Fall, 1992 Name: _____

Grading TA's name: _____ Section Letter : _____ Section Time : _____

Com S 227X — Introduction to Computer Programming

Test on Chapters 5 and 6 (Version 1)

This test has 7 questions and pages numbered 1 through 5.

Special Directions for This Test

This test is closed book and closed notes. *No calculators allowed.*

If you need more space, use the back of a page. Note when you do that on the front.

This test is timed. We will not grade your test if you try to take more than the time allowed. Therefore, before you begin, please take a moment to look over the entire test so that you can budget your time.

Indentation is important to us for "clarity" points; if your code is sloppy, you will lose points.

The "evidence" (argument) that your code is correct described in the *Red Tape* is not needed for this test.

Subset of Scheme You May Use to Define Other Procedures

Unless otherwise stated, when defining procedures you may only use: helping procedures that you define yourself, and the procedures and keywords that are included in the following list. (That is to say, you may *not* use Scheme procedures and keywords that are absent from the following list.) The notation `c...r` means `caar`, `cadr`, `cddr`, `cdar`, `caaar`, etc.

```
#t, #f, *, +, -, /, <, <=, =, >=, >,
abs, acos, add1, and, append, asin, atan, atom?, begin, boolean?,
car, cdr,  c...r, ceiling, cond, cons, cos, define, display,
else, eq?, equal?, eqv?, error, exp, expt, floor, if, integer?,
lambda, length, let, letrec, list, list-ref, log, max, min, modulo,
negative?, newline, not, null?, number?, or, pair?, positive?, procedure?,
quote, quotient, read, real?, remainder, reverse, round,
sin, sqrt,  string=?, string-append, string-ci=?, string-length, substring,
sub1, symbol?, symbol->string, tan, truncate, write, writeln, zero?
```

Directions From the Course "Red Tape"

You are encouraged to use auxiliary (helping) functions whenever they are helpful.

For programming problems, there is usually more than one right answer, hence it is important to write a clear solution. Answers to programming questions will be scored with the following breakdown of points:

- 60% for elegance and clarity (using relevant techniques to modularize the solution, making good use of language features, the creativity of your solution)
- 40% for correctness (whether your program solves the given problem correctly in the given subset of the language)

You should try to eliminate all syntax errors from your programs, as they affect both the correctness and clarity of your solution; you may receive no points for programs with major syntax errors. Checking of external inputs, and other conventions for dealing with humans are not important (for this class), unless the problem says so.

1. (5 points) Convert the following base 10 number to binary: 35.

2. (5 points) Convert the following binary number to base 10: 10110001

3. (5 points) Write an equivalent (nested) `lambda` expression, for the following expression. Your answer should not use `let`. using any `let` expressions.

```
(let ((this (+ a 5))
      (that (- b a)))
  (let ((another (+ this (* that c))))
    (if (zero? this) that another)))
```

Assume that `a`, `b`, and `c` are defined elsewhere.

4. (8 points) Give the output of the following expression. If it encounters an error, write "this is an error" and describe the error, giving any output that happens before the error. If it goes into an infinite loop, write "this goes into an infinite loop," again giving any output that happens (show infinite output with ...). Assume that `writeln` has already been defined.

```
(let ((x 3)
      (y 4))
  (let ((x (+ y 2))
        (y x))
    (letrec ((s (lambda (n)
                  (if (zero? n)
                      0
                      (+ n (s (sub1 n)))))))
      (writeln "y is: " y)
      (writeln "the result of (s y) is: " (s y)))))
```

5. (8 points) Give the output of the following expression. If it encounters an error, write "this is an error" and describe the error, giving any output that happens before the error. If it goes into an infinite loop, write "this goes into an infinite loop," again giving any output that happens (show infinite output with ...). Assume that `writeln` has already been defined.

```
(letrec
    ((du
      (lambda (n)
        (cond
         ((zero? n) (newline) (display 0) (newline))
         (else (show-n n) (du (sub1 n)) (show-n n)))))
     (show-n
      (lambda (n)
        (display "/") (display n) (display "_"))))
  (du 4))
```

6. (12 points) This is a problem about polynomials. Recall that the 5 basic definitions for polynomials are: `the-zero-poly`, `degree`, `leading-coef`, `rest-of-poly`, and `poly-cons`.

   Using the 5 basic definitions and any helping procedures you define yourself, define a procedure `increase-degrees` that takes a polynominal and increases the degree of each of its terms by 1. For example, passing the polynomial

   $$2x^3 + 9x + 7.2$$

   to `increase-degrees` results in the following polynomial.

   $$2x^4 + 9x^2 + 7.2x$$

   Passing the zero polynomial to `increase-degrees` results in the zero polynomial, since $0x = 0$.

   Low level solutions will lose points. (Don't ask us what the 5 basic definitions for polynomials do, you're supposed to know them.) If you want to use other procedures on polynomials besides the 5 basic definitions, you will have to write their definitions here.

7. (12 points) Define an iterative procedure, `list-of-initial-substrings`, that takes a string, `strng`, and returns a list of the initial substrings of `strng`. The term "initial substring of `strng`" means a substring that, if non-empty, contains the first character of `strng`. The initial substrings are to be ordered in the result list by length, from shortest to longest. For example,

```
(list-of-initial-substrings "ant") ==> ("" "a" "an" "ant")
(list-of-initial-substrings "") ==> ("")
(list-of-initial-substrings "apple") ==> ("" "a" "ap" "app" "appl" "apple")
```

   Note that there is only one (1) argument passed to the procedure.

   You must use `let` or `letrec` to define any helping procedures.

   Only solutions that are iterative and that have all helping procedures defined with `let` or `letrec` will get points. (Don't ask us what "iterative" means, or how to define helping procedures with `let` or `letrec`, you're supposed to know that.)

   You will lose points if you recompute values that do not need to be recomputed, or if you pass more arguments to a procedure than would be necessary (if you had made appropriate arrangements).