Com S 342
Spring 2004

Name: ______________________
TA (or Section): ______________________

Principles of Programming Languages

Exam 3 on Data Abstraction

This test has 4 questions and pages numbered 1 through 7.

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	

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

```
((lambda (disk)

   ((lambda (files ignored)

       (get-contents files my-file))

    (map (lambda (sector) (read sector)) (contents-of disk)) disk))

 (drive my-file))
```

   (a) (5 points) In the above expression, draw arrows from each bound variable use to the corresponding formal parameter declaration. (You're supposed to know what a "bound variable use" is.) Don't draw any other arrows.

   (b) (5 points) In the above expression, also draw contours showing the regions of the declarations.

   (c) (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.)

   (d) (10 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.)

   (e) (15 points) Give the lexical address form of the above expression by filling in the rest of the expression below. (You can choose the position numbers for the free variables arbitrarily.)

```
((lambda (disk)
   ((lambda (files ignored)
```

2. (10 points) This is a problem about abstract data types. Suppose we represent the abstract type `text-file` using the concrete type `(list-of string)`. The text files are represented such that the i^{th} string in the list represents the i^{th} line in the file. For example, the list `(list "a file with" "two lines")` represents a file with two lines, the first of which is the string `"a file with"` and the second of which is the string `"two lines"`. So, in this course's type system, we write:

```
(defrep text-file (list-of string))
```

Using this representation, implement the following two procedures with the following abstract types:

```
(deftype empty? (-> (text-file) boolean))
(deftype read-line (-> (text-file number) string))
```

The `empty?` procedure takes an argument `tf`, and returns `#t` just when `tf` has no lines, and otherwise returns `#f`. The `read-line` procedure takes a text-file `tf` and a number `line-num`, and returns the line of `tf` that is `line-num` lines from the beginning of the file (thus the first is numbered 0). You can use Scheme's built-in procedure `list-ref` in your implementation of `read-line`. For example, if `not-empty` is a text-file whose 0-th line is `"my line 0"`, then

```
(empty? not-empty) ==> #f
(read-line not-empty 0) ==> "my line 0"
```

3. (25 points) Consider the following abstract syntax for a multiple argument lambda calculus containing '`freeze`' and '`thaw`' expressions.

```
(define-datatype expression expression?
   (var-exp (id symbol?))
   (lambda-exp (ids (list-of symbol?)) (body expression?))
   (app-exp (rator expression?) (rands (list-of expression?)))
   (freeze-exp (body expression?))
   (thaw-exp (body expression?)) )
```

Your task is to write a procedure,

```
desugar : (-> (expression) expression)
```

that takes an expression, `exp`, and returns an expression that is just like `exp`, except that all, for all subexpressions e of `exp`:

ASTs of form	`(freeze-exp` e`)`	are transformed into	`(lambda-exp () (desugar` e`))`
and ASTs of form	`(thaw-exp` e`)`	are transformed into	`(app-exp (desugar` e`) ())`

That is, no `freeze-exp` or `thaw-exp` trees remain in the result, but otherwise the expression is unchanged. There are several examples on the next page.

The following are examples for the problem on the previous page.

```
(desugar (var-exp 'y))
  = (var-exp 'y)

(desugar (lambda-exp (list 'x) (var-exp 'x)))
  = (lambda-exp (list 'x) (var-exp 'x))

(desugar (lambda-exp (list 'x) (freeze-exp (var-exp 'x))))
  = (lambda-exp (list 'x) (lambda-exp '() (var-exp 'x)))

(desugar (app-exp (var-exp 'f) (list (var-exp 'x) (var-exp 'y))))
  = (app-exp (var-exp 'f) (list (var-exp 'x) (var-exp 'y)))

(desugar (freeze-exp (var-exp 'body)))
  = (lambda-exp '() (var-exp 'body))

(desugar (freeze-exp (lambda-exp (list 'z) (var-exp 'z))))
  = (lambda-exp '() (lambda-exp (list 'z) (var-exp 'z)))

(desugar (freeze-exp (lambda-exp (list 'q 'z) (freeze-exp (var-exp 'z)))))
  = (lambda-exp '() (lambda-exp (list 'q 'z) (lambda-exp () (var-exp z))))

(desugar (thaw-exp (freeze-exp (var-exp 'body))))
  = (app-exp (lambda-exp '() (var-exp 'body)) '())

(desugar (thaw-exp (var-exp 'thunk)))
  = (app-exp (var-exp 'thunk) '())

(desugar (app-exp (lambda-exp (list 'x)
                     (freeze-exp (thaw-exp
                                   (freeze-exp (var-exp 'body)))))
                  (list (var-exp 'arg))))
  = (app-exp (lambda-exp (list 'x)
                     (lambda-exp '() (app-exp
                                         (lambda-exp '() (var-exp 'body)) '())))
             (list (var-exp 'arg)))
```

4. (25 points) This problem is about transforming procedural to abstract syntax tree representations.

   Abstractly, a sparse matrix is a mapping from a pair of numbers to another number. The pair of numbers index elements of the the matrix; that is, they give the row and column of the element. The resulting number is the element of the matrix. A matrix is *sparse* if most of its elements are zero (0.0). Consider the following procedural representation of `sparse-matrix`.

```
(defrep sparse-matrix (-> (number number) number))

(deftype empty-sparse-matrix (-> () sparse-matrix))
(define empty-sparse-matrix
  (lambda ()
    (lambda (i j) 0.0)))

(deftype assign-element
  (-> (sparse-matrix number number number) sparse-matrix))
(define assign-element
  (lambda (old-mat new-i new-j new-val)
    (lambda (i j)
      (if (and (= new-i i) (= new-j j))
          new-val
          (fetch-element old-mat i j)))))

(deftype fetch-element (-> (sparse-matrix number number) number))
(define fetch-element
  (lambda (mat i j)
    (mat i j)))
```

   Examples aren't really going to help you, but if we define

```
(define mat1 (assign-element (empty-sparse-matrix) 1 1 1.14))
(define mat2 (assign-element mat1 2 2 2.73))
(define mat3 (assign-element mat2 1 1 3.14159))
```

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

```
(fetch-element (empty-sparse-matrix) 342 541) ==> 0.0
(fetch-element mat1 342 541) ==> 0.0
(fetch-element mat1 1 1) ==> 1.14
(fetch-element mat1 -3 -3) ==> 0.0
(fetch-element mat2 2 2) ==> 2.73
(fetch-element mat3 1 1) ==> 3.14159
(fetch-element mat3 32234211 99324324) ==> 0.0
(fetch-element mat3 2 2) ==> 2.73
```

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

```
;;; Write the define-datatype declarations below
```

```
;;; Now fill in the code for the operations below
;;; (or explicitly say which parts are not needed if that's so)

(deftype empty-sparse-matrix (-> () sparse-matrix))
```

```
(deftype assign-element
  (-> (sparse-matrix number number number) sparse-matrix))
```

```
(deftype fetch-element (-> (sparse-matrix number number) number))
```