Spring, 1996 Name: ______________________
. My Section Day and Time : ______________________

Com S 342 — Principles of Programming Languages

Test on *EOPL* Chapters 1 and 2

This test has 9 questions and pages numbered 1 through 7.

Reminders

This test is closed book and 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.

For programs, indentation is important to us for "clarity" points; if your code is sloppy or hard to read, you will lose points. We may take off a small amount if you do not give `TYPE` comments for recursive helping procedures. However, you do not have to write such comments for procedures for which the type is stated in the problem. Correct syntax also matters. Check your code over for syntax errors. You will lose points if your code has syntax errors.

Subset of Scheme You May Use

Unless otherwise stated, when defining procedures you may only use: helping procedures that you define yourself, comments, and the procedures and keywords that are included in the following list. (That is, on this test do *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, *, +, -, /, <, <=, =, >=, >,
and, andmap, append, apply, boolean?, car, cdr,  c...r, char?,
cond, cons, define, display, else, eq?, equal?, eqv?, error,
if, let, letrec, lambda, list, length, list?, map, newline,
not, null?, number?, or, pair?, procedure?, quote, string,
string?, string=?, string-append, string-ci=?, string-length,
string-ref, string->list, string->number, string->symbol,
substring, symbol?, vector, vector?, vector-length,
vector->list, vector-ref, zero?
```

1. (10 points) Consider the following grammar.

   $\langle$expr$\rangle$ ::= $\langle$number$\rangle$
   | `[` $\langle$number$\rangle$ `:` $\{\langle$number$\rangle\}^*$ `]`
   | `(` $\langle$expr$\rangle$ $\langle$b-op$\rangle$ $\langle$expr$\rangle$ `)`
   | $\langle$u-op$\rangle$ $\langle$expr$\rangle$
   $\langle$number$\rangle$ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
   $\langle$b-op$\rangle$ ::= `+` | `-` | `*`
   $\langle$u-op$\rangle$ ::= `transpose` | `invert`

   In each of the spaces provided ("_____") below, write "yes" if the text is an example of an $\langle$expr$\rangle$ in the above grammar, and "no" if it is not.

   (a) _____ `[ 5 4 1 ]`
   (b) _____ `[ 3 :  4 2 1 4 5 ]`
   (c) _____ `[ 5 :  ( 4 + 2 ) ]`
   (d) _____ `( x + 3 )`
   (e) _____ `( [ 2 :  1 3 ] + transpose [ 4 :  8 9 7 5 ] )`

2. (5 points) Using Scheme, give an example of a curried procedure. (Don't ask us what a "curried procedure" is, you're supposed to know that.)

3. (5 points) Briefly describe how curried procedures can be useful.

4. (15 points) In each of the spaces provided ("_____") below, write, in set brackets, the entire set of the free variables in the preceeding Scheme expression. For example, write $\{x, y\}$, if the free variables are x and y. If there are no free variables, write $\{\}$. (You're supposed to know what a "free variable" is.)

   (a) `(lambda (ls) (car (cdr ls)))` ____________________

   (b)
   ```
   (lambda (h)
      (lambda (g)
         (lambda (x)
            (h (g x)))))
   ```
   ____

   (c)
   ```
   ((lambda (q r) (f q r))
    x
    (p q))
   ```
   ____

5. (10 points) In the following expression, draw an arrow from each bound ⟨varref⟩ to its declaration.

```
(lambda (f g h)

  ((lambda (x)

     (lambda (z w x)

       (f (g w) x z)))

   (lambda (x ls f)

     (lambda (g)

       (g (f x ls))))))
```

6. (10 points) Consider the following expression.

```
(lambda (append head tail)
  ((lambda (f)
     (lambda (ls append head)
       (head (append (f ls tail) ls))))
   (lambda (z w)
     (lambda (x)
       (append (head z) (w x))))))
```

Give the lexical address form of the above expression, by filling in the blanks below, replacing all the ⟨varref⟩s in the above expression by their lexical addresses. (You're supposed to know what a "lexical address" is.)

```
(lambda (append head tail)
  ((lambda (f)
     (lambda (ls append head)
       ________________________________________________________  ))
   (lambda (z w)
     (lambda (x)
       ________________________________________________________  ))))
```

7. (20 points) Write a procedure, `graph`, with type

```
(-> ((-> (number) number) (list number))
    (list (list number)))
```

that takes a procedure, `f`, and a list of numbers, `lon`, and returns a list of two-element lists of numbers. The first element in each two-element list is an element of `lon`, and the second is the value of invoking `f` on that element. The following are examples.

```
(graph (lambda (x) (+ x 1)) '())
  ==> ()
(graph (lambda (x) (+ x 1)) '(3 7 10))
  ==> ((3 4) (7 8) (10 11))
(graph (lambda (y) (* y 3)) '(1 2 3 4 5 6 7))
  ==> ((1 3) (2 6) (3 9) (4 12) (5 15) (6 18) (7 21))
(graph (lambda (z) (* z z)) '(1 2 3 4 5 6 7))
  ==> ((1 1) (2 4) (3 9) (4 16) (5 25) (6 36) (7 49))
```

8. (20 points) Without using `vector->list`, write a procedure, `vector-positive?`, which has the following type.

   ```
   (-> ((vector number)) boolean)
   ```

   The procedure `vector-positive?` takes a vector of number, `von`, and returns `#t` just when each number in `von` is positive (strictly greater than 0). The following are examples.

   ```
   (vector-positive? '#()) ==> #t
   (vector-positive? '#(1 2 3 4)) ==> #t
   (vector-positive? '#(2 3 4)) ==> #t
   (vector-positive? '#(0 0 7)) ==> #f
   (vector-positive? '#(8 4 -1 3 7)) ==> #f
   (vector-positive? '#(3008 44 33 7 0)) ==> #f
   ```

9. (30 points) Recall the grammar for the language lambda-1:

```
⟨exp⟩ ::= ⟨varref⟩
       | ( lambda ( ⟨var⟩ ) ⟨exp⟩ )
       | ( ⟨exp⟩ ⟨exp⟩ )
⟨varref⟩ ::= ⟨var⟩
⟨var⟩ ::= ⟨symbol⟩
```

Write a procedure, `fv-map`, with the following type

```
  (-> ((-> (symbol) exp) exp) exp)
```

that takes a procedure, `proc`, and an ⟨exp⟩, `e`, and returns an ⟨exp⟩ that is the same shape as `e`, but in which each free varref x is replaced by the value of `(proc` x`)`. (Don't ask us what a "free varref" is, you're supposed to know that.)

The following are examples.

```
(fv-map (lambda (v) 'z) 'x) ==> z
(fv-map (lambda (v) 'z) '(x y)) ==> (z z)
(fv-map (lambda (v) 'z) '(lambda (x) (car x)))
                     ==> (lambda (x) (z x))
(fv-map (lambda (v) v) '(lambda (x) (car x)))
                    ==> (lambda (x) (car x))
(fv-map (lambda (v) (list 'f v)) 'x) ==> (f x)
(fv-map (lambda (v) (list 'f v)) '(lambda (x) x))
                              ==> (lambda (x) x)
(fv-map (lambda (v) (list 'f v))
        '(x (x y)))
     ==> ((f x) ((f x) (f y)))
(fv-map (lambda (v) (list 'f v))
        '(lambda (x) (x (x y))))
     ==> (lambda (x) (x (x (f y))))
(fv-map (lambda (v) (list 'f v))
        '(x (lambda (x) (x (x y)))))
     ==> ((f x) (lambda (x) (x (x (f y)))))
(fv-map (lambda (v) (list 'f v))
       '(lambda (y) (x (lambda (x) (x (x y))))))
    ==> (lambda (y) ((f x) (lambda (x) (x (x y)))))
```

Hint: use a helping procedure with an accumulator (but don't try to make the helping procedure tail-recursive).

There is space for your answer on the next page.