Spring, 1996 Name: _____

. My Section Day and Time : _____

Com S 342 — Principles of Programming Languages

Test on *EOPL* Chapters 5.1–5

This test has 7 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 8pt 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.

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 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 `notequal` to the defined language. This procedure should return a value representing *false* if its two argument numbers are equal, and a value representing *true* otherwise. (You're supposed to know how *true* and *false* are represented in the interpreter.)

   Your task is to add the primitive procedure `notequal` 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: (-> (prim-proc (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) Briefly describe what you would have to do to add the variable `pi` (which should denote 3.14159) as a built-in variable to an interpreter that supports assignment (`:=`). Give the relevant code for maximum credit.

3. (10 points) Briefly (in 1 or 2 sentences) answer the following question. What changes needed to be made to the interpreter to handle `let` expressions?

4. Briefly (in 1 or 2 sentences) answer the following questions.

   (a) (5 points) What is a closure?

   (b) (10 points) Why are closures needed in the interpreter?

   (c) (10 points) Are closures needed in a language, like FORTRAN 77 and C, that does not allow nested procedures (i.e., procedure declarations to appear within the scope of other procedures)? Why or why not?

5. (20 points) Write a version of `syntax-expand`, which takes a parsed expression and returns a parsed expression, expanding the following syntactic sugar for a `when` expression, where *test* and *body* are each an ⟨exp⟩ in the grammar.

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

   The following is an example.

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

   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))`
| `when` ⟨exp⟩ `do` ⟨exp⟩	`when (test body)`
⟨operator⟩ ::= ⟨varref⟩ | (⟨exp⟩)	
⟨operands⟩ ::= ( ) | (⟨exp⟩ {, ⟨exp⟩}*)	

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

6. (20 points) In this problem you will implement the following syntax in the defined language.

   ⟨exp⟩ ::= ⟨var⟩ `:=:` ⟨var⟩

   The meaning of this syntax is supposed to be that the values of the two variables are swapped.

   For example the following expression would return the list `(10 5)`.

```
let a = 5; b = 10
in begin
     a :=: b;
     cons(a, cons(b, emptylist))
   end
```

   Use the following for the abstract syntax for the `swap`-expression.

```
(define-record swap (left-var right-var))
```

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

   To save time, only give the code for the `swap` case, and any auxiliary procedures that are not in the standard interpreters and 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
```

7. (30 points) In this problem you will implement the following syntax in the defined language.

⟨exp⟩ ::= `repeat` ⟨var⟩ `:= 1 to` ⟨exp⟩ `do` ⟨body⟩
⟨body⟩ ::= ⟨exp⟩

The meaning of this syntax is that, if ⟨exp⟩ evaluates to a positive integer, then ⟨body⟩ is evaluated (for its side-effects) that number of times. For example, if ⟨exp⟩ evaluates to 5, then the ⟨body⟩ is evaluated 5 times; however, if ⟨exp⟩ evaluates to 0 or a negative number, then the ⟨body⟩ is not evaluated. Furthermore, the evaluation of ⟨body⟩ takes place in an environment extended with the variable ⟨var⟩; this variable is initialized to 1, and incremented by 1 after each time the ⟨body⟩ is evaluated. The value returned by a `repeat` expression is 0.

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

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

For example the following expression would return 10.

```
let sum = 0; last = 4
in begin
     repeat i := 1 to last
     do begin
          sum := sum + i;
          last := 5
        end;
     sum
   end
```

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

```
(define-record repeat-exp (var final-val body))
```

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

To save time, only give the code for the `repeat-exp` case, and any auxiliary procedures that are not in the standard interpreters and 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
```