Fall, 2000 Name: ____________________

22C:54, section 2 — Programming Language Concepts

Test on *EOPL* Chapter 3

This test has 5 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. No photo-reduction is permitted. These notes are to be handed in at the end of the test. Use of other notes or failure to follow these instructions will be considered cheating. Please put your name in the top right corner of your notes.

During the test, if you need more space for an answer, 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. We may take off a small amount if you do not give `deftype` declarations for recursive helping procedures. However, you do not have to write such comments for procedures for which the type is stated in the problem. Correct syntax also matters. Check your code over for syntax errors. You will lose points if your code has syntax errors.

Subset of Scheme You May Use.

Unless otherwise stated in a problem, when solving problems you may only use features of Scheme, extensions that we discussed in class, and helping functions that you define yourself. The standard is defined by the *Revised^5 Report on the Algorithmic Language Scheme*.

Parts of Scheme You May *Not* Use.

Unless otherwise stated in a problem, you are prohibited from defining your own macros, and from using internal defines, all the input and output facilities, and the following keywords and procedures. (Don't worry if you don't know what these are.)

```
call-with-current-continuation  do      set!    set-car!      set-cdr!
string-set!     string-fill!    vector-set!     vector-fill!
```

1. This is a question about free and bound variables. Consider the following Scheme expression.

```
(let ((x (invert y))
      (y2 (subtract y (invert x))))
  (letrec ((iter (lambda (a b)
                   (if (done? a b)
                       b
                       (iter b (subtract y2 a)))))
           (done? (lambda (old new)
                    (small? (abs (subtract old new))))))
    (iter y y2)))
```

   (a) (10 points) Write, in set brackets, the entire set of the free variables in expression above. For example, write $\{v, w\}$, if the free variables are v and w. If there are no free variables, write $\{\}$. (You're supposed to know what a "free variable" is.)

   (b) (10 points) Write, in set brackets, the entire set of the bound variables in the expression above. For example, write $\{v, w\}$, if the bound variables are v and w. If there are no bound variables, write $\{\}$. (You're supposed to know what a "bound variable" is.)

2. (10 points) Desugar the following Scheme expression by writing and expression that has the same meaning, but which does not use `let`, `and`, `or`.

```
(let ((v (and q r))
      (w (or s t)))
   (f v w))
```

3. Briefly answer the following questions. (You are supposed to know the terms involved.)

   (a) (5 points) What are the advantages of extending a language with syntactic sugars?

   (b) (10 points) Should our type checker's syntax `define-record-type` be considered a syntactic sugar? Why or why not?

4. (30 points) This is a problem about desugaring. The language you will work with has the following concrete syntax.

```
⟨exp⟩ ::= ⟨varref⟩                    | ⟨literal⟩
        | (⟨exp⟩ {⟨exp⟩}*)            | (lambda ({⟨var⟩}* ⟨exp⟩)
        | (if ⟨exp⟩ ⟨exp⟩ ⟨exp⟩)      | (do ⟨exp⟩ unless ⟨exp⟩)
⟨varref⟩ ::= ⟨var⟩
⟨var⟩ ::= ⟨symbol⟩
⟨literal⟩ ::= ⟨number⟩
```

The abstract syntax for this grammar is defined by the following records.

```
(define-record-type varref parsed-exp
  ((var symbol)))
(define-record-type lit parsed-exp
  ((datum number)))
(define-record-type app parsed-exp
 ((rator parsed-exp) (rands (list parsed-exp))))
(define-record-type lambda parsed-exp
 ((formals (list symbol)) (body parsed-exp)))
(define-record-type if parsed-exp
 ((test-exp parsed-exp) (then-exp parsed-exp) (else-exp parsed-exp)))
(define-record-type do-exp parsed-exp
 ((body parsed-exp) (test-exp parsed-exp)))

(define-record varref (var))
(define-record lit (datum))
(define-record app (rator rands))
(define-record lambda (formals body))
(define-record if (test-exp then-exp else-exp))
(define-record do-exp (body test-exp))
```

We will also assume that there are procedures `parse-exp` and `unparse-exp` with the usual functionality and with the following types.

```
(deftype parse-exp (-> (datum) parsed-exp))
(deftype unparse-exp (-> (parsed-exp) datum))
```

Your task is to write the procedure `desugar-do`, with the following type.

```
  (deftype desugar-do (-> (parsed-exp) parsed-exp))
```

Your procedure is to use the following formula for desugaring where `E1` and `E2` stand for arbitrary expressions (instances of ⟨exp⟩) and `E1'` and `E2'` stand for their desugarings.

```
  (do E1 unless E2) ---> (if E2' 0 E1')
```

In the abstract syntax the desugaring rule is:

```
  (make-do-exp E1 E2) ---> (make-if E2' (make-lit 0) E1')
```

To receive full-credit your solution must use `variant-case`. There are some examples and space for your answer on the next page.

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

```
(unparse-exp (desugar-do (parse-exp '(do x unless y))))
 ==> (if y 0 x)
(unparse-exp (desugar-do (parse-exp 'x)))
 ==> x
(unparse-exp (desugar-do (parse-exp '(h (do 5 unless b) a))))
 ==> (h (if b 0 5) a)
(unparse-exp (desugar-do
               (parse-exp '(lambda (x) (do x unless (f x))))))
 ==> (lambda (x) (if (f x) 0 x))
(unparse-exp
  (desugar-do
    (parse-exp
      '(if g
           (do (do c unless b) unless a)
           (f (do 4 unless y))))))
 ==> (if g
         (if a
             0
             (if b 0 c))
         (f (if y 0 4)))
```

5. (25 points) This problem is about transforming procedural to record representations. The procedural representation is given on this page. There is space space for your transformation of it to a record representation on page 7.

   One can imagine a piece of music as a mapping from from time to a notes (sounds). This type will be called `music` below. We will think of time as a non-negative number. Although you don't need to understand how `note` works for this problem, for completeness its implementation is as follows.

```
;;; file note.scm
(define-record-type note note ((pitch number)))
(define-record note (pitch))
(deftype note-transpose (-> (number note) note))
(define note-transpose
  (lambda (half-steps note)
    (make-note (* (expt 2 (/ half-steps 12))
                  (note->pitch note)))))
```

   Consider the following procedural representation of `music`.

```
;;; procedural rep code below

(deftype music-generator (-> ((-> (number) note)) music))
(deftype transpose (-> (music number) music))
(deftype note-at (-> (music number) note))

(defrep music (-> (number) note))

(load-quietly "note.scm")

(define music-generator
  (lambda (f)
    (lambda (t)
      (f t))))

(define transpose
  (lambda (music half-steps)
    (lambda (t)
      (note-transpose half-steps (note-at music t)))))

(define note-at
  (lambda (m t)
    (m t)))
```

You don't need to read this page. In any case, examples aren't really going to help you do this problem.

However, if you insist on reading some examples, suppose we define

```
(deftype minute-waltz music)
(define minute-waltz
  (let ((omph (make-note 110)))
    (let ((pah (note-transpose 6 omph)))
      (music-generator
         (lambda (t)
           (cond
            ((> t 60) (make-note 0))
            ((= (remainder (inexact->exact (floor t)) 3) 0) omph)
            (else pah)))))))
```

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

```
(note-at minute-waltz 3) ==> #(note 110)
(note-at minute-waltz 4) ==> #(note 155.56349186104046)
(note-at minute-waltz 5) ==> #(note 155.56349186104046)
(note-at minute-waltz 6) ==> #(note 110)
(note-at (transpose minute-waltz 12) 3) ==> #(note 220)
(note-at (transpose minute-waltz -1) 3) ==> #(note 103.82617439498628)
```

Your task is to transform the above procedural representation into one that uses records. Do this by giving the `define-record-type` and `define-record` declarations needed and the bodies of the procedures in the spaces provided below. You must use `variant-case` in your solution.

```
(deftype music-generator (-> ((-> (number) note)) music))
(deftype transpose (-> (music number) music))
(deftype note-at (-> (music number) note))

(load-quietly "note.scm")

;;; Write the define-record-type declarations below









;;; Write the define-record declarations below








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

(define music-generator




(define transpose




(define note-at
```