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 — Side-effects and Argument Evaluation Order [5 points]

Scheme leaves unspecified the order in which the subexpressions should be evaluated (e.g., left-to-right or right-to-left). Because Scheme has assignment, the order in which the arguments are evaluated matters, which could potentially lead to different results.

Define a simple procedure `f` such that evaluating

```
(+ (f 0) (f 1))
```

will return `0` if the arguments to `+` are evaluated from left-to-right but will return `1` if the arguments are evaluated from right-to-left.

Q2 — Typing [5 points]

Write the type of this mystery function **in the most general terms**:

```
(define (mystery f g i)
  (let ([r (g (+ i 1))])
    (r f)))
```

If a binding could be of any type, use generic type names like A, B, C, …. If a binding must be: a number, use 'Number'; a string, use 'String'; a Boolean, use 'Boolean.' You must use '→' for function types. Use parentheses for grouping.

For example, A → (B → Number) is the type of a function that takes in an argument of type A and returns a function that takes in an argument of type B and returns a number.

Q3 — Java Tricks [5 points]

Consider the following Java program fragment:

```
public class BooleanFactory {
  public static final MyBoolean TRUE = new MyTrue();
  public static final MyBoolean FALSE = new MyFalse();

  /* Returns TRUE if the given Boolean is true; returns FALSE otherwise. */
  public static MyBoolean make(boolean b) {
    if (b) {
      return TRUE;
    } else {
      return FALSE;
    }
    /* Note: This is equivalent to using the conditional: return (b) ? TRUE : FALSE; */
  }
}
```

Provide an implementation of the `BooleanFactory.make` method (in Java) that behaves the same as the implementation above, **without** using `if`, `switch`, or the conditional (`?:`) operator. You may use arrays, lists, or maps if you wish.

Q4 — Recursion and Higher-Order Functions [5 points]

The procedure `square-list` takes a list of numbers as arguments and returns a list of the squares of those numbers. For example,

```
(square-list (list 1 2 3 4))
```

produces

```
(1 4 9 16).
```

Below are two different, but incomplete, definitions of `square-list`. Complete both of them by providing the program text for the missing expressions *A*, *B*, *C*, *D* (marked with question marks on both sides).

A.

```
(define (square-list items)
  (if (null? items)
      '()
      (cons     ?A?          ?B?          )))
```

B.

```
(define (square-list items)
  (map          ?C?          ?D?          ))
```

Q5 — Continuation Passing Style [5 points]

Convert the `hypotenuse-length` function to continuation passing style.

You may assume that the callback-receiving variants of `+`, `*`, `f`, `g`, and `sqrt` have already been defined as `+/cb`, `*/cb`, `f/cb`, `g/cb`, and `sqrt/cb`. Assume a left-to-right order of evaluation for sub-expressions.

```
(define (hypotenuse-length x y)
  (sqrt (+ (* x x) (* y y))))
```

⇒

```
(define (hypotenuse-length/cb x y cb)
```

)

To help you, here is an example of a conversion to continuation passing style for the function `h`:

```
(define (h u v)
  (+ (f u) (g v)))
```

⇒

```
(define (h/cb u v cb)
  (f/cb u
        (lambda (f-of-u)
          (g/cb v
                (lambda (g-of-v)
                  (+/cb f-of-u g-of-v cb))))))
```