((primitive-procedure? procedure)
 3.          (apply-primitive-procedure procedure arguments))
 4.         ((compound-procedure? procedure)
 5.          (eval-sequence
 6.            (procedure-body procedure)
 7.            (extend-environment (procedure-parameters procedure)
 8.                                arguments
 9.                                (procedure-environment procedure)))))
10.         (else (error "Unknown procedure type -- APPLY" procedure))))
11.
12. (define (extend-environment vars vals base-env)
13.   (if (= (length vars) (length vals))
14.       (cons (make-frame vars vals) base-env)
15.       (if (< (length vars) (length vals))
16.           (error "Too many arguments supplied" vars vals)
17.           (error "Too few arguments supplied" vars vals))))
18.
19. (define (make-frame variables values)
20.   (cons variables values))
```

Question 5 continued:

Change `mc-apply`, `extend-environment`, and/or `make-frame` to make this work. You do not need to worry about error handling.

For this problem, use the sections below to show which lines from the program on the previous page you are changing. If you want to replace a single line, write the same number in both spaces. You are not required to use both sections.

Replace lines _____ through _____ (inclusive) with the following:

Replace lines _____ through _____ (inclusive) with the following:

Question 6 (9 points):

Vectors are created with a fixed size; to add an element to the end of a vector we have to create a new vector with size $n + 1$.

To avoid this, consider the following ADT called `vector-chain`. A `vector-chain` is essentially a sequence with no size limit. You can set any element, and it **automatically grows** to be big enough to fit that element.

```
> (define vc (make-vector-chain))
> (vc-set! vc 28 'hello)
okay
> (vc-ref vc 28)
hello
```

The pictures to the right show the structure of `vc` just before the call to `vc-set!` (top) and just after the call to `vc-set!` (bottom). Notice how the vector chain has automatically grown, and the 28th cell now contains `hello`.
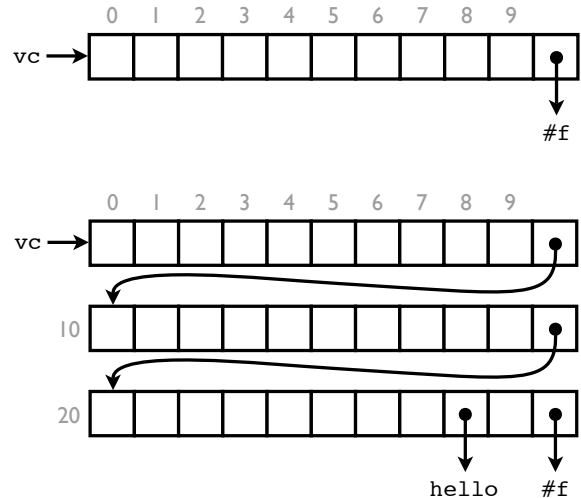
Here's how it works: A `vector-chain` is made up of several **11-element vectors**. The **last vector's** 11th element is `#f`. The 11th element in every other vector points to the **next vector** in the chain.

You can see this in the second picture (above). The last element of the first vector points to the entire second vector, the last element of the second vector points to the entire third vector, and the last element of the third vector is `#f`.

Here is the constructor for a `vector-chain`:

```
(define (make-vector-chain)
  (let ((result (make-vector 11)))
    (vector-set! result 10 #f)
    result))
```

In this problem, you will implement `vc-set!`. But before we can write `vc-set!`, we need a way to extend the chain if we don't have enough space.
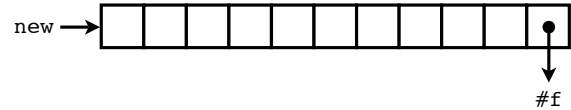
Question 6 continues on the next page.
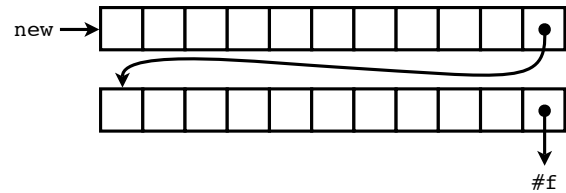
Question 6 continued:

(a) Write a helper procedure `vc-extend!` that adds one more 11-element vector to a vector chain, making sure the new vector's last element is `#f`. The argument to `vc-extend!` is the last vector in an existing vector chain.

The pictures to the right show the structure of `new` after each line in the following example:
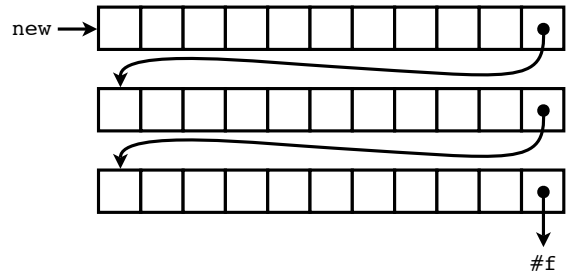
```
> (define new (make-vector-chain))
```

```
> (vc-extend! new)
okay
```

```
> (vc-extend! (vector-ref new 10))
okay
```

```
(define (vc-extend! last-in-chain)
```

Question 6 continues on the next page.

9

Question 6 continued:

(b) Now write `vc-set!`. Use `vc-extend!` to extend the chain as necessary to ensure that there is always enough space.

```
(define (vc-set! vc index value)
```