Com S 342 Name: _____
Spring 2005 TA (or Section): _____

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	25	
3	5	
4	5	
5	10	
6	15	
7	30	

1. This is a problem about static properties of variables. Consider the following Scheme expression, in which `lambda` and `quote` are the only special forms used. (Although perfectly formatted, it has been written with extra space between the lines to aid in the drawings required below.)

```
((lambda (proc)

  ((lambda (mapper items) (mapper proc items))

   (lambda (f ls) (map (lambda (x) (list (proc x))) ls))

   (list (quote item1) (quote item2))))

 proc)
```

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

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

2. (25 points) This problem is about transforming procedural to abstract syntax tree representations.

   Abstractly, an exception handler, h, is a function that takes a symbol, `exception-sym`, a number `exception-val`, and a procedure, `otherwise-proc`, and that returns a number. It can produce this number either by computing some function of `exception-val`, or by calling `otherwise-proc` with the arguments `exception-sym` and `exception-val`. The following is a procedural representation of the type `handler`.

```
(module handler-as-proc (lib "typedscm.ss" "lib342")
  (provide catch composite handle handler?)

  (deftype handler? (type-predicate-for handler))
  (deftype catch (-> ((-> (symbol) boolean) (-> (number) number)) handler))
  (deftype composite (-> (handler handler) handler))
  (deftype handle (-> (symbol number handler (-> (symbol number) number))
                      number))

  (defrep (handler (-> (symbol number (-> (symbol number) number)) number)))

  (define handler?
    (lambda (x)
      (procedure? x)))

  (define catch
    (lambda (applies? body-proc)
      (lambda (exception-sym exception-value otherwise-proc)
        (if (applies? exception-sym)
            (body-proc exception-value)
            (otherwise-proc exception-sym exception-value)))))

  (define composite
    (lambda (h1 h2)
      (lambda (exception-sym exception-value otherwise-proc)
        (handle exception-sym
                exception-value
                h1
                (lambda (sym val)
                  (handle sym val h2 otherwise-proc))))))

  (define handle
    (lambda (exception-sym exception-value h otherwise-proc)
      (h exception-sym exception-value otherwise-proc)))
) ;; end module
```

   Your task is to transform the procedural representation above into one that uses abstract syntax trees. Do this by giving the `define-datatype` declaration needed and the bodies of the necessary procedures in the space provided below.

Examples aren't really going to help you on this problem (so feel free to ignore this page!), but if we define

```
(deftype check-sym (-> (symbol) (-> (symbol) boolean)))
(define check-sym
  (lambda (to-check-for)
    (lambda (exception-sym) (eq? to-check-for exception-sym))))

(deftype catch-io-0 handler)
(define catch-io-0
  (catch (check-sym 'io) (lambda (n) 0)))
(define catch-nullderef-1
  (catch (check-sym 'nullderef) (lambda (n) 1)))
(define not-me
  (catch (lambda (x) #f) (lambda (n) 2)))
(define everything
  (catch (lambda (x) #t) (lambda (n) n)))
```

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

```
(handle 'io 17 catch-io-0 (lambda (sym n) 3)) ==> 0
(handle 'whatever 17 catch-io-0 (lambda (sym n) 3)) ==> 3
(handle 'nullderef 0 catch-nullderef-1 (lambda (sym n) 3)) ==> 1
(handle 'whatever 0 catch-nullderef-1 (lambda (sym n) 55)) ==> 55
(handle 'problem 0 not-me (lambda (sym n) 77)) ==> 77
(handle 'whatever 0 not-me (lambda (sym n) 77)) ==> 77
(handle 'problem 0 everything (lambda (sym n) 88)) ==> 0
(handle 'whatever 5 everything (lambda (sym n) 88)) ==> 5
(handle 'problem
        22
        (catch (check-sym 'problem) (lambda (n) n))
        (lambda (sym n) 541))
   ==> 22
(handle 'io 2 (composite catch-io-0 everything) (lambda (sym n) 3)) ==> 0
(handle 'io 2 (composite everything catch-io-0) (lambda (sym n) 3)) ==> 2
(handle 'io 2 (composite not-me catch-io-0) (lambda (sym n) 3)) ==> 0
(handle 'io 2 (composite catch-io-0 not-me) (lambda (sym n) 3)) ==> 0
(handle 'problem
        85
        (composite (catch (check-sym 'problem) (lambda (n) -2))
                   (composite
                    not-me
                    (catch (check-sym 'problem) (lambda (n) -3))))
        (lambda (sym n) 3))
   ==> -2
```

Please write your solution below. Hint: The type predicate for the type of procedures that take symbols and return booleans is written `(-> (symbol?) boolean?)`, and the type predicate for the type of procedures that take numbers and return numbers is written `(-> (number?) number?)`.

```
(module handler-as-ast (lib "typedscm.ss" "lib342")
  (provide catch composite handle handler?)

  (deftype catch (-> ((-> (symbol) boolean) (-> (number) number)) handler))
  (deftype composite (-> (handler handler) handler))
  (deftype handle (-> (symbol number handler (-> (symbol number) number))
                      number))

  ;; Fill in the code for the rest of this module below
  (define-datatype
```

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))
```

3. (5 points) Below, complete the definition of the defined language interpreter's `init-env` procedure so that it defines the name `ten` to be the number 10. (You don't have to worry about the values of any names other than `ten`.) Once this is done, in the defined language we would have the following examples:

```
--> ten
10
```

Your code must type check to receive full credit. Hint: look at the operations of the standard ADTs on page 6.

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

4. (a) (2 points) Would the interpreter for the defined language be able to correctly implement statically-scoped `let` expressions if `eval-expression` did *not* pass the environment to recursive calls? (Please answer "yes" or "no.")

   (b) (3 points) Briefly explain your answer.

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, `expt`. Its semantics is that `expt(`E_1`,` E_2`)` evaluates E_1 and E_2 (in some unspecified order), and returns $E_1^{E_2}$, that is, the value of E_1 to the power E_2. You can use the Scheme `expt` procedure in your implementation. For example, once this is done, in the defined language we would have the following examples:

```
--> expt(2, 3)
8
--> expt(3, 4)
81
--> expt(6, 0)
1
```

Your code should type check, but you don't have to check for the proper number of arguments to the primitive or any condition on the arguments (e.g., let Scheme handle any errors). Hint: look at the operations of the standard ADTs on page 6.

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 ("expt") expt-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. 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 x = 0
    y = 44
    g = proc() 25
in  let callsg = proc(y) (g)
    in let y = 7
           g = proc() list(x, y)
       in let x = 1
          in list(x, y, (g), (callsg let y = 3 in +(y,y)))
```

   (a) (5 points) What is the result of the expression above in an interpreter that uses static scoping?

   (b) (10 points) What is the result of the expression above in an interpreter that uses dynamic scoping?

7. (30 points) In this problem you will implement statically-scoped condition procedures with the following new syntax for expressions in the defined language.

   ⟨expression⟩ ::= `raise` ⟨identifier⟩ `(` {⟨expression⟩}$^{*(,)}$ `)`
   | `within` ⟨expression⟩ `use` ⟨identifier⟩ `(` {⟨identifier⟩}$^{*(,)}$ `)` ⟨expression⟩ `end`
   | ...

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

```
(define-datatype expression expression?
  ;; ...
  (raise-exp (condition-name symbol?)
             (rands (list-of expression?)))
  (within-exp (body expression?)
              (condition-name symbol?)
              (ids (list-of symbol?))
              (use-body expression?)))
```

   You only have to write the code in `eval-expression` (and any helping procedures you desire) to evaluate the `raise` and `within` expressions. You must 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 expression of the form `raise` I `(`E_1`,...,`E_n`)` is found by evaluating the actual arguments, $E_1, \ldots, E_n$, and then calling the procedure bound to I in the current environment with the values of the actual arguments. It is an error if I is not bound to a procedure in the current environment, and your code should check for it not being bound and give the error message `Unhandled raise` if there I is not defined in the current environment. It is also an error if the value of I in the current environment is not a ProcVal, and your code must check for this and give the error message `Attempt to raise to non-procedure` if it occurs. Your code does not have to check that the right number of arguments are passed to the procedure.

   The value of a `within`-expression of the form

   `within` E `use` I $(I_1, \ldots, I_n)$ E_1 `end`

   in an environment *env* is the value of E in an environment *env*′ that extends *env* with a binding from the identifier I to a procedure value; the procedure value is a closure with formal parameter names $I_1, \ldots, I_n$, with body expression E_1, and with remembered environment *env*.

   There are examples on the next page, and there is space for your answer on the page after that.

The following are examples of these expressions in the defined language.

```
--> within +(5,3) use io(n) n end
8
--> within raise io(+(2,3)) use io (n) n end
5
--> within +(9, raise exc(4,5)) use exc(m,n) +(+(m,6), *(2,n)) end
29
--> within
       within +(9, raise trouble(4)) use io(n) 0 end
    use trouble (m) +(m,6) end
19
--> within
       within +(9, raise trouble(4)) use trouble(n) 0 end
    use trouble (m) +(m,6) end
9
--> within let p = proc(x) let y = +(x, raise myex(3)) in +(100, y)
           in (p 4)
    use myex(k) +(k, 1000) end
1107
--> within
       within +(raise io(3), raise trouble(4)) use io(n) 100 end
    use trouble (m) +(m,6) end
110
--> within
      within +(10, raise problem(3, raise io(4),
                                    within raise foo(3) use foo(n)n end))
      use io(n) n end
    use problem(i, j, k) +(+(*(10000,i), *(100, j)), k) end
30413
--> let x = 10
    in within raise oops(x)
       use oops(y) +(y, x) end
20
--> raise io(4)
Error reported by eval-expression:
Unhandled raise!
--> let z = 5 in raise z(3)
Error reported by eval-expression:
Attempt to raise to non-procedure 5
```

You should implement the `raise` and `within` expressions by filling in the code for them on the next page. For maximum credit, your code should type check.

To save time, only give the code for the `raise` and `within` expression cases of `eval-expression`, and any auxiliary procedures that you call in these cases.

Hint: you can use the operations of the standard ADTs in the interpreter, whose types are given on page 6.

```
(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,
      ;; write your code for raise and within below
```