CISC280 partial Midterm 1, Fall 2005

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 seven pages worth a total of 130 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. (20 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 () ())`

   (c) `(cdr (cons () ()))`

   (d) `((if (< 3 3) + -) 2 3)`

   (e) `'(a b c)`

   (f) `(cons 3 ())`

   (g) `(car ''(a b c))`

   (h) `(cons () (list 7 8))`

   (i) `((lambda (x) (+ x x)) (+ 2 2))`

   (j) `((lambda (x) (+ x x)(* x x)) 4)`

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

3. (8 pts) Write the simple recursive procedure get-MOS that returns a list of the multiples of seven found in an argument list.

```
> (get-MOS (list 1 7 4 29 14 0))
(7 14 0)
```

4. (10 pts) Abstract away from the specifics of this procedure to write a more general procedure for extracting items (including non-numeric items) from a list, as done in class.

5. (7 pts) Show a single call to the general procedure that produces the same result as a call to get-MOS on a given list. You may not use other named procedures, but you will use a lambda expression.

>_____

Complete the following procedure definitions.

6. (5 pts)

```
(define reverse-iter (lambda (alist blist)
  (cond ((null? alist) blist)
        (else
```

7. (5 pts)

```
(define append
  (lambda (alist blist)
    (cond ((null? alist) blist)
          (else
```

8. (5 pts)

```
(define (deep-reverse alist)
  (cond ((null? alist) ())
        ((list? (car alist))
```

```
        (else (append (deep-reverse (cdr alist))
                      (list (car alist))))))
```

9. (5 pts)

```
(define (length alist)
   (accumulate
```

10. (5 pts)

```
;> (count-numbers '(1 7)) returns 2
;> (count-numbers '(() 1 3 () (((4))((8))))) returns 4
;> (count-numbers ()) returns 0
(define (count-numbers alist)
   (accumulate + 0
               (map
```

Consider the fascinating function D.

```
D(n) = 1 if n > 204;
       D(n+1) + D(n + 2) otherwise.
```

11. (5 pts) Write a simple recursive procedure **dproc** to calculate D.

Notice that D recurses to an upper bound, not a lower bound. According to Terry, this makes **dproc** tractable (feasibly computable) for all n, whereas simple recursive Fibonacci is intractable for large values of n.

12. (6 pts) How does **dproc** respond to large values of n? Small values? Negative values?

13. (6 pts) Is Terry correct? Explain, being *as specific as possible*.

Consider the behavior of this procedure. Hint: look at the successive values of x and y both as a call is made and just after the args pass through the dotted-tail notation.

```
(define test (lambda (x . y)
               (cond ((null? y) x)
                     (else (test x (cdr y))))))
```

14. (6 pts) Suppose we call `(test 3 4)`. Show the *next* three recursive calls to test (or write "end" if the process terminates before then.) When you show a recursive call, show the args being passed, not an expression like `(cdr y)`. (Hint: this was the hardest question on this exam.)

(a) _____

(b) _____

(c) _____

15. (3 pts) Extra Credit: Paul Graham said the Viaweb editor was composed of 20 - 25 percent what?