Com S 342 Name: ____________________
Spring 2006

Principles of Programming Languages

Exam 4 on Environment-Passing Interpreters

This test has 7 questions and pages numbered 1 through 10.

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.

For Grading:

Problem	Points	Score
1	10	
2	20	
3	15	
4	12	
5	3	
6(a)	10	
6(b)	10	
6(c)	10	
7	10	

This page and the next just contain reference material for problems on later pages.
The following is the syntax of the expressions in the defined language of section 3.7, with comments on the right enclosed in "quotes" describing the abstract syntax trees.

```
⟨expression⟩ ::= ⟨number⟩                        "lit-exp (datum)"
     | ⟨identifier⟩                              "var-exp (id)"
     | ⟨primitive⟩ ( {⟨expression⟩}*(,) )        "primapp-exp (prim rands)"
     | if ⟨expression⟩                           "if-exp (test-exp
          then ⟨expression⟩                           true-exp
          else ⟨expression⟩                           false-exp)"
     | let {⟨identifier⟩ = ⟨expression⟩}*        "let-exp (ids rands
          in ⟨expression⟩                             body)"
     | proc ( {⟨identifier⟩}*(,) )               "proc-exp (ids
          ⟨expression⟩                                body)"
     | ( ⟨expression⟩ {⟨expression⟩}* )          "app-exp (rator rands)"
     | letrec                                    "letrec-exp
          {⟨identifier⟩ ( {⟨identifier⟩}*(,) )        (proc-names idss
            = ⟨expression⟩}*                           bodies
          in ⟨expression⟩                              letrec-body)"
     | set ⟨identifier⟩ = ⟨expression⟩           "varassign-exp (id rhs-exp)"
     | begin ⟨expression⟩                        "begin-exp (first
          {; ⟨expression⟩}* end                        rest)"

⟨primitive⟩ ::= + | - | * | add1 | sub1
     | equal? | cons | car | cdr | list
```

The following are the expression abstract syntax trees for the interpreter of section 3.7.

```
(define-datatype expression expression?
  (lit-exp (datum number?))
  (var-exp (id symbol?))
  (primapp-exp (prim primitive?) (rands (list-of expression?)))
  (if-exp (test-exp expression?) (true-exp expression?) (false-exp expression?))
  (let-exp (ids (list-of symbol?)) (rands (list-of expression?)) (body expression?))
  (proc-exp (ids (list-of symbol?)) (body expression?))
  (app-exp (rator expression?) (rands (list-of expression?)))
  (letrec-exp (proc-names (list-of symbol?)) (idss (list-of (list-of symbol?)))
              (bodies (list-of expression?)) (letrec-body expression?))
  (begin-exp (first expression?) (rest (list-of expression?)))
  (varassign-exp (id symbol?) (rhs-exp expression?)))
```

The following are some `define-datatype` declarations and the types of associated helpers from the chapter 3.7 interpreters that you can use in the next several problems.

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

;; ----- references -----------
(define-datatype reference reference?
  (a-ref (position integer?) (vec vector?)))

deref : (forall (T) (-> ((ref-of T)) T))
setref! : (forall (T) (-> ((ref-of T) T) void))

;; ---- environment ADT ----------
environment? : (type-predicate-for environment)
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)
apply-env : (-> (environment symbol) Expressed-Value)
apply-env-ref : (-> (environment symbol) (ref-of Expressed-Value)))
defined-in-env? : (-> (environment symbol) boolean)

;; ---- Truth Values ---------
true-value? : (-> (Expressed-Value) boolean)

;; ---- ProcVal (procedure values) --------
(define-datatype procval procval?
  (closure (ids (list-of symbol?)) (body expression?) (env environment?)))

apply-procval : (-> (procval (list-of Expressed-Value)) Expressed-Value)

;; ---- Expressed-Values --------
(define-datatype Expressed-Value expval?
  (number->expressed (num number?))
  (procval->expressed (pv procval?))
  (list->expressed (lst (list-of expval?))))

expressed->number : (-> (Expressed-Value) number)
expressed->procval : (-> (Expressed-Value) Procval)
expressed->list : (-> (Expressed-Value) (list-of Expressed-Value))
number->expressed? : (-> (Expressed-Value) boolean)
procval->expressed? : (-> (Expressed-Value) boolean)
list->expressed? : (-> (Expressed-Value) boolean)
```

1. (10 points) This is a question about adding a primitive to the defined language. Consider an interpreter for the defined language extended with procedures and lists. For this interpreter your task is to add a new built-in primitive `append`, whose syntax is shown below, with comments on the right enclosed in "quotes" describing the abstract syntax trees.

   ⟨primitive⟩ ::= ...
       | `append`   "append-prim ()"

   Its semantics is that `append(`E_1`, `E_2`)` evaluates E_1 and E_2 (in some unspecified order), and returns the interpreter's representation for the list whose elements are the elements of the value of the list E_1 followed by the elements of E_2. The following are examples:

```
--> append(list(), list())
()
--> append(list(3, 4, 2), list(2, 2, 7))
(3 4 2 2 2 7)
--> append(list(5, 3), append(list(7, 2), list(3, 5)))
(5 3 7 2 3 5)
--> append(append(list(1, 2), list(3, 4)), append(list(5, 6), list()))
(1 2 3 4 5 6)
--> append(list(list(3, 4), list(2, 1)), list(list(), list(5, 9, 3), list(2)))
((3 4) (2 1) () (5 9 3) (2))
--> (car(cdr(append(list(proc(x) x), list(proc(y) +(y,1))))) 6)
7
```

   You can assume that this primitive is given exactly two arguments, and you can let the interpreter's ADT operations give error messages if one of the arguments is not the interpreter's representation of a list (see the operations starting on page 3 above). You can use Scheme's procedure `append : (forall (T) (-> ((list-of T) ...)  (list-of T)))`.

   Please complete the code for `apply-primitive` for the case of the `append-prim` below. You don't have to change anything else in the interpreter.

```
(define-datatype primitive primitive?
   ;; ... assume the other primitives are unchanged
   (append-prim))

(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...
```

2. (20 points) In this problem you will implement two new expressions, `freeze` E and `thaw` E, whose syntax is as follows, with comments on the right enclosed in "quotes" describing the abstract syntax trees.

⟨expression⟩ ::= ...	
| `freeze` ⟨expression⟩	"freeze-exp (body)"
| `thaw` ⟨expression⟩	"thaw-exp (body)"

The expression `freeze` E creates a thunk (that is a closure with an empty formal parameter list) containing E and the current environment.

The expression `thaw` E evaluates E to obtain such a thunk, and then obtains the value of the expression inside the thunk, by running that expression in the thunk's environment. You may assume that the argument to `thaw` is a thunk (i.e., you don't have to check for that `thaw`'s argument is a thunk).

Thus, in general the value of `thaw freeze` E is the value of E.

However, note that the expression inside `freeze` E is not evaluated when `freeze` E is evaluated, but only when the resulting thunk is passed as an argument to `thaw`. Thus any side effects, infinite looping, or errors can only happen when this thunk is thawed, not when `freeze` E is evaluated.

The following are examples (in which the first doesn't give any details about the environment):

```
--> freeze 3
(closure () (lit-exp 3) "<env>")
--> thaw freeze 342
342
--> let th = freeze +(100, 242) in +(thaw th, thaw th)
684
--> let x = 6
    in let z = freeze +(x,10)
       in +(200, thaw car(list(z)))
216
--> let x = 6
    in let z = freeze +(x,10)
       in let x = 30
          in thaw z
16
--> letrec loop() = (loop)
    in let p = proc(x,ignored) x
       in let y = freeze (loop)
       in (p 178 y)
178
--> letrec loop() = (loop)
    in let frozen = list(freeze *(10,10), freeze *(5,6), freeze (loop))
       in +(thaw car(frozen), thaw car(cdr(frozen)))
130
```

Please fill in your answer on the next page.

Below complete the code for `eval-expression` for both `freeze-exp` and `thaw-exp`. You don't have to change anything else in the interpreter.

```
(define-datatype expression expression?
   ;; ... assume the other expressions are unchanged
  (freeze-exp (body expression?))
  (thaw-exp   (body expression?)) )

(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 code for freeze and thaw below...
```

3. (15 points) In this problem you will implement a pre-increment expression with the following syntax.

   ⟨expression⟩ ::= ...
       | `++` ⟨identifier⟩   “pre-inc-exp (id)”

   The expression `++`*I* adds 1 to the the value of the identifier *I*, assigns that incremented value to *I*, and also returns that value. Thus it is just like the Java or C++ pre-increment operator expression. The following are examples:

```
--> let x = 0 in ++x
1
--> let x = 0 in let y = ++x in list(x, y)
(1 1)
--> let x = 0 in let y = ++x in list(++x, ++y)
(2 2)
--> let z = 3
        i = 20
    in begin set z = +(++i, ++z); set i = ++i; list(z, i) end
(25 22)
```

   Below complete the code for `eval-expression` for `pre-inc-exp`. You don't have to change anything else in the interpreter. You can assume that the identifier being incremented holds a number.

```
(define-datatype expression expression?
   ;; ... assume the other expressions are unchanged
  (pre-inc-exp (id symbol?)) )

(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 code for pre-inc-exp below...
```

4. (12 points) This is the first of 2 problems about procedures, scoping, and recursion. Suppose we try to implement the factorial procedure in the defined language with static scoping (and call by value) as follows.

```
let fact = proc(n) if equal?(n,0) then 1 else *(n, (fact sub1(n)))
in (fact 3)
```

List the letters of all of the following that are true statements about the above defined language program. Don't include in your list any letters for statements that are false, as you will lose a point for each incorrect answer.

(a) The program terminates normally, and returns 6.
(b) The program goes into an infinite loop.
(c) The program ends with an error because `fact` is not defined.
(d) The scope of the binding of `fact` includes the body of the `let`.
(e) The scope of the binding of `fact` includes the body of the `proc` expression.
(f) The closure formed for the `proc` expression has no environment in it.
(g) The closure formed for the `proc` expression has an environment, but that environment has no binding for `fact`.
(h) The closure formed for the `proc` expression has an environment, and that environment binds `fact` to that closure.
(i) If we used dynamic scoping, then the program would return 6
(j) If we used dynamic scoping, then the program would go into an infinite loop.
(k) If we used dynamic scoping, then the program would end in an error, because `fact` is not defined.
(l) If the above program used call-by-reference, then it would return 6.

5. (3 points) Consider the following defined language program in an interpreter with static scoping.

```
let fact = 0
in begin
    set fact = proc(n) if equal?(n,0) then 1 else *(n, (fact sub1(n)));
    (fact 3)
   end
```

Which of the following is true about the above program?

(a) The program terminates normally, and returns 6.
(b) The program goes into an infinite loop.
(c) The program ends with an error because `fact` is not defined.

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

```
let a = 3
    b = 11
    c = 400
in let p = proc(w, x, y, z)
              begin
               set x = b;
               set b = +(y, z);
               list(a, b, c, w, x, y, z)
             end
   in let q = (p a a b +(c, 44))
      in cons(a, cons(b, cons(c, q)))
```

   (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? (Copy back starting at the left, using left-to-right ordering.)

7. This is a problem about call by name and call by need. Consider the following in an interpreter with static scoping.

```
let count = 0
in let g = proc(x)
             begin
               set count = add1(count);
               x
             end
   in let h = proc(y) -(*(2, y), y)
      in let res = (h (g 3))
         in +(*(100, res), count)
```

(a) (5 points) What is the result of the above program if call-by-name is used as the parameter passing mechanism?

(b) (5 points) What is the result of the above program if call-by-need is used as the parameter passing mechanism?