Spring, 1993 Name: _____

Grading TA's name: _____ Section Name : _____

Com S 227X — Introduction to Computer Programming

Test on Chapter 7 (Version 1)

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

Reminders

This test is closed book and closed notes, except that you are allowed to use your copy of the handout *Make Your own Scheme Reference Card*. On this you may fill in stuff in the blanks, but you may not write out example procedures.

As on homeworks, for sufficiently complex helping procedures, you must write TYPE, ENSURES or EFFECT, (and REQUIRES if appropriate) comments.

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.

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, apply, asin, atan, atom?, atomic-item?, begin, boolean?,
car, cdr,  c...r, ceiling, cond, cons, cos, deep-recur-abs, define, display
else, eq?, equal?, eqv?, error, exp, expt, flat-recur, floor, for-each, if, integer?,
lambda, length, let, letrec, list, list-ref, log, map, 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 "Course Policies and Procedures"

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. (8 points) What is printed (as a side effect) by the following expression? Be sure to write all the output.

```
(let ((me  (lambda (proc) (proc #t)))
      (you (lambda (proc) (proc #f)))
      (us  (lambda (x y)
             (lambda (b)
               (if b x y)))))
    (writeln "a: " (me (us 1 2)))
    (writeln "b: " (you (us 1 2)))
    (writeln "c: " (me (you (us 1 (us 2 3))))))
```

2. (10 points) Define a procedure `curry` that takes a two-argument procedure, `proc`, and returns a curried version of that procedure. (Don't ask us what a curried procedure is, you're supposed to know that.) For example,

```
(((curry +) 3) 4) ==> 7
(((curry *) 5) 6) ==> 30
(((curry cons) 'a) '()) ==> (a)
```

3. (10 points) Define a procedure, `sum-each`, that takes a list of lists of numbers, and returns a list of the sums of each inner list. For example,

```
(sum-each '()) ==> ()
(sum-each '(() () (92))) ==> (0 0 92)
(sum-each '((1 2) (0 0 7) (-10 10 20))) ==> (3 7 20)
(sum-each '((1 2 3 4) (5) (6 7) (8 9))) ==> (10 5 13 17)
```

4. (10 points) What is printed (as a side effect) of the following expression? Be sure to write all the output.

```
(letrec ((from-to ; TYPE: number, number -> ((number -> void) -> void)
           (lambda (from to)
             (lambda (step)
               (letrec ((loop ; TYPE: number -> void
                          (lambda (i)
                            (if (<= i to)
                                (begin
                                  (step i)
                                  (loop (add1 i)))))))
                 (loop from))))))
  ((from-to 3 7) (lambda (i) (begin (display i) (display ", "))))
  (newline)
  ((from-to 7 3) (lambda (i) (begin (display i) (display ", "))))
  (newline)
  (writeln 'done))
```

5. (20 points) Using `deep-recur-abs`, define a procedure `tree-map-maker` that takes a predicate `T?`, and returns a procedure that takes a procedure, `proc`, and a tree, `tr`, and returns a tree with the same shape as `tr` in which each item (of type `T`) in `tr` is replaced by the result of applying `proc` to that item. That is, the type of `tree-map-maker` is

   ```
   all S. all T . (datum -> boolean) -> ((T -> S), (tree of T) -> (tree of S))
   ```

   You may assume that `(T?` x`)` is `#t` if and only if x has a type `T` that is suitable as an argument for `proc`. For example,

   ```
   ((tree-map-maker number?) add1 '(4 (3 () 5 (7)) 2)) ==> (5 (4 () 6 (8)) 3)
   ((tree-map-maker atomic-item?) number? '((3) () (4 asymbol) ((9))))
                                       ==> ((#t) () (#t #f) ((#t)))
   ((tree-map-maker number?) (lambda (x) (+ 7 (/ x 2))) '()) ==> ()
   ```