Fall, 1993 Name: _____

Grading TA's name: _____ Section Day and Time : _____

Com S 227 — Introduction to Computer Programming

Test on Chapters 5 and 6 (Version 1)

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

Reminders

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.

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. You may only use `define` at the top level.

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

Types for Scheme String Procedures

Recall that the Scheme procedures for strings have the following types.

```
string=? : (-> (string string) boolean)
string-append : (-> (string ...) string)
string-ci=? : (-> (string string) boolean)
string-length : (-> (string) natural)
substring : (-> (string natural natural) string)
```

1. (5 points) Convert the following decimal number to binary (by hand): 36

2. (10 points) Consider the following expression.

```
(let ((x   1)     (y   2))

  (let ((x   (+ y 3))     (y   (+ y 10)))

    (writeln "x = " x " and y = " y)))
```

   (a) In the expression, draw a squiggly arrow ($\rightsquigarrow$) going from each formal parameter to the expression to which it is bound. (Don't ask us what a "formal parameter" is, you're supposed to know that.)

   (b) Draw a straight arrow ($\rightarrow$) from each use of a variable in the expression to the formal parameter to which it refers.

   (c) Give the output of the expression.

   You may assume that `writeln` has already been defined.

3. Consider the following procedure definitions.

```
(define print-to-n
  (lambda (n)
    (writeln "a:")
    (print-to-n-it n 1)
    (newline)))

(define print-to-n-it
  (lambda (n m)
    (if (< m n)
        (begin
          (display m)
          (display ",_")
          (print-to-n-it n (add1 m))))))
```

(a) (5 points) Given the above definitions, show what is printed of the following expression.

```
(print-to-n 5)
```

Show the output as precisely as possible, paying particular attention what is on the same line, and what is on different lines.

(b) (10 points) Rewrite the definition of `print-to-n`, so that: (i) `print-to-n-it` is not defined as a separate procedure, but is defined using a `letrec`, and (ii) you make any other simplifications this rewriting using `letrec` allows. (Your code should behave the same as the code above.)

4. (20 points) This is a problem about polynomials. Recall that the basic procedures for the abstract data type poly have the following types:

```
the-zero-poly : poly
degree :       (-> (poly) integer)
leading-coef : (-> (poly) real)
rest-of-poly : (-> (poly) poly)
poly-cons :    (-> (natural real poly) poly)
```

(Don't ask us what these procedures do, you're supposed to know that.)

Using the basic procedures and any helping procedures you define yourself, define a procedure `poly-equal?` that takes two polys, `p1` and `p2`, and returns `#t` if and only if `p1` and `p2` are equal. You may *not* use `equal?` to compare polys in your solution; instead you must use the basic procedures listed above.

```
(poly-equal? the-zero-poly the-zero-poly) ==> #t
(poly-equal? (poly-cons 5 1.7 the-zero-poly) (poly-cons 5 1.7 the-zero-poly)) ==> #t
(poly-equal? (poly-cons 5 99.8671 the-zero-poly) (poly-cons 5 1.7 the-zero-poly)) ==> #f
(poly-equal? (poly-cons 72 1.7 the-zero-poly) (poly-cons 5 1.7 the-zero-poly)) ==> #f
(poly-equal?
 (poly-cons 227 1.7 (poly-cons 42 9.3 (poly-cons 0 4.1 the-zero-poly)))
 (poly-cons 227 1.7 (poly-cons 42 9.3 (poly-cons 0 4.1 the-zero-poly)))) ==> #t
```

If you want to use other procedures on polynomials besides the basic procedures listed above, you will have to write their definitions here (that is part of this problem). Look over the list of basic procedures above carefully!

Low level solutions will lose points. Don't use `equal?` to compare polys.

5. (20 points) Define an interactive procedure, `prefixes`, that takes no arguments and has the following behavior.

   The procedure `prefixes` prompts for a string, and then prints all prefixes of that string. A "prefix" is a substring that starts at the beginning of the string. The procedure continues prompting and printing in this way until the empty string is entered, and then it prints "`Goodbye.`" instead of continuing.

   You no *not* have to check that the input is a string. For example, consider the following transcript.

```
> (prefixes)
What string? "abcde"
"abcde"
"abcd"
"abc"
"ab"
"a"
""
What string? "bee"
"bee"
"be"
"b"
""
What string? ""
Goodbye.
>
```