Last Name: _____ (as on your ID)

First Name: _____

ID: _____ Score: /25

Instructions: Write your solutions to the following problems in the space provided. Read each problem carefully and recheck your work. All work must be your own and you must work independently.

Q1 — Dotted Tails [5 points]

Recall that Scheme will print *cons* cells in a list format unless the *cdr* is not a pair or the empty list (*null*). In the event the *cdr* is not a pair or *null* (for example, when it's a number), then a "`.`" will be placed before the *cdr*'s print representation.

Write what the Scheme interpreter prints for each of the following expressions:

A.) `(cons 1 2)`

B.) `(cons 1 (cons 2 '()))`

C.) `(cons 1 (cons 2 3))`

D.) `(cons 1 (list 2 3))`

Q2 — Typing [5 points]

Specify the types of the following Scheme functions. You must:

1. Write 'List<T>' to specify the type of a list with elements of type T.
2. Write '(S,T) → U' to specify the type of a function that takes two parameters (one S and one T), and returns a value of type of U.
3. Write 'T | U' to specify the type of a value that could be *either of* type T or type U. For example, the *cdr* of a cons cell for a list of A elements has the type 'List<A> | Null'.

If "any" type can be allowed, use type variables like A, B, C, D, …. Be careful for when two such types need to be consistent. Use the types String, Number, and List, along with the function types and others.

A.) append:

```
(define (append x y)
  (if (null? x)
      y
      (cons (car x) (append (cdr x) y))))
```

B.) even-fibs:

```
(define (even-fibs n)
  (let next ([k 0])
    (if (> k n)
        '()
        (let ([f (fib k)])
          (if (even? f)
              (cons f (next (+ k 1)))
              (next (+ k 1)))))))
```

Assume: fib: Number → Number

C.) average-damp:

```
(define (average-damp f)
  (lambda (x)
    (average x (f x))))
```

Assume: average: Number, Number → Number

Q3 — Conspiracy [5 points]

Consider the following group of Scheme functions: The procedure `make-triple` takes in three arguments and returns a *triple*. A *triple* can have its first, second, and third elements accessed, respectively, by the `first`, `second`, and `third` functions.

Thus, the following properties hold:

```
 (first (make-triple a b c)) → a
(second (make-triple a b c)) → b
 (third (make-triple a b c)) → c
```

Figure out how a *triple* is represented by looking at the already-completed `second` function. Then, complete the implementation of the remaining functions.

A.)
```
(define (make-triple a b c)
```

B.)
```
(define (first triple)
```

[Given.]
```
(define (second triple)
  (triple (lambda (a b c) b)))
```

C.)
```
(define (third triple)
```

Q4 — Quoted Output [5 points]

Write what the following expression evaluates to:

```
((lambda (x) (list x (list (quote quote) x)))
 (quote (lambda (x) (list x (list (quote quote) x)))))
```

Please show the output with all valid `(quote` *datum*`)` special-form expressions in their (correct) abbreviated form: `'`*datum*.

Q5 — Unification [5 points]

Show how the following pairs of patterns unify with each other or explain why they cannot be unified. Variables start with an uppercase letter (e.g., `W`, `X`, `Y`, `Z`), and atoms start with a lowercase letter (`a`, `b`, `c`, `d`, `e`, `f`...).

If there is an assignment to all of the variables that is consistent with both patterns, then show the bindings in the most general terms possible. For example, the pattern `[a|X]` and the pattern `[a, b, Y]` unify, where `X` must be a list that starts with `b` and ends with whatever `Y` is. The bindings would be written as: `X = [b, Y]`.

A.) `[X, X] = [[a, Y, c], [a, b, Z]]`

B.) `[X, Y, [Z, a, W]] = [X, [a, W, b], Y]`

C.) `[X, a] = [[b, Y], Z]`

D.) `[f, X, Y] = [f, [g, Y], X]`