Fall, 1993 Name: _____

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

Com S 227 — Introduction to Computer Programming

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

This test has 8 questions and pages numbered 1 through 9.

Reminders

We will take off some points for inelegant solutions. but don't let that stop you from writing down a solution that works; write that first, and then write something more elegant if you have time.

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.

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?
```

Types of Built-in Procedures from Chapters 7, 9, and 11

Name	: Type
`apply`	: (→ ((→ (datum ...) T) (list datum)) T)
`deep-recur-abs`	: (→ (T (→ (S T) T) (→ (T T) T) (→ (datum) boolean)) (→ ((tree S)) T))
`flat-recur`	: (→ (T (→ (S T) T)) (→ ((list S)) T))
`for-each`	: (→ ((→ (S) void) (list S)) void)
`map`	: (→ ((→ (S) T) (list S)) (list T))
`make-vector`	: (and (→ (natural) (vector datum)) (→ (natural T) (vector T)))
`list->vector`	: (→ ((list T)) (vector T))
`vector`	: (→ (T ...) (vector T))
`vector?`	: (→ (datum) boolean)
`vector-generator`	: (→ ((→ (natural) T)) (→ (natural) (vector T)))
`vector-length`	: (→ ((vector T)) natural)
`vector-ref`	: (→ ((vector T) natural) T)
`vector-set!`	: (→ ((vector T) natural T) void)
`while-proc`	: (→ ((→ () boolean) (→ () void)) void)

1. Use `map` to solve the following problems.

   (a) (5 points) Define a procedure `lengths` that takes a list of vectors, and returns a new list of their lengths. For example,

   ```
   (lengths '()) ==> ()
   (lengths '(#() #(1 2 3 4) #(1 2 3 4 5 6) #(1 2 3 4 5 6 7))) ==> (0 4 6 7)
   (lengths '(#(n o w) #(w h a t) #(a) #(f i n e) #(m e s s) #(t h i s) #(i s)))
            ==> (3 4 1 4 4 4 2)
   ```

   You must use `map` *in your solution.*

   (b) (7 points) Define a procedure `cube-plus-self` that takes a list of numbers, `ls`, and returns a new list, such that each number x in `ls` is replaced by $x^3 + x$ in the result. For example,

   ```
   (cube-plus-self '()) ==> ()
   (cube-plus-self '(1 2 3)) ==> (2 10 30)
   (cube-plus-self '(10 20 1 5 7 9)) ==> (1010 8020 2 130 350 738)
   ```

   You must use `map` *in your solution.*

2. Use the procedure `deep-recur-abs` to solve the following problems.

   (a) (8 points) Define a procedure `tree-sum-of-squares` that takes a tree of numbers and returns the sum of the squares of the numbers in the tree.

```
(tree-sum-of-squares '()) ==> 0
(tree-sum-of-squares '(() ())) ==> 0
(tree-sum-of-squares '((((3)) () (4)))) ==> 25
(tree-sum-of-squares '((1 (2)) () (3 () ((4) 5)) (6 () (((7 8))) 9))) ==> 285
```

   You must use `deep-recur-abs` *in your solution.*

   (b) (10 points) Define a procedure `tree-map` that takes a procedure, `proc`, and tree of atomic-items, and returns a new tree which is like the original but has each atomic-item replaced by the result of calling `proc` and passing it that item. For example,

```
(tree-map add1 '()) ==> ()
(tree-map add1 '((1 (2)) () (3 () ((4) 5)) (6 () (((7 8))) 9)))
            ==> ((2 (3)) () (4 () ((5) 6)) (7 () (((8 9))) 10))
(tree-map vector '((a good time) ((was)) (had (by all))))
            ==> ((#(a) #(good) #(time)) ((#(was))) (#(had) (#(by) #(all))))
(tree-map (lambda (v) (vector-ref v 0)) '(#(a b c) () (((#(do re me))))))
           ==> (a () (((do))))
```

   You must use `deep-recur-abs` *in your solution.*

3. (10 points) Define a procedure `curry-1st` that takes a procedure, `f`, of type

$$(\rightarrow \text{(S T ...) U})$$

and returns a procedure of type

$$(\rightarrow \text{(S) } (\rightarrow \text{(T ...) U}))$$

which is like `f` but curried in its first argument. That is, for all f, x, a, b, c, ..., `curry-1st` satisfies

$$\texttt{(((curry-1st } f\texttt{) } x\texttt{) } a\ b\ c\ \ldots\texttt{)} = \texttt{(}f\ x\ a\ b\ c\ \ldots\texttt{)}$$

For example, consider the following transcript.

```
> (((curry-1st *) 10) 1 20 -2)
-400
> (((curry-1st *) 10) 1 20 -2 -1 10 1 2)
8000
> (define add2 ((curry-1st +) 2))
> (add2 3 4 6)
15
> (add2 3 4 6 10)
25
> (define messenger ((curry-1st writeln) "Message: "))
> (messenger "hello earthlings!")
Message: hello earthlings!
> (messenger "This is the planet " 'zork " speaking.")
Message: This is the planet zork speaking.
```

4. (8 points) What is the result of the following expression?

```
(let ((v (vector 'a 'fine 'mess 'you 'have 'gotten 'us 'into)))
  (let ((another-vec v)
        (yet-another (vector 'a 'fine 'mess 'you 'have 'gotten 'us 'into)))
    (let ((part-a (eq? v another-vec))
          (part-b (eq? v yet-another))
          (part-c (equal? another-vec yet-another)))
      (vector-set! v 3 'sam)
      (vector-set! another-vec 4 'tina)
      (vector-set! yet-another 4 'rover)
      (vector 'a: part-a   ; yes, a symbol may have a colon in it, like a:
              'b: part-b
              'c: part-c
              'd: (vector-ref v 3)
              'e: (vector-ref another-vec 3)
              'f: (vector-ref yet-another 3)
              'g: (vector-ref yet-another 4)
              'h: (equal? v yet-another)))))
```

5. (15 points) Write a procedure `vector-map2-c` that takes a procedure `proc` of type (→ (S T) U), and returns a procedure of that takes a vector `v1`, and returns a procedure that takes a vector `v2` of the same length as `v1` and returns a new vector whose ith element is the result of applying `proc` to the ith elements of `v1` and `v2`. Do not mutate either `v1` or `v2`. For example,

```
(((vector-map2-c +) (vector 1 2 3)) '#(4 5 6)) ==> #(5 7 9)
(((vector-map2-c -) (vector 1 2 3)) '#(4 5 6)) ==> #(-3 -3 -3)
(((vector-map2-c -) (vector )) '#()) ==> #()
(((vector-map2-c equal?) '#(a b (+ 6 7))) '#(a xy 13)) ==> #(#t #f #f)
(((vector-map2-c vector) '#(5 6 7 8)) '#(a b c d))
     ==> #(#(5 a) #(6 b) #(7 c) #(8 d))
```

6. (10 points) Define a procedure `vector-assign!` that takes a vector `target` and a vector `source`, which must have the same length as `target`, and mutates `target` so that for all indexes i, the ith element of `target` is the ith element of `source`. There is no return value for `vector-assign!`. For example, consider the following transcript.

```
> (define v (vector 'school 'is 'out 'for 'winter))
> (define avec (vector 'a 'good 'time 'for 'reading))
> (vector-assign! v avec)
> v
#(a good time for reading)
> (vector-assign! v '#(3 5 7 9 11))
> v
#(3 5 7 9 11)
> avec
#(a good time for reading)
```

7. (10 points) Using `while-proc`, define a procedure `vector-subst!`. The procedure `vector-subst!` takes three arguments: an item `new`, an item `old`, and a vector, `vec`, and mutates `vec` so that each occurrence of `old` in `vec` if any, is changed to `new`. There is no return value for `vector-subst!`. For example, consider the following transcript.

```
> (define v '#(3 5 7 9 11))
> (vector-subst! 'it 9 v)
> v
#(3 5 7 it 11)
> (vector-subst! 'time 'it v)
> v
#(3 5 7 time 11)
> (define v2 '#(i found it and it is really it))
> (vector-subst! 'time 'it v2)
> v2
#(i found time and time is really time)
```

You must use `while-proc` *in your solution.*

8. (10 points) What is the result of the following expression?

```
(let ((a 'hope)
      (b 'that)
      (c 0)
      (d 'i)
      (e 'have))
  (set! b a)
  (set! e (+ 200 27 c))
  (set! a d)
  (set! c 'you)
  (set! d 'enjoyed)
  (vector a b c d e))
```