Fall, 2000 Name: _____

22C:54, section 2 — Programming Language Concepts

Test on *EOPL* Sections 3.5–3.6, 4.5–4.6, and 5.1–5.4

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. Please put your name in the top right corner of your notes.

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 may take off a small amount if you do not give `deftype` declarations 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 features of Scheme, extensions that we discussed in class, and helping functions that you define yourself. The standard is defined by the *Revised*^{5} *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
```

This page just contains reference material for problems on later pages.
The code for the `send` procedure for OO implementations of ADTs, which we discussed in class:

```
(define send
  (lambda send-args
    (if (< (length send-args) 2)
        (error "send: too few args")
        (let ((obj (car send-args))
              (msg (cadr send-args))
              (args (cddr send-args)))
          (apply (obj msg) args)))))
```

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))

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

;; Denoted-Value ADT
number->denoted : (-> (number) Denoted-Value)
denoted->number : (-> (Denoted-Value) number)
procedure->denoted : (-> (Procedure) Denoted-Value)
denoted->procedure : (-> (Denoted-Value) Procedure)
list->denoted : (-> ((list Denoted-Value)) Denoted-Value)
denoted->list : (-> (Denoted-Value) (list Denoted-Value))

;; 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)

;;; environment ADT
the-empty-env : Environment
extend-env : (-> ((list symbol) (list Denoted-Value) Environment) Environment)
apply-env : (-> (Environment symbol) Denoted-Value)
defined-in-env? : (-> (Environment symbol) boolean)
```

1. (5 points) What is the result of the following Scheme expression?

```
(let ((my-list '(5 4)))
  (begin
    (set! my-list (cons 99 my-list))
    (cons 36 (cdr my-list))
    my-list))
```

2. (10 points) Write a object-oriented (OO) implementation of the cell abstract data type. In this version of the ADT, there is a procedure, `make-cell`, that creates objects that respond to the messages `cell-ref` and `cell-set!`. Note that the `cell-set!` message has a side effect. The protocol you are to implement is given by the following example transcript. (The code for `send` is given on page 2.)

```
> (define my-cell (make-cell 54))
> (send my-cell 'cell-ref)
54
> (send my-cell 'cell-set! 342)
> (send my-cell 'cell-ref)
342
> (define another (make-cell 99))
> (send another 'cell-ref)
99
```

Write your implementation of `make-cell`, which returns a method map that includes code for the messages `cell-ref` and `cell-set!`. Don't worry about type declarations for this problem.

3. (5 points) Assume for this problem that the defined language has been extended with the top-level form `define`, and with a primitive procedure `/` that takes two numbers and returns the result of dividing the first by the second.

   Write in the defined language (not Scheme), a procedure, `average`, that takes two numbers, `x` and `y`, as parameters, and returns their numeric mean (i.e., $\frac{x+y}{2}$).

   For example:

   ```
   average(7,5)    ==> 6
   average(0,6)    ==> 3
   average(16,200) ==> 108
   ```

   Write your answer by completing the defined language code below.

   ```
   define average =
   ```

4. (5 points) Briefly answer the following question. What changes were needed to the defined language's interpreter to support statically-scoped procedures (the `proc` special form)?

5. (10 points) Imagine you are in charge of designing and maintaining a programming language, called "Strict." You have designed "Strict" to allow users make abstract data types (ADTs) with opaque representations, so that they can enforce representation independence. But since "Strict" is designed for computation-intensive systems, "Strict" does not support first-class procedures and closures. (You are supposed to know what the these terms mean.)

   A department head comes to your office and says that code written in "Strict" is too slow. In fact, she says that the problem is due to the enforcement of representation independence. She claims that this forces client code to use the interface procedures of ADTs, which is slower than directly accessing the representation. She would like you to change "Strict" so that it does not have opaque representations and does not enforce representation independence.

   What arguments can you give this department head about the benefits of keeping opaque representations and representation independence in "Strict"?

6. In this problem you will add

   (a) (15 points) a primitive procedure `isPositive`, and

   (b) (5 points) a new built-in variable `one`,

   to an interpreter for the defined language.

   The procedure `isPositive` returns a value representing *true* if its argument is a positive number (i.e., strictly greater than 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.)

   The built-in variable `one` will have the value 1.

   For example, the code `if isPositive(one) then 54 else 342` will return 54.

   Your task is to add the primitive procedure `isPositive` and the built-in variable `one` by filling in the code for the necessary changes in the blank spaces below. You may assume the procedures for the standard domains (see page 2). You can also use the Scheme procedure `positive?` in your code. If you need any other auxiliary procedures for your definition, you must also write out those in your solution.

```
(deftype apply-prim-op (-> (symbol (list Expressed-Value)) Expressed-Value))
(define apply-prim-op
  (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)))))

(deftype prim-op-names (list symbol))
(define prim-op-names
  '(+ - * add1 sub1
   ))

(deftype init-env Environment)
(define init-env
```

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

   ⟨exp⟩ ::= `or` ⟨exp⟩ ⟨exp⟩ | ...

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

```
(define-record-type or-exp parsed-exp
  ((left-exp parsed-exp) (right-exp parsed-exp)))
(define-record or-exp (left-exp right-exp))
```

   The meaning of this syntax is supposed to be that of a short-circuit "or". The following are examples in the defined language:

```
or 1 0  ==> 1
or 0 1  ==> 1
or 1 1  ==> 1
or 0 0  ==> 0
or 1 /(4,0)  ==> 1
```

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

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

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

8. (25 points) In this problem you will implement the following syntax in the defined language.

⟨exp⟩ ::= `with` ⟨decls⟩ `do` ⟨body⟩ | ...
⟨decls⟩ ::= ⟨decl⟩ {`;` ⟨decl⟩}*
⟨body⟩ ::= ⟨exp⟩

Assume that the following is the abstract syntax for the `with-do` and `decl` records.

```
(define-record-type with-do parsed-exp
  ((decls (list (decl parsed-exp))) (body parsed-exp)))
(define-record with-do (decls body))
(define-record-type decl (decl T)
  ((var symbol) (exp T)))
(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. The following are examples in the defined language:

```
with x = 3 do x
   ==> 3
with a = 1; b = +(a,1); c = +(b,1); d = +(c,1) do list(b,c,d)
   ==> (2 3 4)
with x = 7; y = -(x,2) do list(x,y)
   ==> (7 5)
with x = 7; y = -(x,2) do *(+(x,y),y)
   ==> 60
let c = 2 in with a = 1; b = +(c,a); c = +(b,4) do list(b,c)
   ==> (3 7)
```

It follows that `with` v_1 `=` e_1`;` v_2 `=` e_2`;` ... v_n `=` e_n `do` b 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.

Please answer by filling in the code for the `with-do` case of `eval-exp` on the next page.

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

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