Fall, 1997 Name: _____

. My Section Day and Time : _____

Com S 342 — Principles of Programming Languages

Test on *EOPL* Chapters 6.1–3, 7, and a bit of 5

This test has 10 questions and pages numbered 1 through 9.

Reminders

For this test, you can use one (1) page (8.5 by 11 inches, one (1) side, no less than 9pt font) of notes. No photo-reduction is permitted. These notes are to be handed in at the end of the test. Use of other notes or failure to follow these instructions will be considered cheating.

During the test, if you need more space for an answer, 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 will 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 in a problem, when solving problems you may only use standard ADTs used in the interpreter (such as cells, environments, etc.), standard features of Scheme that we discussed in class, `define-record`, `variant-case`, 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 defining your own macros, and from using internal defines, all the input and output facilities, and the following keywords and procedures. (Don't worry if you don't know what these are.)

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

1. (5 points) In the the defined language with objects and classes (the interpreter of section 7.1), what parameter passing mechanism is used to call methods?

2. (10 points) Briefly describe how a closure is like an object.

3. (5 points) What is a meta-class?

4. (5 points) What is the default parameter passing mechanism used in Scheme and Java?

5. (5 points) What is the array model used in Scheme and Java?

6. (15 points) Briefly explain the following in English. Assuming the indirect model of arrays, what changes are made in the (chapter 6) interpreter when changing from call-by-value to call-by-reference?

7. (10 points) Consider the following expression in the defined language.

```
letrecproc
  fact(x) = if zero(x) then 1 else *(x, fact(-(x,1)))
in fact(12)
```

Write, in the defined language's concrete syntax, an equivalent desugared form of the above expression, which does not use `letrecproc` or `letrec`. (You may use `let` in the desugared form.)

8. (30 points) In this problem you will implement the following syntax in the defined language.

⟨exp⟩ ::= `foreach` ⟨var⟩ `in` ⟨exp⟩ `do` ⟨body⟩
⟨body⟩ ::= ⟨exp⟩

Use the following for the abstract syntax of a `foreach`-expression.

```
(define-record foreach (var list-exp body))
```

The meaning of this syntax is that, if ⟨exp⟩ evaluates to a list, then ⟨body⟩ is evaluated (for its side-effects) for each element of the list, with ⟨var⟩ bound to a cell containing each successive element of the list. The evaluation starts with the element in the head of the list, if any, and continues to the other end of the list. The value returned by a `foreach` expression is 0.

The ⟨exp⟩ should only be evaluated once. The region of the ⟨var⟩ declared in the `foreach` expression is just the ⟨body⟩ of that expression. (For this problem, assume static scoping, call-by-value, and the indirect model.)

For example the following expression would return 10.

```
let total = 0
in begin
     foreach elem in list(0,1,2,3,4) do total := +(total, elem);
     total
   end
```

As another example, suppose ⟨exp⟩ evaluates to the list `(5 9 3 1)`. Then the interpreter is to bind the ⟨var⟩ to a cell containing 5 and evaluate ⟨body⟩. After this finishes, the interpreter binds the ⟨var⟩ to a cell containing 9 and evaluates ⟨body⟩ again. It continues by evaluating the ⟨body⟩ in an environment with the ⟨var⟩ bound to a cell containing 3 and then to 1.

To save time (on this test) you may assume that the ⟨exp⟩ evaluates to a list; that is you don't have to check for type errors.

Your task is to implement the above syntax, by filling in the code for the `foreach` case of `eval-exp` on the next page.

To save time, only give the code for the `foreach` case, and any auxiliary procedures that you call in that case. However, you don't have to rewrite the procedures for the environment ADT.

```
(define eval-exp
  ; TYPE: (-> (parsed-exp Environment) Expressed-Value)
  (lambda (exp env)
    (variant-case exp
      (lit (datum) datum)
      (varref (var) (cell-ref (apply-env env var)))
      ; ...
      ; put your code below
```

9. (15 points) Assuming static scoping, consider the following session with the defined language interpreter's read-eval-print loop

```
--> define i = 342;
--> define lst = emptylist;
--> define f = proc(x,l) begin x := +(8,i); l := cons(i, cons(x, lst)) end;
--> f(i, lst);
```

Fill in the following table with the final values of `i` and `lst` after running the above session, in each of the given parameter mechanisms. (If need be, you may use "?" to represent an undefined (i.e., unspecified) value.)

calling mechanism	ending value of `i`	ending value of `lst`
call-by-value		
call-by-reference		
call-by-value-result		

10. (60 points) This is a problem about parameter passing mechanisms and array models. Throughout this problem use static scoping. Consider the following expression.

```
letarray a[2]; b[2]
in begin
     a[0] := 3; a[1] := 6; b[0] := 9; b[1] := 12;
     let i = 0; j = 1
     in let g = proc(t, k, u,  u0,   y0,  m, n, y, x)
                  begin
                    k := +(k,t); j := -(m,i);
                    u0 := +(a[m], n); y := a; x[0] := b[k];
                    %%% draw a picture for this point
                    +(u0,u[0])
                  end
        in let r = g(1, i, a, a[i], b[0], j, j, b, a)
           in
              list(a[0], a[1], b[0], b[1], i, j, r)
   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 result of the expression. The combinations you are to do are as follows (there are more on the following pages).

(a) 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] := 3; a[1] := 6; b[0] := 9; b[1] := 12;
     let i = 0; j = 1
     in let g = proc(t, k, u,  u0,   y0,  m, n, y, x)
                  begin
                    k := +(k,t); j := -(m,i);
                    u0 := +(a[m], n); y := a; x[0] := b[k];
                    %%% draw a picture for this point
                    +(u0,u[0])
                  end
        in let r = g(1, i, a, a[i], b[0], j, j, b, a)
           in
              list(a[0], a[1], b[0], b[1], i, j, r)
   end
```

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

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

```
letarray a[2]; b[2]
in begin
     a[0] := 3; a[1] := 6; b[0] := 9; b[1] := 12;
     let i = 0; j = 1
     in let g = proc(t, k, u,  u0,   y0,  m, n, y, x)
                  begin
                    k := +(k,t); j := -(m,i);
                    u0 := +(a[m], n); y := a; x[0] := b[k];
                    %%% draw a picture for this point
                    +(u0,u[0])
                  end
        in let r = g(1, i, a, a[i], b[0], j, j, b, a)
           in
              list(a[0], a[1], b[0], b[1], i, j, r)
   end
```

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