Spring, 1993 Name: _____

Grading TA's name: _____ Section Name : _____

Com S 227X — Introduction to Computer Programming

Test on Chapters 9 and 11 (Version 1)

This test has 7 questions and pages numbered 1 through 6.

Reminders

This test is closed book and closed notes.

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.

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, 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!, while-proc, write, writeln, zero?
```

You may also use `while` as a synonym for `while-proc`.

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. (5 points) Fill in the blanks in the following transcript.

```
> (define va (vector 1 2 3))
> (define vb va)
> (vector-set! vb 1 99)
> va
__________________________
> vb
__________________________
> (set! va (vector 22 33 44))
> va
__________________________
> vb
__________________________
> (eq? (cons 3 '()) (cons 3 '()))
__________________________
```

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-cdr! b c)
  (set-cdr! a b)
  (set-car! a b)
  a)
```

3. (5 points) What is the value of the following expression?

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

4. (10 points) This is a problem about vectors. Recall that the types of the basic procedures for vectors are as follows.

```
make-vector: all T . natural, T -> (vector of T)
         and all T . natural -> (vector of T)
vector: all T . T ... -> (vector of T)
vector?: datum -> boolean
vector-generator: all T . (natural -> T) -> (natural -> (vector of T))
vector-length: all T . (vector of T) -> natural
vector-ref: all T .(vector of T), natural -> T
vector-set!: all T . (vector of T), natural, T -> void
```

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

Write a procedure `vector-xerox` that takes a vector `vec` and a non-negative integer `n` and returns a new vector that has `n` copies of each element of `vec`. You should not mutate the argument `vec`. For example,

```
(vector-xerox '#(x y z) 4) ==> #(x x x x y y y y z z z z)
(vector-xerox '#(x y z) 1) ==> #(x y z)
(vector-xerox (vector 2 4 6) 0) ==> #()
```

5. (20 points) Consider a long street with many street lights. Each street light can be either red, yellow, or green.

   In a program, the lights along the street are represented by a vector of symbols, where the symbols are all either `red`, `yellow`, or `green`. You are to write a procedure, `update-lights!` that takes the vector of lights as an argument, and changes each of the lights in the vector according to the pattern:

```
red     changes to  green
green   changes to  yellow
yellow  changes to  red
```

   Note that you are to mutate the argument, and not return it. (Don't ask us what "mutate" means, you're supposed to know that.) For example, consider the following transcript.

```
> (define signals '#(red red yellow green red))
> (update-lights! signals)
> signals
#(green green red yellow green)
> (update-lights! signals)
> signals
#(yellow yellow green red yellow)
```

6. (20 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: all T . (natural, natural -> T) -> (natural, natural -> matrix of T)
matrix-ref: all T . (matrix of T) -> (natural, natural -> T)
num-rows: all T . (matrix of T) -> natural
num-cols: all T . (matrix of T) -> natural
matrix-set!: all T . (matrix of T) -> (natural, natural, T -> void)
```

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

Write a procedure `matrix-add-scalar` that takes a matrix `m` and a number `c` and returns a new matrix (of the same size and shape as `m`) such that the (i,j)-th element of the result is `c` plus the (i,j)-th element of `m`. You should not mutate the argument matrix. For example, using the `matrix-view` procedure you did in your homework, consider the following transcript.

```
> (matrix-view A)
1 2 3
4 5 6
> (define B (matrix-add-scalar A 10.0 ))
> (matrix-view B)
11.0 12.0 13.0
14.0 15.0 16.0
> (matrix-view A)
1 2 3
4 5 6
> (define C (matrix-add-scalar B 200))
> (matrix-view C)
211.0 212.0 213.0
214.0 215.0 216.0
```

7. (10 points) Recall that `while-proc` (which you can also call `while`) and `make-vector` have the following types

```
while-proc: ( -> boolean), ( -> void) -> void
make-vector: all T . natural -> (vector of T)
```

For this problem you cannot use `make-vector` with 2 arguments. You are supposed to know what these do.

Using `while-proc`, and `make-vector` write a procedure `duplicate`, that takes a string `s` and a natural number `n`, and returns a new vector with `n` copies of `s` in it. Do *not* use `vector-generator`. **Solutions that do not use `while-proc` (or `while`) and `make-vector` will receive zero (0) points.**

For example,

```
(duplicate "C++ is really super" 0) ==> #()
(duplicate "never write the same thing twice" 2)
  ==> #("never write the same thing twice" "never write the same thing twice")
(duplicate "lambda" 4) ==> #("lambda" "lambda" "lambda" "lambda")
```