Spring, 1999 Name: _____
My Section Letter: _____ My Section Day and Time : _____

Com S 342 — Principles of Programming Languages

Test on *EOPL* Sections 3.6, 4.5–6, and 5.1–5

This test has 7 questions and pages numbered 1 through 11.

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. Handwriting is okay. No photo-reduction is permitted. Don't use anything with printing on the other side, please. These notes are to be handed in at the end of the test. Have your name in the top right corner. 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*^{5} *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. (5 points) Assume for this problem that the defined language has been extended with the top-level form `define`, and with primitive procedures: `less` and `lessOrEqual`, `greater` and `greaterOrEqual`. Write in the defined language (not Scheme), a procedure, `largest`, that takes four numbers as parameters, and returns the largest of these. (You can use helping procedures if you wish.)

   For example:

```
largest(7,4,2,9)    ==> 9
largest(4,7,9,2)    ==> 9
largest(0,0,15,0)   ==> 15
largest(15,15,3,15) ==> 15
largest(0,0,0,0)    ==> 0
```

   Write your answer by completing the defined language code below.

```
define largest = proc(w, x, y, z)
```

2. (5 points) Briefly (in no more than 5 sentences) answer the following question. What changes were needed to the defined language's interpreter to support statically-scoped procedures (the `proc` special form)?

3. (10 points) In this problem you will add a primitive procedure `tail` to the defined language's interpreter. This procedure should return the tail of its argument, assuming that the argument is a non-empty list. For example `tail(list(3,4,5))` would be `(4 5)`. You do not have to check for errors.

   Your task is to add the primitive procedure `tail` 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->list` (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
   ))
```

4. This problem is about mutation and sharing in Scheme.

   (a) (5 points) Draw a box and pointer diagram for the state after executing all of the following top-level Scheme forms.

```
(define mylist (cons 9 (cons 10 '())))
(define ml2 (cons 8 mylist))
(set-car! mylist 7)
(set-cdr! ml2 (cdr mylist))
```

   (b) (5 points) Continuing from the above, what is the current value of the following expression?

```
(list 'mylist-is mylist
      'ml2-is ml2)
```

   (c) (5 points) Continuing from the above, what is the value of the following expression? (You can draw another diagram, but please leave the one for part (a) alone.)

```
(begin
  (set-car! (cdr mylist) 6)
  (list 'mylist-is mylist
        'ml2-is ml2))
```

5. (20 points) This problem is about transforming procedural to record representations.

   One can imagine a potentially infinite string as a mapping from the natural numbers to characters. This type will be called `i-string` below. Consider the following procedural representation of `i-string`.

```
(define i-string-such-that  ; TYPE: (-> ((-> (number) char)) i-string)
  (lambda (f)
    (lambda (n)
      (f n))))

(define i-string-update     ; TYPE: (-> (i-string number char) i-string)
  (lambda (s m c)
    (lambda (n)
      (if (= m n) c (i-string-access s n)))))

(define i-string-access     ; TYPE: (-> (i-string number) char)
  (lambda (s n)
    (s n)))
```

   Examples aren't really going to help you, but if we define

```
(define all-a      ; TYPE: i-string
  (i-string-such-that (lambda (n) #\a)))
(define alphabet   ; TYPE: i-string
  (i-string-such-that (lambda (n) (integer->char (remainder n 128)))))
```

   then the following are examples of how the above code works:

```
(i-string-access all-a 0) ==> #\a
(i-string-access all-a 342342342342342342342342) ==> #\a
(i-string-access alphabet 342342342342342342342342) ==> #\F
(i-string-access alphabet 70) ==> #\F
(i-string-access alphabet 71) ==> #\G
(i-string-access alphabet 103) ==> #\g
(i-string-access (i-string-update alphabet 103 #\a) 103) ==> #\a
(i-string-access (i-string-update alphabet 103 #\a) 70) ==> #\F
```

Your task is to transform the procedural representation on the previous page into one that uses records. Do this by giving the `define-record` declarations needed and the bodies of the procedures in the spaces provided on the next page. You must use `variant-case` in your solution.

```
;;; Write the define-record declarations below




;;; Now fill in the code for the operations below

(define i-string-such-that  ; TYPE: (-> ((-> (number) char)) i-string)




(define i-string-update     ; TYPE: (-> (i-string number char) i-string)




(define i-string-access     ; TYPE: (-> (i-string number) char)
```

6. (20 points) In this problem you will implement the following syntax in the defined language, starting from an interpreter that supports assignment (`:=`) and `begin`.

   ⟨exp⟩ ::= `do` ⟨body⟩ `until` ⟨test-exp⟩ | ...
   ⟨body⟩ ::= ⟨exp⟩
   ⟨test-exp⟩ ::= ⟨exp⟩

   Assume the following is the abstract syntax of a `do until` expression, where both the `body` and the `test-exp` fields are of type `parsed-exp`.

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

   The meaning of this syntax is that, first the ⟨body⟩ is evaluated (for its side-effects), then if the value of ⟨test-exp⟩ is a number that represents true, evaluation ends; otherwise the evaluation process is repeated. When evaluation ends, the result of the `do until` expression is the number 0.

   For example the following expression prints `0` then `1` then `2` then `3` and then `4`, and then returns 10.

   ```
   let total = 0;
       i = 0
   in
      begin
        do
          begin
            total := +(total, i);
            print(i);
            i := +(i,1)
           end
        until greater(i, 4);
        total
      end
   ```

   Note that ⟨test-exp⟩ should be evaluated each time around the loop.

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

To save time, only give the code for the `do-until` case, and any auxiliary procedures that you call in that case (if they are not given on page 2).

```
(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
```

7. (25 points) In this problem you will implement the following syntax in the defined language, starting from an interpreter that supports assignment (`:=`) and `begin`.

   ⟨exp⟩ ::= `within` ⟨body⟩ `use` ⟨decls⟩ `end` | ...
   ⟨body⟩ ::= ⟨exp⟩
   ⟨decls⟩ ::= ⟨decl⟩ {; ⟨decl⟩}*

   Assume that the following is the abstract syntax for the `within-use` and `decl` records. In a `within-use` record, the field `body` has type `parsed-exp` and `decls` has type `(list (decl parsed-exp))`.

```
(define-record within-use (body decls))
(define-record decl (var exp))
```

   The meaning of this syntax is supposed to be that the declarations in the list ⟨decls⟩ are sequentially processed, and then the ⟨body⟩ is evaluated in an environment that has the bindings for all the declarations. The result of the ⟨body⟩ is the result of the whole expression. *Sequential processing* for declarations means that the expression in the first declaration in the list is evaluated in the original environment, then its binding is added to the environment used to process the remaining declarations. For example, in the defined language we would have

```
within x use x = 3 end
    ==> 3
within list(b,c,d) use a = 1; b = +(a,1); c = +(b,1); d = +(c,1) end
    ==> (2 3 4)
within list(x,y) use x = 7; y = -(x,2) end
    ==> (7 5)
within begin x := +(x,y); *(x,y) end use x = 7; y = -(x,2) end
    ==> 60
let c = 2 in within list(b,c) use a = 1; b = +(c,a); c = +(b,4) end
    ==> (3 7)
```

   It follows that `within` b `use` v_1 `=` e_1`;` v_2 `=` e_2`;` ... v_n `=` e_n `end` is equivalent to the expression `let` v_1 `=` e_1 `in let` v_2 `=` e_2 `in` ... `let` v_n `=` e_n `in` b. However, you are *not* to implement this as a syntactic sugar. That is, do not use `make-let`, `make-app`, or `parse` in your solution. Instead you will implement this in `eval-exp` directly, by filling in the code for the `within-use` case of `eval-exp` on the next page.

To save time, only give the code for the `within-use` 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
```