Fall, 1995 Name: _____

. My Section Day and Time : _____

Com S 342 — Principles of Programming Languages

Test on *EOPL* 3.6, 4.1-3, 4.5-6, 5.1-5.6

This test has 5 questions and pages numbered 1 through 5.

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. (15 points) Simplify the following lambda calculus expressions as much as possible by using beta (β) and eta (η) reduction. (If the expression cannot be simplified, or gets into an infinite loop, write that.) You may use alpha (α) conversion if that helps you. For partial credit, show your work.

   (a)

```
((lambda (x)
   ((f x) 4))
 (lambda (x) x))
```

   (b)

```
(((lambda (z)
   (lambda (y)
      (z y)))
  (lambda (x) (x y)))
 x)
```

   (c)

```
(((lambda (x) (lambda (y) x))
  (lambda (x) x))
 ((lambda (x) (x x)) (lambda (x) (x x))))
```

   (Hint: use leftmost reduction.)

2. (10 points) In this problem you will add a primitive procedure `negative` to the defined language. This procedure should return a value representing *true* when called with a negative number, and a value representing *false* when called with 0 or a positive number. (You're supposed to know how *true* and *false* are represented in the interpreter. If you need any auxiliary procedures for your definition, write those into your solution.)

   Your task is to add the primitive procedure `negative` by making the necessary changes to the code from the interpreter below.

```
(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
   ))
```

3. (10 points)

   (a) What changes to the data structures (ADTs, domains) used in the interpreter were needed to handle variable assignment (`:=`)?

   (b) Briefly describe other changes to the code in the interpreter that were needed to handle the variable assignment itself and the changes to the data structures.

4. (20 points) In this problem you will implement the following syntax in the defined language.

⟨exp⟩ ::= `do` ⟨body⟩ `until` ⟨exp⟩
⟨body⟩ ::= ⟨exp⟩

The meaning of this syntax is supposed to be that the ⟨body⟩ is evaluated (for its side-effects), then the ⟨exp⟩ is evaluated; if it results in a true value, the value of the whole `do`-⟨exp⟩ is 0, otherwise, the evaluation process is repeated (by evaluating the ⟨body⟩, etc.).

For example the following expression would return 10.

```
let x = 4; sum = 0
in begin
     do begin
             sum := sum + x;
             x := x - 1
        end
     until (x = 0);
     sum
   end
```

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

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

Your task is to implement the above syntax, by filling in the code for the `do-exp` case of `eval-exp` below. To save time, only give the code for the `do-exp` case, and any auxiliary procedures 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
```

5. (25 points) Write a version of `syntax-expand`, which takes a parsed expression and returns a parsed expression, expanding the following syntactic sugar for `sequential-let`, where each d_i is a ⟨decl⟩, and e is an ⟨exp⟩ in the grammar.

$$\texttt{sequential-let}\ d_1\texttt{;}\ \ldots\texttt{;}d_n\ \texttt{in}\ e \quad \Rightarrow \quad \texttt{let}\ d_1\ \texttt{in}\ \ldots\ \texttt{let}\ d_n\ \texttt{in}\ e$$

The following is an example.

```
(syntax-expand
 (parse "+(3, sequential-let x = 4; y = *(x,x); z = *(y,x) in +(y,z))"))
 = (parse "+(3, let x = 4 in let y = *(x,x) in let z = *(y,x) in +(y, z))")
```

The syntax and abstract syntax (on the right) that 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)`
| `let` ⟨decls⟩ `in` ⟨exp⟩	`let (decls body)`
| `sequential-let` ⟨decls⟩ `in` ⟨exp⟩	`sequential-let (decls body)`
⟨operator⟩ ::= ⟨varref⟩ | `(`⟨exp⟩`)`	
⟨operands⟩ ::= `( )` | `(`⟨exp⟩ {`,` ⟨exp⟩}*`)`	
⟨decls⟩ ::= ⟨decl⟩ {`;` ⟨decl⟩}*	
⟨decl⟩ ::= ⟨var⟩ `=` ⟨exp⟩	`decl (var exp)`

Your code should expand `sequential-let` expressions nested within other expressions. It does not have to expand `let` expressions. (You may assume the interpreter handles `let`.)