Fall, 1995 Name: _____

My Section Number: _____ My Section Day and Time : _____

Com S 342 — Principles of Programming Languages

Test on *EOPL* Chapters 1 to 2.2

This test has 5 questions and pages numbered 1 through 4.

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.

Indentation is important to us for "clarity" points; if your code is sloppy or hard to read, you will lose points. Please indent as described in class.

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.

⟨expr⟩ ::= ⟨var⟩
    | `(` `fn` ⟨var⟩ `=>` ⟨expr⟩ `)`
    | ⟨expr⟩ `(` ⟨expr⟩ `)`
⟨var⟩ ::= `x` | `y` | `z`

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

(a) _____ `( fn x => x )`

(b) _____ `( fn y => y x )`

(c) _____ `( ( fn x => x ) y )`

(d) _____ `x ( z )`

(e) _____ `( fn x => ( fn y => x ( x ( y ) ) ) )`

2. (10 points) Briefly describe one (1) way that Scheme exhibits regularity. (Don't ask us what regularity means, you're supposed to know that.)

3. (10 points) Write a procedure, `duplicate-each` that takes a list of symbols, `lsym`, and returns a list with two copies of each symbol, in the same order as the original list. The following are examples.

```
> (duplicate-each '(a b c a))
(a a b b c c a a)
> (duplicate-each '(b c a))
(b b c c a a)
> (duplicate-each '())
()
> (duplicate-each '(it was a dark and stormy night))
(it it was was a a dark dark and and stormy stormy night night)
```

4. (15 points) Write a procedure, `product-nn`, that takes a list of numbers, `lon`, and a datum, `errvalue`, and returns the product of the numbers in the list, except that it returns `errvalue` if there is a number in the list that is negative.

   The following are examples.

```
> (product-nn '(1 2 3 4) 'found-negative)
24
> (product-nn '() 'found-negative)
1
> (product-nn '(-1 -2) 'found-negative)
found-negative
> (product-nn '(8 9 10 -1 3 4) #f)
#f
> (product-nn '(8 9 10 3 4) #f)
8640
> (product-nn (list 10 20 30) 'kaboom)
6000
```

5. (15 points) Write a procedure, `slst-map`, that takes a procedure, `proc`, and a ⟨s-list⟩, `slst`, and returns a ⟨s-list⟩ that has each symbol s in `slst` replaced by the value of `(proc` s`)`. (Recall that a ⟨s-list⟩ is defined by the following grammar.)

⟨s-list⟩ ::= ({⟨symbol-expression⟩}*)
⟨symbol-expression⟩ ::= ⟨symbol⟩ | ⟨s-list⟩

The following are examples.

```
> (slst-map (lambda (sym) 'z) '((a) () ((b () c)) d))
((z) () ((z () z)) z)
> (slst-map (lambda (sym) 'z) '())
()
> (slst-map (lambda (sym) (if (eq? sym 'a) 'x sym)) '(a b c a))
(x b c x)
> (slst-map (lambda (sym) (if (eq? sym 'a) 'x sym)) '(b () c ((a))))
(b () c ((x)))
> (slst-map (lambda (sym) (if (eq? sym 'a) 'x sym)) '((a) b () c ((a))))
((x) b () c ((x)))
> (slst-map (lambda (sym) (if (eq? sym 'a) 'x sym)) '((a a) b () c ((a))))
((x x) b () c ((x)))
```