Fall, 1995 Name: _____

. My Section Day and Time : _____

Com S 342 — Principles of Programming Languages

Final test focusing on *EOPL* 5.7, 6.1-6.3

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

Reminders

For this test, you can use one page (one side, no less than 10pt font) of notes. These notes are to be handed in at the end of the test.

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 in a problem, when solving problems you may only use standard features of the language that we discussed in class, and helping functions that you define yourself. The standard is defined by the *Revised*^{4} *Report on the Algorithmic Language Scheme*.

Parts of Scheme You May *Not* Use

Unless otherwise stated in a problem, you are prohibited from using internal defines, all the input and output facilities, macros, and the following keywords and procedures. (Don't worry if you don't know what these are.)

```
call-with-current-continuation  do
```

1. This is a question about parameter passing mechanisms. Choose a programming language you know (e.g., Scheme, Common Lisp, C, C++, Pascal, FORTRAN) and tell us:

   (a) The name of the language.

   (b) (5 points) The name of its default parameter passing mechanism. (By default, we mean the mechanism used if no additional syntax, like `&` in C++ or `var` in Pascal is used to declare formal parameters.)

   (c) (5 points) The name of its array model.

2. (5 points) What is the name of the scope rule used with dynamic assignment?

3. (5 points) Briefly describe why the following does not work with dynamic scoping.

```
let ct = proc(m)
           proc(r)
             *(m, r)
   in (ct(5))(6);
```

4. (10 points) This is a question about dynamic scoping. Consider the following expression in the defined langauge (using call-by-value).

```
let  x = 3; y = 4
in let f = proc(x) *(x, y);
       g = proc(y) f(+(5,y))
   in g(+(x,2))
```

Using dynamic scoping, (a) Draw a picture of the run-time stack to show how the computation proceeds, and (b) give the result of the above expression.

5. (10 points) This is a question about dynamic assignment. Consider the following expression in the defined langauge (using call-by-value).

```
let  x = 3; y = 4
in let f = proc(x) *(x, y);
       g = proc() x := f(+(5,y))
   in begin
        y := +(x,2) during g();
        list(x,y)
      end
```

Give the result of the above expression.

6. (80 points) This is a problem about parameter passing mechanisms and array models. Consider the following expression

```
letarray a[2]; b[2]
in begin
     a[0] := 10; a[1] := 11; b[0] := 20; b[1] := 21;
     let x = 3
     in let p = proc(c,e,v,w)
                  begin
                    b[0] := v; a := c; v := e; w := x;
                    %%% draw a picture for this point
                    c[0]
                  end
        in let r = p(b,a[1],x,b[0])
           in list(r, a[0], a[1], b[0], b[1], x)
   end
```

For each of the following combinations of parameter passing mechanism and array model: (i) draw a picture of the execution (as discussed in class) for the point noted by the comment, and (ii) give the output of the expression. The combinations you are to do are as follows (there are more on the next page).

(a) Call-by-value with the indirect model.

(b) Call-by-value with the direct model.

Here is another copy of the expression, for your convenience.

```
letarray a[2]; b[2]
in begin
     a[0] := 10; a[1] := 11; b[0] := 20; b[1] := 21;
     let x = 3
     in let p = proc(c,e,v,w)
                  begin
                    b[0] := v; a := c; v := e; w := x;
                    %%% draw a picture for this point
                    c[0]
                  end
        in let r = p(b,a[1],x,b[0])
           in list(r, a[0], a[1], b[0], b[1], x)
   end
```

(c) Call-by-reference with the indirect model

(d) Call-by-reference with the direct model

7. (10 points) This is a problem about call-by-value-result. Consider the following expression.

```
let x = 3; y = 10
in let p = proc(a, b)
             begin
               a := +(a, b);
               b := +(b, x);
               a := +(a, y);
               %%% draw a picture for this point
               a
             end
   in let r = p(x, y)
      in list(r,x,y)
```

   Using call-by-value-result: (i) draw a picture of the execution (as discussed in class) for the point noted by the comment, and (ii) give the result of the expression.

8. (10 points) Briefly explain the following in English. What changes are made in the interpreter when changing from the indirect model of arrays to the direct model (in the call-by-value interpreter)?

9. (13 points) Write a version of `syntax-expand`, which takes a parsed expression and returns a parsed expression, expanding the following syntactic sugar for an `unless` expression, where *test* and *body* are each an ⟨exp⟩ in the grammar.

`unless` *test* `do` *body* ⇒ `if` *test* `then 0 else` *body*

The following is an example.

```
(syntax-expand
 (parse "+(3, unless gt(x,4) do x := *(x,x)"))
 =  (parse "+(3, if gt(x,4) then 0 else x := *(x,x)"))
```

The syntax and abstract syntax your code should handle are given below.

⟨exp⟩ ::= ⟨varref⟩	`varref (var)`
| ⟨integer-literal⟩	`lit (datum)`
| ⟨operator⟩ `(`⟨operands⟩`)`	`app (rator rands)`
| `if` ⟨exp⟩ `then` ⟨exp⟩ `else` ⟨exp⟩	`if (test-exp then-exp else-exp)`
| `(proc` ⟨varlist⟩ ⟨exp⟩	`proc (formals body))`
| `unless` ⟨exp⟩ `do` ⟨exp⟩	`unless (test body)`
⟨operator⟩ ::= ⟨varref⟩ | `(`⟨exp⟩`)`	
⟨operands⟩ ::= `( )` | `(`⟨exp⟩ {`,` ⟨exp⟩}*`)`	

Your code should expand `unless` expressions nested within other expressions.