Fall, 1997 Name: ______________________________

My Section Day and Time : ______________________________

Com S 342 — Principles of Programming Languages

Test on *EOPL* Chapter 5

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

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. (10 points) In this problem you will add a primitive procedure `lessOrEqual` to the defined language. This procedure should return a value representing *true* if its first argument number is less than or equal to its second number argument, and a value representing *false* otherwise. (You're supposed to know how *true* and *false* are represented in the interpreter.)

   Your task is to add the primitive procedure `lessOrEqual` by filling in the code for the necessary changes below. If you need any auxiliary procedures for your definition, you must also write out those in your solution.

```
(define apply-prim-op
  ; TYPE: (-> (symbol (list Expressed-Value)) Expressed-Value)
  (lambda (prim-op args)
    (case prim-op
      ((+) (+ (car args) (cadr args)))
      ((-) (- (car args) (cadr args)))
      ((*) (* (car args) (cadr args)))
      ((add1) (+ (car args) 1))
      ((sub1) (- (car args) 1))

      (else (error "Invalid prim-op name:" prim-op)))))

(define prim-op-names  ; TYPE: (list symbol)
  '(+ - * add1 sub1
   ))
```

2. (10 points) Consider an interpreter that supports assignment (`:=`). Show the code you would write in such an interpreter that would define the initial environment such that the variable `e` would be predefined to contain the value 2.71828.

3. (10 points) Briefly (in 1 or 2 sentences) answer the following question. What changes were needed to the interpreter to make it use dynamic scoping?

4. (5 points) What is the scope rule used with dynamic assignment?

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

```
let roman = 1; italic = 2
in let font = roman; output = emptylist
   in let addchar = proc(c) output := cons(cons(c,font), output)
      in begin
            addchar(43);
            font := italic during addchar(52);
            list(font, output)
         end
```

Give the result of the above expression.

6. (5 points) Consider the following expression in the defined language.

```
letrecproc
  even(x) = if zero(x) then 1 else odd(sub1(x));
  odd(x)  = if zero(x) then 0 else even(sub1(x))
in odd(342)
```

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

7. (10 points) This is a question about dynamic scoping. Consider the following expression in the defined langauge (using call-by-value).

```
let  x = 100; lst = list(3000)
in let addxlist = proc(lst)
                  if null(lst)
                  then x  %%% draw the picture for this point
                  else +(car(lst), addxlist(cdr(lst)))
   in let x = 5; lst = list(6)
      in addxlist(list(7, 9))
```

Using dynamic scoping, (a) draw a picture of the run-time stack when execution reaches the point indicated (with the stack growing up the page), and (b) give the result (if any) of the above expression. (If the expression has no result, or encounters an error, write that.)

8. (20 points) In this problem you will implement the following syntax in the defined language.

   ⟨exp⟩ ::= `and` ⟨exp⟩ ⟨exp⟩

   Use the following as the abstract syntax for the `and`-expression.

```
(define-record and-exp (left-exp right-exp))
```

   The meaning of this syntax is supposed to be that of a short-circuit "and". For example, in the defined language we would have

```
and 1 0  ==> 0
and 0 1  ==> 0
and 1 1  ==> 1
and 0 0  ==> 0
letrecproc ohno() = ohno() in and 0 ohno()  ==> 0
```

   That is, `and` e_1 e_2 is equivalent to `let x =` e_1 `in if x then` e_2 `else x`. However, you are *not* to implement this as a syntactic sugar. Instead you will implement this in `eval-exp` directly, by filling in the code for the `and-exp` case of `eval-exp` below.

   To save time, only give the code for the `and-exp` case, and any auxiliary procedures that you call in that case.

   Hint: think about the types!

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

   ⟨exp⟩ ::= `case` ⟨exp⟩ `of` ⟨one-cases⟩ `end`
   ⟨one-cases⟩ ::= ⟨one-case⟩ | ⟨one-case⟩ {`;` ⟨one-case⟩}*
   ⟨one-case⟩ ::= ⟨label⟩ `:` ⟨body⟩
   ⟨label⟩ ::= ⟨integer-literal⟩
   ⟨body⟩ ::= ⟨exp⟩

   We will use the following for the abstract syntax. Note that the `cases` field of a `case-exp` record has the type `(list one-case)`. Also the `label` field of a `one-case` record has type `integer`.

```
(define-record case-exp (exp cases))
(define-record one-case (label body))
```

   The meaning of this syntax is that, if ⟨exp⟩ evaluates to an integer that occurs as a label in some ⟨one-case⟩ in the list of ⟨one-case⟩s, then the first such ⟨one-case⟩ has its ⟨body⟩ evaluated, and that value is returned as the value of the `case` expression; otherwise 0 is returned as the value of the `case` expression.

   Note that the ⟨exp⟩ between `case` and `of` should only be evaluated once. To save time (on this test) you may assume that this ⟨exp⟩ evaluates to an integer; that is you don't have to check for the user's type errors.

   The following are examples of evaluations in the defined language.

```
case 3 of 3: 7 end
     ==> 7

case 9 of 3: 7; 9: 22 end
     ==> 22

case 9 of 9: 30; 3: 7; 9: 22 end
     ==> 30

case +(2,5) of 1: 1; 2: begin x := 7; +(x,1) end; 7: 10; 8: 31 end
     ==> 10

case *(4,+(1,2)) of 1: 8; 2: 14 end
     ==> 0

case *(4,+(1,2)) of 1: 8; 2: 14; 1: 9; 5: 4 end
     ==> 0
```

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

To save time, only give the code for the `case-exp` case, and any auxiliary procedures that you call in that case.

```
(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
```