Fall, 1995 Name: ____________________
. My Section Day and Time : ____________________

Com S 342 — Principles of Programming Languages

Test on *EOPL* Chapters 2.3 through 3

This test has 6 questions and pages numbered 1 through 4.

Reminders

For this test, you can use one page (one side, no less than 10pt font) of notes. These notes are to be handed in at the end of the test.

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.

Indentation is important to us for "clarity" points; if your code is sloppy or hard to read, you will lose points. Please indent as described in class.

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, and helping functions that you define yourself. The standard is defined by the *Revised*^{4} *Report on the Algorithmic Language Scheme.*

Parts of Scheme You May *Not* Use

Unless otherwise stated in a problem, you are prohibited from using internal defines, all the input and output facilities, macros, 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. (10 points) In each of the spaces provided ("____________") below, write, in set brackets, the entire set of the free variables in the preceeding expression. For example, write $\{x, y\}$, if the free variables are x and y. If there are no free variables, write $\{\}$. (You're supposed to know what a "free variable" is.)

   (a) `(lambda (z) ((lambda (ls) (car ls)) (f z)))` ____________

   (b) `((lambda (q) q) q)` ____________

   (c) `(let ((x 3) (y x)) (+ x 7))` ____________

2. (5 points) In each of the spaces provided ("____________") below, write, in set brackets, the entire set of the bound variables in the preceeding expression. For example, write $\{x, y\}$, if the bound variables are x and y. If there are no bound variables, write $\{\}$. (You're supposed to know what a "bound variable" is.)

   (a) `(lambda (ls) (car (cdr ls)))` ____________

   (b) `(lambda (x) (lambda (y) x))` ____________

3. (10 points) Consider the following expression.

```
(lambda (list car cdr)
  ((lambda (x)
     (lambda (z w x)
       (list (car w) x z)))
   (lambda (x ls list)
     (lambda (car)
       (car (list x ls))))))
```

   Give the lexical address form of the above expression, by filling in the blanks below, replacing all the ⟨varref⟩s in the above expression by their lexical addresses. (You're supposed to know what a "lexical address" is.)

```
(lambda (list car cdr)
  ((lambda (x)
     (lambda (z w x)
       __________________________________________________))
   (lambda (x ls list)
     (lambda (car)
       __________________________________________________))))
```

4. (10 points) Name a programming language in which the representations of user-defined ADTs are:

   (a) opaque ____________

   (b) transparent ____________

   (You're supposed to know what these terms mean.)

5. (20 points) Consider the following grammar for a version of the lambda calculus, where ⟨number⟩ and ⟨symbol⟩ are as usual.

   ⟨exp⟩ ::= ⟨varref⟩ | ⟨number⟩ | (⟨exp⟩ ⟨exp⟩) | (lambda (⟨var⟩) ⟨exp⟩)
   ⟨varref⟩ ::= ⟨symbol⟩
   ⟨var⟩ ::= ⟨symbol⟩

   Write a procedure, `count-free-occurrences`, that takes a symbol, `var`, and a ⟨exp⟩, `exp`, and returns the number of free occurrences of `var` in `exp`. The following are examples.

```
(count-free-occurrences 'x 'x) ==> 1
(count-free-occurrences 'x '((g x) (f x))) ==> 2
(count-free-occurrences 'z '(lambda (z) ((g z) (f z)))) ==> 0
(count-free-occurrences 'q '(lambda (x) ((g x) (f x)))) ==> 0
(count-free-occurrences 'g '(lambda (x) ((g x) (f x)))) ==> 1
(count-free-occurrences 'car
                        '((lambda (f) (lambda (ls) (car (car ls))))
                          ((lambda (car) ((ccompose car) cdr))
                           (lambda (x) (lambda (y) ((car (car y)) (car x)))))))
       ==> 5
```

   Hint: *don't* use an abstract syntax or too many helping procedures.

6. (20 points) In this problem, we will use the following records to represent the type "binary-tree". That is, the type binary-tree is the union of the three record types below. (Note that this is slightly different than in the text.)

```
(define-record empty ())
(define-record leaf (number))
(define-record interior (left-tree number right-tree))
```

Write a procedure `inorder-traversal`, that takes a binary-tree, `btree`, and returns a list of the numbers in `btree` "in order". That is, the list returned is such that each occurrence of a number in the list preceeds the numbers to its right in `btree`, and follows any numbers to its left in `btree`. The following are examples.

```
(inorder-traversal (make-empty)) ==> ()
(inorder-traversal (make-leaf 3)) ==> (3)
(inorder-traversal (make-interior (make-empty) 3 (make-empty))) ==> (3)
(inorder-traversal (make-interior (make-leaf 2) 3 (make-empty))) ==> (2 3)
(inorder-traversal (make-interior (make-leaf 2) 3 (make-leaf 4))) ==> (2 3 4)
(let ((tree-578 (make-interior (make-leaf 5) 7 (make-leaf 8))))
  (let ((tree-123 (make-interior (make-leaf 1) 2 (make-leaf 3))))
    (inorder-traversal
     (make-interior (make-interior tree-123 4 tree-123)
                    6
                    (make-interior tree-578 9 (make-empty))))))
    ==> (1 2 3 4 1 2 3 6 5 7 8 9)
```

You must use `variant-case` in your solution. (Hint: use Scheme's `append` procedure.)