Fall, 1998 Name: ______________________

My Section Day and Time : ______________________

Com S 342 — Principles of Programming Languages

Test on *EOPL* Chapter 5

This test has 8 questions and pages numbered 1 through 8.

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*.

There is a list of the types of some of the procedures from the "standard ADTs" on the next page.

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
```

Types of helpers from the “standard ADTs” for chapter 5:

```
;; Expressed-Value ADT
number->expressed : (-> (number) Expressed-Value)
expressed->number : (-> (Expressed-Value) number)
procedure->expressed : (-> (Procedure) Expressed-Value)
expressed->procedure : (-> (Expressed-Value) Procedure)
list->expressed : (-> ((list Expressed-Value)) Expressed-Value)
expressed->list : (-> (Expressed-Value) (list Expressed-Value))
void->expressed : (-> (void) Expressed-Value)

expressed->denoted : (-> (Expressed-Value) Denoted-Value)
denoted->expressed : (-> (Denoted-Value) Expressed-Value)

;; Denoted-Value ADT
make-cell : (-> (Expressed-Value) Denoted-Value)
cell-ref : (-> (Denoted-Value) Expressed-Value)
cell-set! : (-> (Denoted-Value Expressed-Value) void)
cell-swap! : (-> (Denoted-Value Denoted-Value) void)

;; Procedure ADT
prim-proc? : (-> (Procedure) boolean)
make-prim-proc : (-> (symbol) Procedure)
prim-proc->prim-op : (-> (Procedure) symbol)

closure? : (-> (Procedure) boolean)
make-closure : (-> ((list symbol) parsed-exp Environment) Procedure)
closure->formals : (-> (Procedure) (list symbol))
closure->body : (-> (Procedure) parsed-exp)
closure->env : (-> (Procedure) Environment)

;; from ignore.scm
ignore : (-> (T) void)
```

1. (10 points) In this problem you will add a primitive procedure `isZero` to the defined language. This procedure should return a value representing *true* if its argument is the number zero (0), and a value representing *false* if its argument is some other number. (You're supposed to know how *true* and *false* are represented in the interpreter.)

   Your task is to add the primitive procedure `isZero` by filling in the code for the necessary changes below. Assume that you are given the appropriate procedures for the domain Expressed-Value, such as `expressed->number` (see the previous page), but if you need any other 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
      ((+) (number->expressed (+ (expressed->number (car args))
                                 (expressed->number (cadr args)))) )
      ((*) (number->expressed (* (expressed->number (car args))
                                 (expressed->number (cadr args)))) )
      ((-) (number->expressed (- (expressed->number (car args))
                                 (expressed->number (cadr args)))) )
      ((add1) (number->expressed (+ (expressed->number (car args)) 1)) )
      ((sub1) (number->expressed (- (expressed->number (car args)) 1)) )



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

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

2. (10 points) Briefly (in 3 or 4 sentences) answer the following question. What changes were needed to the interpreter to allow local binding (the `let` special form)?

3. (5 points) What is the scope rule used with dynamic assignment?

4. (10 points) This is a question about dynamic assignment. Consider the following expression in the defined language.

```
let blue = 512; red = 1024
in let color = blue; output = emptylist
   in let addint = proc(c) output := cons(list(c, color), output)
      in begin
            addint(45);
            color := red during addint(33);
            cons(color, output)
         end
```

Give the result of the above expression.

5. (10 points) Consider the following expression in the defined language.

```
letrecproc
  map(f, ls) = foldr(proc(x,ans) cons(f(x),ans),
                     emptylist,
                     ls);
  foldr(f, z, ls) = if null(ls) then z
                                else f(car(ls), foldr(f,z,cdr(ls)))
in map(myFun, list(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.)

6. (15 points) This is a question about dynamic scoping. Consider the following expression in the defined language.

```
let y = 342; z = 541
in let f = proc(x)
             list(y, +(x, z)); %%% draw the picture when execution is here
       p = proc(y)
             f(let z = 52 in +(z, 3))
   in let y = 5; z = 10
      in p(+(y, z))
```

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.)

7. (20 points) In this problem you will implement the following syntax in the defined language.

   ⟨exp⟩ ::= `unless` ⟨exp⟩ `do` ⟨exp⟩

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

```
(define-record unless-exp (test-exp body))
```

   The meaning of this syntax is supposed to be that if the test expression (following `unless`) is *false*, then the body expression (following `do`) is evaluated and its value is returned; otherwise 0 is returned. For example, in the defined language we would have

```
let c = 7 in begin unless 0 do c := 2; c end ==> 2
let c = 7 in begin unless 1 do c := 2; c end ==> 7
```

   That is, `unless` e_1 `do` e_2 is equivalent to `if` e_1 `then 0 else` e_2. 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 `unless-exp` case of `eval-exp` below.

   To save time, only give the code for the `unless-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) (number->expressed datum))
      (varref (var) (denoted->expressed (apply-env env var)))
      ; ...
      ; put your code below
```

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 each successive element of the list. The evaluation starts at the front of the list.

   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
   ```

   For example, suppose ⟨exp⟩ evaluates to the list `(5 9 3 1)`. Then the interpreter is to bind ⟨var⟩ to 5 and evaluate ⟨body⟩. After this finishes, the interpreter binds ⟨var⟩ to 9 and evaluates ⟨body⟩ again. It continues by evaluating the ⟨body⟩ in an environment with ⟨var⟩ bound to 3 and then to 1. The value returned by a `foreach` expression is 0.

   Note that ⟨exp⟩ should only be evaluated once. The region of the ⟨var⟩ declared in the `foreach` expression is just the ⟨body⟩ of that expression.

   To save time (on this test) you may assume that the ⟨exp⟩ evaluates to a list.

   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.

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