Com S 342
Spring 2004

Name: _____
Section: _____

Principles of Programming Languages

Exam 4 on Interpreters and Language Semantics

This test has 8 questions and pages numbered 1 through 9.

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.

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. Correct syntax also matters. Check your code over for syntax errors. You will lose points if your code has syntax errors.

You can use helping procedures whenever you like. If you write recursive helping procedures, please give a deftype declaration for them.

For Grading

Problem	Points
1	
2	
3	
4	
5	
6	
7	
8	

This page just contains reference material for problems on later pages.
Types of helpers from the chapter 3 interpreters used on this test. These ADTs correspond roughly to those in section 3.7 of the text.

```
eopl:error : (-> (symbol string datum ...) poof)

;; ---- ProcVal (procedure values) ADT --------
procval? : (type-predicate-for procval)
closure : (-> ((list-of symbol) expression environment) procval)
apply-procval : (-> (procval (list-of Expressed-Value)) Expressed-Value)

;; ---- Expressed-Value ADT --------
;; upcasts
number->expressed : (-> (number) Expressed-Value)
procval->expressed : (-> (Procval) Expressed-Value)
list->expressed : (-> ((list-of Expressed-Value)) Expressed-Value)
;; downcasts
expressed->number : (-> (Expressed-Value) number)
expressed->procval : (-> (Expressed-Value) Procval)
expressed->list : (-> (Expressed-Value) (list-of Expressed-Value))

;; ---- reference ADT --------
a-ref : (forall (T) (-> (number (vector-of T)) (ref-of T)))
deref : (forall (T) (-> ((ref-of T)) T))
setref! : (forall (T) (-> ((ref-of T) T) void))

;; ---- environment ADT ----------
;; type predicate
environment? : (type-predicate-for environment)
;; constructors
empty-env : (-> () environment)
extend-env : (-> ((list-of symbol) (list-of Expressed-Value) environment)
                 environment)
extend-env-recursively : (-> ((list-of symbol) (list-of (list-of symbol))
                              (list-of expression) environment)
                             environment)
;; observers
apply-env : (-> (environment symbol) Expressed-Value)
apply-env-ref : (-> (environment symbol) (ref-of Expressed-Value))
defined-in-env? : (-> (environment symbol) boolean)
```

1. (5 points) Below, complete the definition of the defined language interpreter's `init-env` procedure so that it defines the name `true` to be the number 1. (You don't have to worry about the values of any names other than `true`.) Once this is done, in the defined language we would have the following examples:

```
--> true
1
--> if true then 3 else 42
3
```

Your code must type check to receive full credit. Hint: look at the operations of the standard ADTs on page 2.

```
(deftype init-env (-> () environment))
(define init-env
  (lambda ()
```

2. Suppose we add `true` as a name to the initial environment, as specified in the previous problem.

   (a) (3 points) With this change, the follow expression is legal. Assuming static scoping, what is the result of this expression?

```
let true = 0
in if true then 3 else 77
```

   (b) (2 points) Briefly explain why that output occurs.

3. (5 points) Write the code to make `truth` a reserved word, so that every occurrence of the expression `truth` in an expression always denotes 1. This is done by making `truth` an expression, instead of defining it in the initial environment. The syntax changes below make it so that an expression such as `let truth = 342 in truth` is a syntax error. Your code will make the following examples work:

```
--> truth
1
--> if truth then 342 else 104
342
```

Please fill in your answer in the appropriate places below. We have already completed the concrete syntax input for SLLGEN.

```
(define the-grammar
  '((program (expression) a-program)
    ;; ... assume the other parts of the grammar are done
    (expression ("truth") truth-exp)))

(define-datatype expression expression?
   ;; ... assume the other expressions are done and add yours below ...



(deftype eval-expression (-> (expression environment) Expressed-Value))
(define eval-expression
  (lambda (exp env)
    (cases expression exp
       ;; ... assume the other expression cases are done,
       ;; and add yours below...
```

4. (5 points) This is a question about local binding and statically-scoped procedures in the defined language. In this problem, the defined language is extended with lists, so that `list(1,2,3)` returns the list `(1 2 3)`. What is the result of the following expression?

```
let a = 99
    b = 2
in let a = 700
       b = 33
       f = proc() list(a, b)
       g = proc(a) list(a, b)
       h = proc() begin set b = 87; set a = *(a, add1(b)) end
   in list(a, b, (f), let b = 5 in (g b))
```

5. (10 points) This is a question about adding a primitive to the defined language. Consider an interpreter for the defined language extended with procedures. For this interpreter your task is to add a new built-in primitive, `isProcedure`. Its semantics is that `isProcedure(`E`)` evaluates E, and returns the interpreter's representation for true just when the value of E is a procedure closure. (If E loops forever, so does the call to this primitive.) For example, once this is done, in the defined language we would have the following examples:

```
--> let add3 = proc(y) +(y, 3) in isProcedure(add3)
1
--> isProcedure(proc(x) x)
1
--> isProcedure(342)
0
```

Your code should type check, but you don't have to check for the proper number of arguments to the primitive in the defined language. Hint: look at the operations of the standard ADTs on page 2.

Please fill in your answer in the appropriate places below. We have already completed the concrete syntax input for SLLGEN.

```
(define the-grammar
  '((program (expression) a-program)
    ;; ... assume the other parts of the grammar are done
    (primitive ("isProcedure") isProcedure-prim)))

(define-datatype primitive primitive?
   ;; ... assume the other primitives are done and add yours below ...



(deftype apply-primitive
  (-> (primitive (list-of Expressed-Value)) Expressed-Value))
(define apply-primitive
  (lambda (prim args)
    (cases primitive prim
       ;; ... assume the other primitive cases are done,
       ;; and add yours below...
```

6. (10 points) This is a question about dynamic scoping (dynamic binding). Consider the following code in the defined language (As in the homework, `greater?(`v_1`,`v_2`)` is true when $v_1 > v_2$, and false otherwise, and `less?(`v_1`,`v_2`)` is true when $v_1 < v_2$, and false otherwise.)

```
let total = 0
in let overflowHandler = proc () begin set total = 999; total end
       underflowHandler = proc() begin set total = 0; total end
       ok = proc(n) begin set total = n; total end
   in let f = proc(a,b)
               let total = +(a,b)
               in if greater?(total, 999)
                  then (overflowHandler)
                  else if less?(total, 0)
                       then (underflowHandler)
                       else (ok total)
      in let x = begin (f 998 2); total end
         in let y = begin (f -(0,2000) 2); total end
            in list(x, y)
```

What is the result of the expression above in an interpreter that uses dynamic scoping?

7. This is a problem about parameter passing mechanisms. Consider the following code, written in the defined language with static scoping, assignment, and lists.

```
let q = 1
    r = 5
in let p = proc (a, b) begin
                          set b = +(r,a);
                          set q = *(a,b);
                          list(a, b, q, r)
                        end
   in let result = (p +(q, 2) r)
      in cons(q, cons(r, result))
```

(a) (10 points) What is the result of the above program if call-by-value is used as the parameter passing mechanism?

(b) (10 points) What is the result of the above program if call-by-reference is used as the parameter passing mechanism?

(c) (10 points) What is the result of the above program if call-by-value-result is used as the parameter passing mechanism? (Use left-to-right ordering if necessary.)

8. (30 points) In this problem you will implement the following syntax for an expression in the defined language.
⟨expression⟩ ::= `typecase` ⟨expression⟩ {⟨typeexp⟩ `==>` ⟨expression⟩}* `end` | ...
⟨typeexp⟩ ::= `number` | `procedure` | `list` | `any`

   Use the following as the abstract syntax for the `typecase`-expression and for ⟨typeexp⟩.

```
(define-datatype expression expression?
  ;; ...
  (typecase-exp (tested-exp expression?)
                (types (list-of typeexp?)) (bodies (list-of expression?)))
(define-datatype typeexp typeexp?
  (number-type) (procedure-type) (list-type) (any-type))
```

   To evaluate a `typecase`-expression, of the form

`typecase` E_0
  T_1 `==>` E_1 ... T_n `==>` E_n
`end`

   the interpreter first evaluates the expression E_0. Suppose this evaluation of E_0 yields a value, v_0. Then each clause of the form T_i `==>` E_i, is tested in order from left to right, and if v_0 has a type that matches T_i (but not any T_j with $j < i$), then the value of the entire expression is the value of E_i. If none of the types T_i matches the value v_0, then the result of the entire expression is 0.

   An expressed value matches the ⟨typeexp⟩ `number` if it is a number, it matches the ⟨typeexp⟩ `procedure` if it is a procedure closure, it matches the ⟨typeexp⟩ `list` if it is a list of expressed values. All expressed values match the ⟨typeexp⟩ `any`.

   Thus the following are examples in the defined language:

```
--> typecase +(5, 3)
      number ==> 7  procedure ==> 541  any ==> 342
    end
7
--> let id = proc (x) x
    in typecase (id id)
         number ==> (id 7)  procedure ==> (id 541)
         procedure ==> 88   any ==> (id 342)
       end
541
--> let myList = list(1, proc() 3, 51)
    in typecase myList
         number ==> car(myList)                  procedure ==> car(cdr(myList))
         any ==> +(291, car(cdr(cdr(myList))))  list ==> 6
       end
342
--> typecase 342 end
0
```

You should implement `typecase` in `eval-expression` directly, by filling in the code for the `typecase-exp` case of `eval-expression` below. You may also need to write one or more helping procedures. (Hint, you can use the operations on page 2 as well as Scheme's `number?` and `list?` predicates.)

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

```
(deftype eval-expression (-> (expression environment) Expressed-Value))
(define eval-expression
  (lambda (exp env)
    (cases expression exp
      (lit-exp (datum)
        (number->expressed datum))
      ;; ... assume that the rest of this is done
      (typecase-exp
```