Fall, 1992 Name: _____

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

Com S 227X — Introduction to Computer Programming

Test on Chapters 7, 9, and 11 (Version 1)

This test has 9 questions and pages numbered 1 through 10.

Special Directions for This Test

You should think about what will make your solution clear and elegant before starting to write code. We will take points off for inelegant and unclear code.

For the procedures we explicitly ask you to write, you don't have to write `TYPE`, `REQUIRES`, `ENSURES`, `MODIFIES`, or `EFFECT` comments. (However, if you wish to, you may). On the other hand, for helping procedures you should write such comments.

This test is closed book and closed notes.

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.

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, you may *not* use Scheme procedures and keywords that are absent from the following list, unless you define them yourself.) The notation `c...r` means `caar`, `cadr`, `cddr`, `cdar`, `caaar`, etc.

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

Directions From the Course "Red Tape"

The remarks about correctness arguments have been deleted.

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) What is the value of the following expression?

```
(let ((x 'vanilla)
      (y 'strawberry))
  (let ((f (lambda (z) (list z x))))
    (let ((x 'chocolate))
      (set! x 'peach)
      (set! y 'cherry)
      (f y))))
```

2. (5 points) Draw a box and pointer diagram for the result of the following expression. (Don't ask us what a box and pointer diagram is, you're supposed to know that.)

```
(let ((a (cons 'what '()))
      (b (cons 'why '()))
      (c (cons 'how '())))
  (set-car! b c)
  (set-car! a b)
  a)
```

3. (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))))))
```

4. (8 points) What is printed (as a side effect) by the following expression?

```
(let ((foobar (lambda (x)
                (lambda args
                  (let ((m (car args)))
                    (cond
                     ((eq? m 'c) (set! x (cadr args)))
                     ((eq? m 'v) x)
                     (else (error "I don't understand"))))))))
  (let ((this (foobar 3)))
    (writeln "a: " (this 'v))
    (this 'c 6)
    (writeln "b: " (this 'v))))
```

5. (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)
```

6. (12 points) Define a procedure `tree-map` that takes a procedure, `proc`, and a tree, `tr`, and returns a tree in which each atomic item in `tr` is replaced by the result of applying `proc` to that item. The original tree is not to be mutated. Your code should work for any single argument procedure, `proc`. For example,

```
(tree-map add1 '(4 (3 () 5 (7)) 2)) ==> (5 (4 () 6 (8)) 3)
(tree-map number? '((3) () (4 asymbol) ((9)))) ==> ((#t) () (#t #f) ((#t)))
(tree-map procedure? '()) ==> ()
```

In your solution you may use the following procedure.

```
(define atomic-item? ; TYPE: datum -> boolean
  (lambda (x)        ; ENSURES: result is true iff x is a leaf of a tree
    (not (or (pair? x) (null? x)))))
```

In your solution you are to use one of the following procedures: `deep-recur` (from the text) or `deep-recur-abs` (described in lecture). Recall that they have types:

```
deep-recur: b, (a, b -> b), (b, b -> b) -> (tree of a -> b)
deep-recur-abs: b, (a, b -> b), (b, b -> b), (datum -> boolean) -> (tree of a -> b)
```

7. (12 points) This is a problem about vectors. Recall that the basic procedures for vectors are as follows.

```
make-vector: integer, a -> vector of a
vector-generator: (integer -> a) -> (integer -> vector of a)
vector?: datum -> boolean
vector-length: vector -> integer
vector-ref: vector of a, integer -> a
vector-set!: vector of a, integer, a -> void
```

Define a procedure, `proc-vector`, that takes an arbitrary number of procedures as arguments and returns a procedure that, when given a vector of the same length as the number of procedures, returns a vector whose ith element is the result of applying the ith procedure to the ith element of the vector. You may assume that the length of the vector argument is the same as the number of procedure arguments. The original vector is not to be mutated. For example,

```
((proc-vector add1 sub1) '#(3 2)) ==> #(4 1)
((proc-vector (lambda (x) (+ x 6))) '#(3)) ==> #(9)
((proc-vector) '#()) ==> #()
```

8. (15 points) This is a problem about matrices. In your solution you can use the basic procedures for matrices, which have the following types:

```
matrix-generator: (integer, integer -> a) -> (integer, integer -> matrix of a)
matrix-ref: matrix of a -> (integer, integer -> a)
num-rows: matrix -> integer
num-cols: matrix -> integer
matrix-set!: matrix of a -> (integer, integer, a -> void)
```

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

In linear algebra, the there are three basic operations that one applies to a matrix A that preserve its mathematical properties. One of these elementary row operations is to interchange two rows of a matrix.

Write a procedure, `interchange-rows`, that takes a matrix, `mat`, as an argument, and returns a procedure that takes two integers, `i1` and `i2`, and returns a new matrix, call it A such that, the (i, j)th element of A, $A_{i,j}$, is

$$A_{i,j} = \begin{cases} \texttt{mat}_{i,j} & \text{if } i \neq \texttt{i1} \text{ and } i \neq \texttt{i2} \\ \texttt{mat}_{\texttt{i2},j} & \text{if } i = \texttt{i1} \\ \texttt{mat}_{\texttt{i1},j} & \text{if } i = \texttt{i2}. \end{cases}$$

For example, if `my-mat` is

$$\begin{pmatrix} 9 & 8.77 & 9.2 & 4.5 \\ 10 & 6 & 5.2 & 2.73 \\ 0 & 1.1 & 2.3 & 4.4 \end{pmatrix},$$

then the call `((interchange-rows my-mat) 2 0)` returns a new matrix:

$$\begin{pmatrix} 0 & 1.1 & 2.3 & 4.4 \\ 10 & 6 & 5.2 & 2.73 \\ 9 & 8.77 & 9.2 & 4.5 \end{pmatrix}.$$

The call `((interchange-rows my-mat) 0 2)` returns another new matrix with the same elements as the result of the first call. The call `((interchange-rows my-mat) 1 1)` returns a new matrix that has the same elements as `my-mat`.

Your program should check that `i1` and `i2` are legal row indexes for the matrix, and if they are not, it should give the error "`illegal row index`".

You should *not* assume anything about the representation of matrices; low level solutions will lose points. You should not mutate the argument to `interchange-rows`. You may use the following (blank) page to write your answer.

9. (15 points) In Scheme (and LISP) one can use lists and nested lists to represent expressions. For example the expression `(car (cons 3 x))` can be thought of as a tree, which might be formed as follows: `(list 'car (list 'cons 3 x))`. Even more simply, we can form this tree by writing: `'(car (cons 3 x))`.

   In this problem you will write a procedure, `let->lambda`, that takes an expression, represented as a tree, and transform the expression to an equivalent one that does not use `let`. The procedure `let->lambda` is not supposed to evaluate the expression, just to produce another, equivalent, expression. (Don't make this problem harder than it is.) The expressions passed to `let->lambda` may include `let`, `lambda`, and applications. They don't include any other special forms, such as `'` or `quote`. So for this problem the argument expressions are represented by trees (of numbers and symbols) having one of the following forms:

   - the empty list `()`,
   - a list whose `car` is the symbol `let`, representing a `let` expression,
   - a list whose `car` is the symbol `lambda`, representing a `lambda` expression, (of the usual kind, not unrestricted lambda),
   - a list whose `car` is neither `let` nor `lambda`, which should be assumed to be an application.

   Note that you must replace each `let` expression by an equivalent application of a `lambda` expression, even those nested in subexpressions of an application, the values being bound inside a let, and in the body of a `let` or `lambda`. For example,

```
(let->lambda '(+ 3 4))                    ==> (+ 3 4)
(let->lambda '(let ((x (+ 3 4))) (f x))) ==> ((lambda (x) (f x)) (+ 3 4))
(let->lambda '(+ 7 (let ((x (let ((z 3)) (g z z)))
                         (y (* 3 2)))
                     (* x y))))           ==> (+ 7 ((lambda (x y) (* x y))
                                                    ((lambda (z) (g z z)) 3)
                                                    (* 3 2)))
(let->lambda '(let ((a 5))
                (let ((timesa (lambda (x) (let ((y 2)) (+ y (* x a)))))
                      (a 0.5))
                  (- b (timesa a)))))     ==> ((lambda (a)
                                                 ((lambda (timesa a)
                                                    (- b (timesa a)))
                                                  (lambda (x)
                                                    ((lambda (y) (+ y (* x a))) 2))
                                                  0.5))
                                               5)
```

   Although `let` must not appear in the result, in your code you can use `let`. You may assume that the expression has no syntax errors. You may use the following (blank) page to write your answer.

   Hints: how can you get at the parts of a list representing a `let` expression? How would you reassemble those parts into a list representing an equivalent `lambda`? What kind of recursion is this?