CISC280 Midterm Learning Experience 1, Spring 2006

NAME _____

General instructions:

Turn off all noise-making electronic devices, such as cell phones. Disturbance by such a device during the exam may result in a penalty not to exceed a full letter grade.

You may leave the classroom once the exam has begun, but you may not return.

There are 7 pages worth a total of points, and an extra credit problem worth 3 points. Read the problems very carefully. Identify what kind of answer the problem asks for. If writing a procedure, carefully look at requirements for input and output, and any restrictions on how it must be written or which other procedures may be used.

You may assume a list argument will be flat unless it is otherwise specified, and that the elements will not produce errors for the procedures described.

Do not do unnecessary testing. For example, testing for both list? and null? instead of using one test and then else would be considered unnecessary testing.

Do not make code unnecessarily inefficient. An extra procedure call here or there is ok, but do not make an $O(n)$ problem into $O(n^2)$.

Do problems you are confident about first. If you finish the problems you know, write what you do know about other problems to gain partial credit; but erroneous information may detract from that credit, so don't make stuff up.

You may use any of the Scheme primitives we use in class.

1. (26 pts) Show what is returned by the interpreter in response to each of these expressions. Each is to be evaluated by itself. If the result is an error, or other interpreter message, you do not have to be exact. For example, if the expression were the input "expt", you could show the interpreter's response as "primitive" instead of writing #<primitive:expt>. If you write more than one thing after a question **CIRCLE** the one you want graded.

   (a) `(cons 3 4)`

   (b) `(cons 8 ())`

   (c) `(list 8 ())`

   (d) `(append () ())`

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

   (f) `(list (list 1 2)(list 3 4))`

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

   (h) `(cdr (cons 3 ()))`

   (i) `(if (eq? (list 1) (list 1)) 2 3)`

   (j) `(quote a)`

   (k) `(cons () '(1))`

   (l) `((lambda (x) (+ x x)))`

   (m) `((lambda (x) (* x x 3) 4) 5)`

2. (7 pts) Draw the box and pointer notation for (2 ((3 (4)) 5) 6)

3. (7 pts) Write the interpreter's representation of this structure:

4. (10 pts)Write the simple recursive-process procedure product-list, which takes two non-nested lists of integers, and returns the list of integers resulting from multiplying each element of blist by the corresponding element in alist. If one list is longer, the remaining elements are multiplied by 1.

```
> (product-list '(4 5 6) '(1 2 7))
(4 10 42)
> (product-list '(1 2 3) '(1 2 3 6))
(1 4 9 6)
>
```

5. (5 pts) Write the simple recursive procedure **scale** that multiplies each member of a list by some factor. Do not use the sequence toolkit.

```
> (scale 2 (list 1 2 3))
(2 4 6)
```

6. (5 pts) Write **scale** using your sequence toolkit.

7. (5 pts) Write a **call** to your sequence toolkit that generates the list of the cubes of the even numbers from 1 to 1000. You may use lambdas, but do not separately define procedures.

8. (5 pts) Write a **call** to your sequence toolkit that generates the matrix shown. You may use lambdas, but do not separately define procedures.

```
((0 1 2 3 4) (5 6 7 8 9) (10 11 12 13 14) (15 16 17 18 19))
```

Complete the following procedure definitions.

9. (5 pts)

```
(define (fast-expt x y)
  (cond ((= y 0) 1)

        ((odd? y) _____________________________________

        _____________________________________________
```

10. (5 pts)

```
;You can put this on more than one line if you wish.
(define (filter proc alist)

  (accumulate _____________________________________________________________
```

11. (5 pts)

```
(define (equals? a b)
  (cond ((pair? a)




        (else (eq? a b))))
```

12. (5 pts)

```
(define (append a b)
  (define (iter x y)
    (cond ((null? x) y)

          (else _____________________________________________________________
  (iter (reverse a) b))
```

13. (12 pts) Write sort as an accumulation. You may write a helper procedure if needed.

14. (5 pts) What is big O for time for the sorting procedure in 13? Explain.

15. (5 pts) What is big O for time of f1? Explain. (Remember that you aren't as concerned with the result of this call as you are with the behavior of the process.)

```
(define (f1 alist)
  (accumulate (lambda (a b) (+ (length alist) b))
              0
              alist))
```

16. (5 pts) What is big O for time of f2? Explain.

```
(define (f2 alist)
  (accumulate (lambda (x y) (cons (fast-expt 2 x) y))
              ()
              alist))
```

17. (5 pts) Consider the following two procedures. Xenon thinks that one of these should be faster than the other, but isn't sure which one. Help Xenon decide, and briefly explain your answer.

```
(define (one alist)
  (let ((x (length alist)))
    (* x x)))
```

```
(define (two alist)
    (* (length alist)(length alist)))
```

18. (5 pts) Define a function **make-multiplier** that takes a single factor as a parameter and returns a procedure that multiplies its argument by that factor:

```
> ((make-multiplier 3) 3)
9
>
```

(a) (3 pts) Now make a single call to a sequence tool that will use make-multiplier, and define **x** to be a list of multipliers that use the factors in the list `(2 3 4 5)`. After the call x will be a list of procedures.

```
(define x
```

(b) (3 pts) Now use the list of multipliers, x, and a single call to a sequence tool to make a list of the results of calling each multiplier on the number 3, so the result will be `(6 9 12 15)`. You will need a lambda function.

```
> _____________________________________________________
```

19. (5 pts)In class we defined a procedure cons. Fill in the three blanks:

```
(define cons

  _____________________________
   (lambda (msg)
     (cond ((equal? msg 'car) _______)

           ((equal? msg 'cdr) _______)
           (else 'error)))))
```

20. (3 pts) Extra Credit: The Joel Spolsky article refers to a process that uses two sequence operators, map and accumulate (but he uses another name for accumulate). What was the name of the process and what company uses it?