Com S 342
Spring 2006

Name: ______________________________

Principles of Programming Languages

Exam 3 on Scoping, Data Abstraction, and Interpreters

This test has 7 questions and pages numbered 1 through 12.

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	10	
5	10	
6	15	
7	20	

1. This is a problem about static properties of variables. Consider the following Scheme expression, in which `lambda` and `'` (sugar for `quote`) are the only special forms used. (Although perfectly formatted, it has been written with extra space between the lines, in case you want to draw arrows.)

```
((lambda (double two)

   ((lambda (up)

      ((up double double) three))

    (lambda (p f) (lambda (three) (p (f three))))))

 (lambda (x) (times x 2))

 'dos)
```

   (a) (5 points) Write, below, in set brackets, the entire set of the bound variables that occur in above expression. For example, write $\{v, w\}$, if the bound variables that occur are v and w. If there are no bound variables used, write $\{\}$. (You're supposed to know what a "bound variable" is.)

   (b) (5 points) Write, below, in set brackets, the entire set of the free variables that occur in above expression. For example, write $\{v, w\}$, if the free variables used are v and w. If there are no free variables that occur, write $\{\}$. (You're supposed to know what a "free variable" is.)

2. (20 points) This problem is about transforming procedural to abstract syntax tree representations of abstract data types.

   Abstractly, a directed graph (or digraph) can be thought of as a mapping from a two nodes to a boolean, where the boolean is true just when there is an edge from the first node to the second. Consider the following procedural representation of `digraph`.

```
(module digraph-as-proc  (lib "typedscm.ss" "typedscm")


  (provide empty-digraph add-edge adjacent-to?)

  (deftype empty-digraph (-> () digraph))
  (deftype add-edge
    (-> (digraph node node) digraph))
  (deftype adjacent-to? (-> (digraph node node) boolean))

  (require "node-mod.scm")  ;; provides node-equal? and node?

  (defrep (digraph (-> (node node) boolean)))

  (define empty-digraph
    (lambda ()
      (lambda (from to) #f)))

  (define add-edge
    (lambda (old-graph new-from new-to)
      (lambda (from to)
        (or (and (node-equal? new-from from)
                 (node-equal? new-to to))
            (adjacent-to? old-graph from to)))))

  (define adjacent-to?
    (lambda (graph from to)
      (graph from to)))

) ;; end module
```

   The above code uses a module `node-mod` which defines the following operations on the type node.

```
node-equal? : (-> (node node) boolean)
node? : (type-predicate-for node)
a-node : (-> (number) node)
```

   For purposes of examples, we assume that the only information in a node is a number.

   There are some examples, which you can ignore, on the next page. The area to write your answer is on the page after that.

Examples aren't really going to help you (so please skip this page), but if we assume that the types of the node operations are as above and if we make the following definitions

```
(define dag1 (add-edge (empty-digraph) (a-node 1) (a-node 2)))
(define dag2 (add-edge dag1 (a-node 2) (a-node 3)))
(define dag3 (add-edge dag2 (a-node 3) (a-node 1)))
```

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

```
(adjacent-to? (empty-digraph) (a-node 342) (a-node 541)) ==> #f
(adjacent-to? dag1 (a-node 342) (a-node 541)) ==> #f
(adjacent-to? dag1 (a-node 1) (a-node 1)) ==> #f
(adjacent-to? dag1 (a-node 1) (a-node 2)) ==> #t
(adjacent-to? dag1 (a-node 2) (a-node 1)) ==> #f
(adjacent-to? dag2 (a-node 2) (a-node 3)) ==> #t
(adjacent-to? dag3 (a-node 3) (a-node 1)) ==> #t
(adjacent-to? dag3 (a-node 2) (a-node 3)) ==> #t
(adjacent-to? dag3 (a-node 1) (a-node 2)) ==> #t
(adjacent-to? dag3 (a-node 3) (a-node 2)) ==> #f
```

Please go to the next page for the place to write your answer.

Your task is to transform the procedural representation given above into one that uses abstract syntax trees. Do this writing a `define-datatype` declaration and the bodies of the procedures in the spaces provided on the this page.

```
(module digraph-as-ast  (lib "typedscm.ss" "typedscm")

  (provide empty-digraph add-edge adjacent-to?)

  (deftype empty-digraph (-> () digraph))
  (deftype add-edge
    (-> (digraph node node) digraph))
  (deftype adjacent-to? (-> (digraph node node) boolean))

  (require "node-mod.scm")  ;; provides node-equal? and node?

  ;; Write the define-datatype declarations below









  ;; Now write any other needed code below.
















) ;; end module
```

3. This is a question about scope rules. Consider the following code in the defined language extended with the `list` primitive, so that `list(8, 9, 10)` returns `(8 9 10)`.

```
let t = 3
    w = 2
in let d = proc (x) list(t, w, x)
       t = 30
       w = 200
   in (d t)
```

   (a) (7 points) What is the result of the expression above in an interpreter that uses static scoping?

   (b) (8 points) What is the result of the expression above in an interpreter that uses dynamic scoping?

This page just contains reference material for problems on later pages.

The following are the expression abstract syntax trees for the interpreter of section 3.6.

```
(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?)))
  (begin-exp (first expression?) (rest (list-of expression?)))
  (letrec-exp (proc-names (list-of symbol?)) (idss (list-of (list-of symbol?)))
              (bodies (list-of expression?)) (letrec-body expression?)))
```

The following are the types of the helpers from the chapter 3 interpreters that you may assume on this test. These ADTs correspond to those in section 3.6 of the text.

```
eopl:error : (-> (symbol string datum ...) poof)
;; ---- 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)
defined-in-env? : (-> (environment symbol) boolean)
;; ---- 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)
;; ---- Truth Values ---------
true-value? : (-> (Expressed-Value) boolean)
;; ---- Expressed-Value ADT --------
number->expressed : (-> (number) Expressed-Value)
procval->expressed : (-> (Procval) Expressed-Value)
list->expressed : (-> ((list-of Expressed-Value)) Expressed-Value)
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)
```

4. (10 points) Below, complete the definition of the defined language interpreter's `init-env` procedure so that it defines the name `id` to be the procedure that returns its argument (i.e., the value of the defined language expression `proc (x) x`). (You don't have to worry about the values of any names other than `id`.) Once this is done, in the defined language we would have the following examples:

   ```
   --> (id 10)
   10
   --> +(3, ((id id) 5))
   8
   --> (id (id (id 10)))
   10
   ```

   Hint: look at the operations of the standard ADTs on page 7 and use an empty environment for the closure's environment.

   ```
   (deftype init-env (-> () environment))
   (define init-env
     (lambda ()
       ;; fill in your answer below...
   ```

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 and lists. For this interpreter your task is to add a new built-in primitive, `member?`. Its semantics is that `member?(`E_1`,` E_2`)` evaluates E_1 and E_2 (in some unspecified order), and returns the interpreter's representation for true just when the value of E_1 is an element in the list E_2. For example, the following are examples:

```
--> member?(2, list())
0
--> member?(2, list(2, 2, 7))
1
--> member?(3, list(3, 4, 2, 1))
1
--> member?(list(3), list(list(3)))
1
--> member?(list(5, 3), list(list(7, 2), list(3, 5)))
0
```

You don't have to check for the proper number of arguments to the primitive or any condition on the arguments (i.e., you can let Scheme handle any errors). Hint: look at the operations of the standard ADTs on page 7. You can use Scheme's procedure `member` : `(forall (T) (-> (T (list-of T)) boolean))`.

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

```
(define-datatype primitive primitive?
   ;; ... assume the other primitives are unchanged
   (member?-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...
```

6. (15 points) In this problem you will implement a new kind of conditional expression, the "`all`" expression. This has the following syntax.

   $\langle$expression$\rangle$ ::= `all` $\{\langle$expression$\rangle\}^*$ `then` $\langle$expression$\rangle$ `else` $\langle$expression$\rangle$
   $\quad\quad\quad\quad\quad\quad\quad\mid \ldots$

   To save time, you don't have to change the grammar to parse the above concrete syntax. We will use the following as the abstract syntax.

```
(define-datatype expression expression?
  ;;  ... rest of the abstract syntax is unchanged
  (all-exp
   (test-exps (list-of expression?))
   (true-exp expression?)
   (false-exp expression?))
 )
```

   You only have to write the code in `eval-expression` (and any helping procedures you desire) to evaluate the `all` expression. You should do this *without* creating new abstract syntax trees (i.e., use a direct translation instead of considering this to be a desugaring problem).

   The value of an all-exp of the form `all` $E_1 \cdots E_n$ `then` E_t `else` E_f is the value of E_t if all of the $E_1 \cdots E_n$ evaluate to true, and otherwise is the value of E_f. The $E_1, \ldots, E_n$ are evaluated left to right until one of them is found to be false. That is, if for some $i \in \{1, \ldots, n\}$, one of the E_i has a value that represents false in the defined language, then all of the E_j with $j > i$ are left unevaluated, and the value of E_f is the value of the entire expression.

   The following are examples of the `all` expression in the defined language.

```
--> all then 32 else 45
32
--> all 1 1 then 5 else 7
5
--> all 1 0 then 5 else 7
7
--> let x = 10
    in all equal?(x,10) less?(x,15) then 7 else 2
7
--> letrec loop() = (loop)
    in +(3, all 0 (loop) then (loop) else 7)
10
```

   Hint: you can also use the operations of the standard ADTs in the interpreter, whose types are given on page 7.

Please write your answer 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...
```

7. (20 points) Several functional languages, such as Haskell and Scala, treat procedures as if they were implicitly curried. In such a language, a procedure may be applied with fewer actual parameters than the number of declared formal parameters. If a closure is applied with fewer actual parameters than its expected number of formal parameters, then the application returns a closure that is prepared to accept the remaining parameters. However, if a closure is applied with the expected number of arguments, it runs its body as usual.

   For example, in the defined language, with lists, we would have the following.

```
--> let add = proc(x,y) +(x,y)
    in let add3 = (add 3)
       in (add3 5)
8
--> let listem = proc(a,b,c,d,e,f) list(a,b,c,d,e,f)
    in list((((listem 3) 4 2) 9 10 11), ((((listem) 1 2 3 4) 5) 6))
((3 4 2 9 10 11) (1 2 3 4 5 6))
```

   In this problem you will change the interpreter for the defined language so that its procedures are implicitly curried as described above. To do this you will modify the code for `apply-procval`.

   You can assume that closures are never applied to more actual parameters than the expected number of formal parameters; that is, you don't have do check for errors. Hint: you can use Scheme's procedure `list-tail`: `(forall (T) (-> ((list-of T) number) (list-of T)))`, which is such that `(list-tail '(a b c d e) 3)` returns `(d e)`.

```
(define apply-procval
  (lambda (proc args)
    (cases procval proc
      (closure (ids body env)
          ;; put your code below...
```