Spring, 1999 Name: ______________________

My Section Letter: ______________ My Section Day and Time : ______________

Com S 342 — Principles of Programming Languages

Test on *EOPL* Chapters 2.3 and 3

This test has 7 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 8pt 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 will take off a small amount if you do not give `TYPE` comments 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 standard features of the language that we discussed in class, `define-record`, `variant-case`, 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
```

1. (15 points) This is a problem about bound variables. In each of the spaces provided below, write, in set brackets, the entire set of the bound variables in the corresponding Scheme expression. 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.)

   (a) `(car (cdr ls))`

   (b) `(lambda (i n ls) (plus n (car ls)))`

   (c)
```
(let ((avg3 (lambda (f p n) (average (car p) (cdr p) c n)))
      (a (car ls))
      (b (cadr ls))
      (c (caddr ls)))
  (avg3 foo b ten))
```

   (d)
```
(letrec ((sing (lambda (ls pitch endval)
                 (if (null? ls)
                     endval
                     (((return-last3 (set-pitch! pitch))
                        (talk (car ls)))
                       (sing (cdr ls))))))
         (talk (lambda (word) (say word)))
         (last (lambda (a) (lambda (b) b)))
         (return-last3 (lambda (x) (lambda (y) (lambda (val) val)))))
  (sing notes))
```

2. (15 points) This is a problem about free variables. In each of the spaces provided below, write, in set brackets, the entire set of the free variables in the corresponding Scheme expression. 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.)

   Note: these are the same expressions as above.

   (a) `(car (cdr ls))`

   (b) `(lambda (i n ls) (plus n (car ls)))`

   (c)
```
(let ((avg3 (lambda (f p n) (average (car p) (cdr p) c n)))
      (a (car ls))
      (b (cadr ls))
      (c (caddr ls)))
  (avg3 foo b ten))
```

   (d)
```
(letrec ((sing (lambda (ls pitch endval)
                 (if (null? ls)
                     endval
                     (((return-last3 (set-pitch! pitch))
                       (talk (car ls)))
                      (sing (cdr ls))))))
         (talk (lambda (word) (say word)))
         (last (lambda (a) (lambda (b) b)))
         (return-last3 (lambda (x) (lambda (y) (lambda (val) val)))))
  (sing notes))
```

3. (5 points) Desugar the following expression by writing a Scheme expression that has the same meaning, but which does not use `let`.

```
(+ a (let ((x (f b c)))
       (+ 3 x)))
```

4. (10 points) In the following expression, draw an arrow from each bound ⟨varref⟩ to its declaration.

```
(lambda (f g h)

  ((lambda (x)

     (lambda (z x)

       (f (g x) z)))

   (lambda (x ls f)

     (lambda (g)

       (g (f x ls))))))
```

5. (10 points) Give the lexical address form of the above expression, by filling in the blanks below. (You are supposed to know what the "lexical address form" of an expression is.)

```
(lambda (f g h)
  ((lambda (x)
     (lambda (z x)


   (lambda (x ls f)
     (lambda (g)
```

6. (20 points) Suppose a picture (e.g., a JPEG image) is represented in Scheme as a vector of pixels, where each pixel is a number. Without using `vector->list`, write a procedure,

```
  picture-similar? : (-> ((vector number) (vector number)) boolean)
```

that takes two such vectors, `pic1` and `pic2`, and returns true if the two vectors have the same length, and if the absolute value of the difference between the nth number in `pic1` and the nth number in `pic2` is strictly less than 5.

The following are examples.

```
(picture-similar? (vector 0 0 0) (vector 0 3 4)) ==> #t
(picture-similar? (vector 0 0 0) (vector 0 3 4 0 5 6)) ==> #f
(picture-similar? '#(1 2 3 4 5 6 7 9) '#(0 1 2 3 4 5 6 8)) ==> #t
(picture-similar? '#(6 7 8 9 10 11 12 21) '#(0 1 2 3 4 5 6 8)) ==> #f
(picture-similar? '#(0 5 6) '#(-4 9 6)) ==> #t
(picture-similar? '#(-4 9 6) '#(0 5 6)) ==> #t
(picture-similar? '#() '#()) ==> #t
```

Hints: remember that Scheme vectors have indexes that start with zero. You can use `abs` to compute absolute values, `vector-length` to get the length of a vector, and `vector-ref` to access an element.

7. (25 points) Consider the following grammar.

$$
\begin{aligned}
\langle\text{exp}\rangle &::= \langle\text{varref}\rangle \\
&\mid \langle\text{lit}\rangle \\
&\mid \texttt{(}\ \langle\text{exp}\rangle\ \{\langle\text{exp}\rangle\}^*\ \texttt{)} \\
&\mid \texttt{(}\ \texttt{if}\ \langle\text{exp}\rangle\ \langle\text{exp}\rangle\ \langle\text{exp}\rangle\ \texttt{)} \\
&\mid \texttt{(}\ \texttt{when}\ \langle\text{exp}\rangle\ \langle\text{exp}\rangle\ \texttt{)} \\
\langle\text{varref}\rangle &::= \langle\text{symbol}\rangle \\
\langle\text{lit}\rangle &::= \langle\text{number}\rangle
\end{aligned}
$$

The abstract syntax for this grammar is defined by the following records. The variant record type formed from the union of these record types is called `parsed-exp` below.

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

Write a procedure, `syntax-expand` of type `(-> (parsed-exp) parsed-exp)` that desugars `when` expressions into `if` expressions. Your procedure is to use the following formula for desugaring, where `E1` and `E2` are arbitrary parsed-exps.

```
  #(when E1 E2) ==> #(if E1 E2 #(lit 0))
```

There are some examples on the next page.

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

```
(syntax-expand (make-when (make-varref 'x) (make-varref 'y)))
  = (make-if (make-varref 'x) (make-varref 'y) (make-lit 0))
  ==> #(if #(varref x) #(varref y) #(lit 0))

(syntax-expand (make-varref 'x))
  = (make-varref 'x)
  ==> #(varref x)

(syntax-expand (make-app (make-varref 'h)
                         (list (make-when (make-varref 'b) (make-lit 5))
                               (make-varref 'a))))
  = (make-app (make-varref 'h)
              (list (make-if (make-varref 'b) (make-lit 5) (make-lit 0))
                    (make-varref 'a)))
  ==> #(app #(varref h)
            (#(if #(varref b) #(lit 5) #(lit 0))
             #(varref a)))

(syntax-expand (make-if (make-varref 'g)
                  (make-when (make-varref 'a)
                             (make-when (make-varref 'b) (make-varref 'c)))
                  (make-app (make-varref 'f)
                            (list (make-when (make-varref 'y) (make-lit 4))))))
  = (make-if (make-varref 'g)
             (make-if (make-varref 'a)
                      (make-if (make-varref 'b) (make-varref 'c) (make-lit 0))
                      (make-lit 0))
             (make-app (make-varref 'f)
                       (list (make-if (make-varref 'y)
                                      (make-lit 4)
                                      (make-lit 0)))))
  ==> #(if #(varref g)
           #(if #(varref a)
                #(if #(varref b) #(varref c) #(lit 0))
                #(lit 0))
           #(app #(varref f)
                 (#(if #(varref y)
                       #(lit 4)
                       #(lit 0)))))
```