Com S 342 Name: ______________________
Fall 2006

Principles of Programming Languages

Exam 3 on Scoping, Data Abstraction, and Interpreters

This test has 7 questions and pages numbered 1 through 13.

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 (planner)

   ((lambda (drawing goal)

       (planner drawing goal))

    (build-drawing outline (lambda (room z) (add-chair room 'brown)) room)))

 (lambda (x y) (find-goal x y)))
```

   (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, the store type considered in this problem can be thought of as a mapping from a number (a location) to a symbol (a value). Consider the following procedural representation of `store`.

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

  (provide empty-store update lookup)

  (deftype empty-store (-> () store))
  (deftype update (-> (store number symbol) store))
  (deftype lookup (-> (store number) symbol))

  (defrep (store (-> (number) symbol)))

  (define empty-store
    (lambda ()
      (lambda (loc) 'undefined)))

  (define update
    (lambda (old-store new-loc new-val)
      (lambda (loc)
        (if (equal? new-loc loc)
            new-val
            (lookup old-store loc)))))

  (define lookup
    (lambda (s loc)
      (s loc)))

) ;; end module
```

   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 s1 (update (update (empty-store) 0 'zero-val) 1 'all))
(define s2 (update s1 2 'both))
(define s3 (update s2 0 'none))
```

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

```
(lookup (empty-store) 0) ==> undefined
(lookup s1 0) ==> zero-val
(lookup s1 1) ==> all
(lookup s1 2) ==> undefined
(lookup s2 0) ==> zero-val
(lookup s2 1) ==> all
(lookup s2 2) ==> both
(lookup s2 3) ==> undefined
(lookup s3 0) ==> none
(lookup s3 1) ==> all
(lookup s3 2) ==> both
```

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 store-as-ast (lib "typedscm.ss" "typedscm")

  (provide empty-store update lookup)

  (deftype empty-store (-> () store))
  (deftype update (-> (store number symbol) store))
  (deftype lookup (-> (store number) symbol))

  ;; 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(3, 4, 2)` returns `(3 4 2)`, and with a `less?` primitive, so that `less?(3,44)` returns 1 and `less?(44,3)` returns 0.

```
let retired = 65
    younger = 15
in let isSenior? = proc (age) less?(retired,age)
   in let p = proc (younger) list(younger, retired, (isSenior? younger))
      in let retired = 9
         in (p younger)
```

   (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 `ignore` to be the procedure that pays no attention to its argument and instead returns 0 (i.e., the value of the defined language expression `proc (x) 0`). (You don't have to worry about the values of any names other than `ignore`.) Once this is done, in the defined language we would have the following examples:

```
--> (ignore 10)
0
--> +(3, (ignore 5))
3
--> (ignore (ignore 11))
0
```

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 an `equal?` primitive. For this interpreter your task is to add a new built-in primitive, `not?`. Its semantics is that `not?(`E`)` evaluates E, and returns the interpreter's representation for true just when the value of E is the interpreter's representation for false, and vice versa. The following are examples:

```
--> not?(0)
1
--> not?(1)
0
--> not?(+(1,341))
0
--> not?(equal?(3,4))
1
--> not?(equal?(3,3))
0
```

You don't have to check for the proper number of arguments to the primitive or any condition on the arguments (such as that it's a number, 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 `not : (-> (boolean) boolean)`.

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

```
(define-datatype primitive primitive?
   ;; ... assume the other primitives are unchanged
   (not?-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 "`find`" expression. This has the following syntax.

   ⟨expression⟩ ::= `find` ⟨expression⟩ `in` ⟨expression⟩ `then` ⟨expression⟩ `else` ⟨expression⟩
   | ...

   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
  (find-exp
   (sought-exp expression?)
   (list-exp expression?)
   (found-exp expression?)
   (not-found-exp expression?))
 )
```

   You only have to write the code in `eval-expression` (and any helping procedures you desire) to evaluate the `find` 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 a find-exp of the form `find` E_1 `in` E_2 `then` E_3 `else` E_4 is the value of E_3 if the value of E_1 is a member of the list that is the value of E_2, and otherwise is the value of E_4. You may assume that E_2 evaluates to a list, that is you don't have to do any special checking to prevent an error if E_2 does not evaluate to a list.

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

```
--> find 1 in list(1, 2, 3) then 32 else 45
32
--> find 1 in list(3, 2, 3) then 32 else 45
45
--> (find 72 in list(72, 72, 3) then proc (x) 0 else proc (y) 1  7)
0
--> let x = 10
        y = list(8, 9, 12, 78, 22, 5, 9, 8, 13, 55, 78)
    in find +(x,3) in y then list() else list(3)
()
--> let x = 10
        y = list(8, 9, 12, 78, 22, 5, 9, 8, 13, 55, 78)
    in find -(x,0) in y then list() else list(3)
(3)
```

   Hint: you can also use the operations of the standard ADTs in the interpreter, whose types are given on page 7. You can also use Scheme's procedure `member` which has type `(forall (T) (-> (T (list-of T)) boolean))`.

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) In this problem you will add to the interpreter a variation on the statically-scoped `let` expression, a `do-where` expression that can declare procedures used in its body:

   $\langle\text{expression}\rangle$ ::= `do` $\langle\text{expression}\rangle$
   `where` $\{\langle\text{identifier}\rangle$ `(` $\{\langle\text{identifier}\rangle\}^{*(,)}$ `)` `=` $\langle\text{expression}\rangle\}^{*}$ `end`
   | ...

   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
  (dowhere-exp
   (body expression?)
   (proc-names (list-of symbol?))
   (idss (list-of (list-of symbol?)))
   (bodies (list-of expression?)))
 )
```

   When a `do-where` expression is executed, it first makes closures for each procedure, using the i^{th} list in `idss`, the i^{th} body from the list `bodies`, and the surrounding environment to make the i^{th} closure. Then it executes the body of the `do-where` expression in an environment where the i^{th} `proc-name` is bound to the i^{th} closure. Note that the resulting procedures are not recursive. For example, in the defined language, with lists, we would have the following.

```
--> do (add3 7) where add3(y) = +(3,y) end
10
--> do do +(3, (add3 5)) where add3(y) = (add 3 y) end
    where add(x,y) = +(x,y)
    end
11
--> do do (runIt) where add3(y) = +(y, 7)
                        runIt() = (add3 6)
                  end
    where add3(y) = +(y, 3)
    end
9
--> do list((a 10), (b 10), (c 10), (d 10))
    where a(x) = +(x,0)
          b(x) = +(x,1)
          c(x) = +(x,2)
          d(x) = do (e x) where e(x) = +(x,3) end
    end
(10 11 12 13)
```

   In this problem you will implement the `do-where` expression directly, without making new abstract syntax trees. For example, don't use `let-exp` or `proc-exp` in your solution. There is space for your answer on the next page.

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...
```