Fall, 1997 Name: _____

. My Section Time : _____

Com S 342 — Principles of Programming Languages

Test on *EOPL* Chapters 1 to 2.2

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

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.

⟨bexp⟩ ::= ⟨atomic⟩
    | (∀ ⟨var⟩ : ⟨bexp⟩ : ⟨bexp⟩)
    | ⟨bexp⟩ ⟨lop⟩ ⟨bexp⟩
    | (⟨bexp⟩)
⟨atomic⟩ ::= P | Q | R | f . ⟨var⟩
⟨lop⟩ ::= ∧ | ∨ | ⇒
⟨var⟩ ::= i | j | x | y | z

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

(a) _____ `x = 3`

(b) _____ `f . x ∨ P`

(c) _____ `(∀ x : x = 3 : f . x)`

(d) _____ `(∀ x : f . x : P ∧ f . i)`

(e) _____ `P ∨ (∀ y : Q : f . x ⇒ (R))`

2. (5 points) Using Scheme, write a definition for a curried procedure that is a version of the following. (Don't ask us what a "curried procedure" means, you're supposed to know that.)

```
(define dielectric-force
  ; TYPE: (-> (number number number) number)
  (lambda (e1 r e2)
        (/ (* E e1 (- e2))
           (* r r))))
```

3. (20 points) Write a procedure, `append-all`, with type

```
(-> ((list (list T))) (list T))
```

that takes a list of lists, and returns a list of all these lists appended together in the same order. In your code you may only use Scheme's `append` with two (2) arguments. The following are examples.

```
(append-all '())
  ==> ()
(append-all '((3 7 10) (20 25 30) (5 4 3)))
  ==> (3 7 10 20 25 30 5 4 3)
(append-all '((20 25 30) (5 4 3)))
  ==> (20 25 30 5 4 3)
(append-all '((l i k) (e) (t h i s) () (o k)))
  ==> (l i k e t h i s o k)
```

4. (10 points) Write a Scheme procedure `append*` with the type

```
(-> ((list T) ...) (list T))
```

that takes 0 or more argument lists and returns a list of all these argument lists appended together in the same order. In your code you may only use Scheme's `append` with two (2) arguments; you may use the `append-all` procedure if you wish as well. The following are examples.

```
(append* )
  ==> ()
(append* '(3 7 10) '(20 25 30) '(5 4 3))
  ==> (3 7 10 20 25 30 5 4 3)
(append* '(20 25 30) '(5 4 3))
  ==> (20 25 30 5 4 3)
(append* '(l i k) '(e) '(t h i s) '() '(o k))
  ==> (l i k e t h i s o k)
```

5. (20 points) Without using `vector->list`, write a procedure, `vector-same?`, which has the following type.

```
(-> ((vector symbol)) boolean)
```

The procedure `vector-same?` takes a vector of symbols `vos`, and returns `#t` just when each of the symbols in `vos` is the same. You may assume that `vos` has at least one element. The following are examples.

```
(vector-same? '#(ok)) ==> #t
(vector-same? '#(yeah yeah)) ==> #t
(vector-same? '#(what is the most important idea in real estate)) ==> #f
(vector-same? '#(location location location)) ==> #t
(vector-same? '#(and and in and in computers)) ==> #f
(vector-same? '#(think think think)) ==> #t
```

6. (30 points) Consider the following grammar.

⟨exp⟩ ::= ⟨number⟩ | ⟨varref⟩
    | `(print` ⟨exp⟩`)`
    | (⟨exp⟩ `-` ⟨exp⟩) | (⟨exp⟩ `/` ⟨exp⟩)
⟨varref⟩ ::= ⟨symbol⟩

In this grammar, the nonterminals ⟨number⟩, ⟨string⟩, and ⟨symbol⟩ have the same syntax as in Scheme. Write a procedure, `subst-exp`, with the following type

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

that takes two symbols `new` and `old`, and an ⟨exp⟩, `e`, and returns an ⟨exp⟩ that is the same as `e`, except that each ⟨varref⟩ in `e` that is the same as `old` is replaced by the value of `new`. The following are examples.

```
(subst-exp 'new 'old 3) ==> 3
(subst-exp 'z 'x 'x) ==> z
(subst-exp 'z 'x 'y) ==> y
(subst-exp 'z 'x '(x - x)) ==> (z - z)
(subst-exp 'z 'x '((x - x) - (3 / x))) ==> ((z - z) - (3 / z))
(subst-exp 'z 'x '(print (3 - (y / x)))) ==> (print (3 - (y / z)))
(subst-exp 'total 't '((t / 0.34) - (x - 5))) ==> ((total / 0.34) - (x - 5))
(subst-exp '+ '- '(x - x)) ==> (x - x)
(subst-exp 'f 'print '(print 5)) ==> (print 5)
```

Hint: don't hesitate to write helping procedures. There is more space on the next page.