Com S 342 Name: _____
Fall 2001 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.

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

```
((lambda (patient rooms)

   ((lambda (families)

      (find families rooms))

    (map (lambda (r) (second r)) (rest rooms))))

 patient (second db))
```

   (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 (patient rooms)
   ((lambda (families)
```

2. (10 points) This is a problem about abstract data types. Suppose we represent the abstract type `color` using the concrete type `number`. That is, in this course's type system, we write:

```
(defrep color number)
```

We will represent the color 'red' using the number 1. Using this representation, implement the following two procedures with the following abstract types:

```
(deftype red (-> () color))
(deftype red? (-> (color) boolean))
```

For example, if `not-red` is a color that is not red, then

```
(red? (red)) ==> #t
(red? not-red) ==> #f
```

3. (25 points) Consider the following abstract syntax for a multiple argument lambda calculus with '`bind`' 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?)))
   (bind-exp (ids (list-of symbol?))
             (bindings (list-of expression?))
             (body expression?)))
```

Your task is to write a procedure `desugar`

```
(deftype desugar (-> (expression) expression))
```

that takes an expression, `exp`, and returns an expression that is just like `exp`, except that all abstract syntax trees of the form

(`bind-exp (list` $v_1 \ldots v_n$`) (list` $e_1 \ldots e_n$`)` e_0`)`

are transformed into the form

(`app-exp (lambda-exp (list` $v_1 \ldots v_n$`) (desugar` e_0`))`
`(map desugar (list` $e_1 \ldots e_n$`))),`

so that no `bind-exp` trees remain in the result. 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 (app-exp (var-exp 'f) (list (var-exp 'x))))
  = (app-exp (var-exp 'f) (list (var-exp 'x)))

(desugar (bind-exp (list 'v) (list (var-exp 'e)) (var-exp 'body)))
  = (app-exp (lambda-exp (list 'v) (var-exp 'body)) (list (var-exp 'e)))

(desugar (app-exp (bind-exp (list 'v) (list (var-exp 'e))
                            (var-exp 'body))
                  (list (var-exp 'q)
                        (bind-exp (list 'id)
                                  (list (lambda-exp (list 'z)
                                                    (var-exp 'z)))
                                  (app-exp (var-exp 'q)
                                           (list (var-exp 'v)))))))
  = (app-exp (app-exp (lambda-exp (list 'v) (var-exp 'body))
                      (list (var-exp 'e)))
             (list (var-exp 'q)
                   (app-exp (lambda-exp (list 'id)
                                        (app-exp (var-exp 'q)
                                                 (list (var-exp 'v))))
                            (list (lambda-exp (list 'z) (var-exp 'z))))))

(desugar (lambda-exp (list 'x 'y)
                     (bind-exp (list 'v 'w) (list (var-exp 'y) (var-exp 'x))
                               (app-exp (var-exp 'write)
                                        (list (var-exp 'v) (var-exp 'w))))))
  = (lambda-exp (list 'x 'y)
                (app-exp (lambda-exp (list 'v 'w)
                                     (app-exp (var-exp 'write)
                                              (list (var-exp 'v)
                                                    (var-exp 'w))))
                         (list (var-exp 'y) (var-exp 'x))))
```

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

   One can imagine a schedule as a mapping from hours to meetings. We will use numbers (e.g., 0-23) for the hours, and symbols for the meetings. We use the symbol `none` for the lack of a meeting. The type of schedules will be called `schedule` below. Consider the following procedural representation of `schedule`.

```
(defrep schedule (-> (number) symbol))

(deftype empty-schedule (-> () schedule))
(define empty-schedule
  (lambda ()
    (lambda (hour)
      'none)))

(deftype add-meeting (-> (number symbol schedule) schedule))
(define add-meeting
  (lambda (when what sched)
    (lambda (hour)
      (if (= when hour) what (meeting-at sched hour)))))

(deftype meeting-at (-> (schedule number) symbol))
(define meeting-at
  (lambda (sched hour)
    (sched hour)))
```

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

```
(define cs342 (add-meeting 14 'ComS342 (empty-schedule)))
(define thursday (add-meeting 16 'colloquium cs342))
```

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

```
(meeting-at (empty-schedule) 8) ==> none
(meeting-at cs342 14) ==> ComS342
(meeting-at cs342 16) ==> none
(meeting-at thursday 16) ==> colloquium
(meeting-at thursday 14) ==> ComS342
(meeting-at thursday 8) ==> none
```

   Your task is to transform the procedural representation on the previous 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 declaration below




;;; Now fill in the code for the operations below

(deftype empty-schedule (-> () schedule))
(define empty-schedule




(deftype add-meeting (-> (number symbol schedule) schedule))
(define add-meeting




(deftype meeting-at (-> (schedule number) symbol))
(define meeting-at
```