Fall, 1999 Name: _____

My Grading TA: _____ My Section Day and Time : _____

Com S 342 — Principles of Programming Languages

Test on *SICP* Sections 2.2.2–3 and 2.3.1–2

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 9pt font) of notes. Don't use anything with printing on the other side. No photo-reduction is permitted. You may not share your notes with anyone else during the test. These notes are to be handed in at the end of the test, so please have your name in the upper right hand corner of your notes. Use of other notes or failure to follow these instructions will be considered cheating.

WARNING: you won't have time to learn the material on during the test. Just write down what would be too tedious to remember otherwise.

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.

Of course, you may write helping procedures or methods whenever you wish. It may be helpful to give a comment that describes what they do, if it's not completely clear from the name.

You may use any of the built-in procedures in Scheme.

1. (10 points) Draw a box-and-pointer diagram for the following Scheme expression. You are supposed to know what a "box-and-pointer diagram" is.

```
(let ((ls (cons 1 nil)))
   (list 3 4 ls (list ls 2)))
```

2. (10 points) Fill in the missing expressions to complete the definition of the `filter` procedure, using the `accumulate` procedure from the book. (You are supposed to know what `filter` and `accumulate` do.)

```
(define (filter pred sequence)
  (accumulate (lambda (x y) ____________________________________________)
              nil
              sequence))
```

3. (10 points) Fill in the missing expressions to complete the definition of the `flatmap` procedure, using the `accumulate` procedure from the book. (You are supposed to know what `flatmap` and `accumulate` do.)

```
(define (flatmap proc sequence)
  (accumulate (lambda (x y) ____________________________________________)
              nil
              sequence))
```

4. (10 points) Using `accumulate`, `filter`, and `map`, write a procedure `root-mean-square` that takes a list of symbols and numbers, throws away the symbols, and returns the positive square root of the sum of the squares of the numbers in the list. (You can use `sqrt` to return the positive square root of a number.) The following are examples.

```
(root-mean-square '()) ==> 0.0
(root-mean-square '(I swam 4 hours at 3 am and then drove to work)) ==> 5.0
(root-mean-square '(hah I ran 16 miles at 2 am and walked 5 miles))
    ==> 16.8819430161341
```

You will only get full credit if you take appropriate advantage of `accumulate`, `filter`, and `map`.

5. (20 points) Write a procedure subst-tree, that takes two symbols, `new` and `old`, and a tree of symbols, `tree`, and returns a tree of symbols that is like `tree` except that all occurrences of `old` are replaced by `new`. The following are examples.

```
(subst-tree 'strange 'charm '()) ==> ()
(subst-tree 'strange 'charm '((charm) is (good)))
                        ==> ((strange) is (good))
(subst-tree 'scheme 'java
            '((sentence (np () (java))
                        (vp (has (np () (great success)))))))
         ==> ((sentence (np () (scheme))
                        (vp (has (np () (great success))))))
(subst-tree 'n 'o '((n o t) (o n e) (b a l l (o o n))))
               ==> ((n n t) (n n e) (b a l l (n n n)))
```

You may use the `atom?` procedure from the library in your solution if you wish.

6. (20 points) Consider the following grammar, which is a highly restricted subset of Scheme.

```
<stmt> ::= <assigment>
         | <conditional>
         | <compound>
<assignment> ::= (set! <symbol> <number>)
<conditional> ::= (if <number> <stmt> <stmt>)
<compound> ::= (begin <stmt> <stmt>)
```

The nonterminals `<symbol>` and `<number>` respectively represent Scheme symbols and numbers. In Scheme, write a procedure `simplify`, that takes a `<stmt>` and returns an equivalent statment, where statements containing `(if` n `s1 s2)` become statements containing `s2`, if n is `0`, and `s1` otherwise. The following are examples:

```
(simplify (make-assignment 'x 3)) ==> (set! x 3)
(simplify '(set! x 3)) ==> (set! x 3)
(simplify '(if 1 (set! x 3) (set! y 4))) ==> (set! x 3)
(simplify '(if 0 (set! x 3) (set! y 4))) ==> (set! y 4)
(simplify '(begin (if 0 (set! z 7) (set! q 10))
                  (if 0 (set! x 3) (set! y 4))))
       ==> (begin (set! q 10) (set! y 4))
(simplify '(begin (set! a 0)
                  (if 1
                      (begin (if 1 (set! m 1) (set! n 8))
                             (if 1 (set! f 5) (set! g 6)))
                      (begin (if 0 (set! z 7) (set! q 10))
                             (if 0 (set! x 3) (set! y 4))))))
       ==> (begin (set! a 0) (begin (set! m 1) (set! f 5)))
```

You can use the helping procedures for this grammar on the next page in solving the problem.

The following are helping procedures for the problem on the previous page and the one on the next page.

```
(define (assignment-stmt? stmt)
  ;; TYPE: (-> (stmt) boolean)
  (eq? (car stmt) 'set!))
(define (make-assignment symbol number)
  ;; TYPE: (-> (symbol number) assignment)
  (list 'set! symbol number))
(define (assignment-variable assignment)
  ;; TYPE: (-> (assignment) symbol)
  (cadr assignment))
(define (assignment-expr assignment)
  ;; TYPE: (-> (assignment) number)
  (caddr assignment))

(define (conditional-stmt? stmt)
  ;; TYPE: (-> (stmt) boolean)
  (eq? (car stmt) 'if))
(define (make-conditional number s1 s2)
  ;; TYPE: (-> (symbol stmt stmt) conditional)
  (list 'if number s1 s2))
(define (conditional-test conditional)
  ;; TYPE: (-> (conditional) number)
  (cadr conditional))
(define (conditional-true-stmt conditional)
  ;; TYPE: (-> (conditional) stmt)
  (caddr conditional))
(define (conditional-false-stmt conditional)
  ;; TYPE: (-> (conditional) stmt)
  (cadddr conditional))

(define (compound-stmt? stmt)
  ;; TYPE: (-> (stmt) boolean)
  (eq? (car stmt) 'begin))
(define (make-compound s1 s2)
  ;; TYPE: (-> (stmt stmt) compound)
  (list 'begin s1 s2))
(define (compound-first compound)
  ;; TYPE: (-> (compound) stmt)
  (cadr compound))
(define (compound-second compound)
  ;; TYPE: (-> (compound) stmt)
  (caddr compound))
```

7. (20 points) Using the grammar from the previous problem, and the helpers on the previous page, write a procedure, `count-false-sets`, that takes a symbol, `var`, and a statement, `stmt`, and returns the number of times that `var` is assigned within the false part of any conditional statement, including substatements. The following are examples.

```
(count-false-sets 'x '(set! x 3)) ==> 0
(count-false-sets 'x '(if 1 (set! x 3) (set! y 4))) ==> 0
(count-false-sets 'x '(if 0 (set! x 3) (set! y 4))) ==> 0
(count-false-sets 'x '(if 1 (set! y 4) (set! x 3))) ==> 1
(count-false-sets 'x '(if 0 (set! y 4) (set! x 3))) ==> 1
(count-false-sets 'x '(if 0 (set! x 7) (set! x 6))) ==> 1
(count-false-sets 'x '(begin
                        (if 1 (set! y 9) (set! x 8))
                        (if 1 (set! y 4) (set! x 3)))) ==> 2
(count-false-sets 'x '(if 1
                          (if 0 (set! y 5) (set! x 1))
                          (if 0 (set! y 4) (set! x 3)))) ==> 2
(count-false-sets 'y '(begin (set! y 7)
                             (if 1
                                 (if 0
                                     (set! y 5)
                                     (begin (set! y 9)
                                            (set! y 8)))
                                 (if 0 (set! y 4) (set! y 3))))) ==> 4
```