Instructions: Write your solutions to the following problems in your blue book. Be sure to write your name and student ID number on the front. Have your photo identification ready when you hand in your blue book.

Read each problem carefully and recheck your work. Clearly indicate your answer. If you provide what looks like multiple answers for the same problem then no points will be awarded (even if all answers provided are correct).

All work must be your own and you must work independently.

Q1 — Implement a Scheme Function [10 points]

Define a Scheme procedure named `last-pair` that returns the list that contains *only* the last element of a given (non-empty) list. For example:

```
> (last-pair (list 23 72 149 34))
'(34)
```

Q2 — Data Examples [10 points]

Ben Bitdiddle decides to write a procedure to count the number of pairs in any list structure. "It's easy," he reasons. "The number of pairs in any structure is the number in the *car* plus the number in the *cdr* plus one more to count the current pair."

So, Ben writes the following procedure:

```
(define (count-pairs x)
  (if (not (pair? x))
      0
      (+ (count-pairs (car x))
         (count-pairs (cdr x))
         1)))
```

Show that this procedure is *not* correct. In particular: **Draw box-and-pointer diagrams representing list structures made up of exactly three pairs for which Ben's procedure would:**

A.) Return 3

B.) Return 4

C.) Return 7

D.) Never return at all

Q3 — Code in the Blank [10 points]

For this problem and the next, use the following definition of `fold-right`:

```
(define (fold-right op init lst)
  (if (null? lst)
      init
      (op (car lst) (fold-right op init (cdr lst)))))
```

Evaluating a polynomial in x at a given value of x can be formulated as an accumulation. We evaluate the polynomial

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

using a well-known algorithm called Horner's rule, which structures the computation as

$$(\ldots (a_n x + a_{n-1})x + \cdots + a_1)x + a_0.$$

In other words, we start with a_n, multiply by x, add a_{n-1}, multiply by x, and so on, until we reach a_0. **Complete the *MISSING-PART* of the following template to produce a procedure that evaluates a polynomial using Horner's rule**

```
(define (horner-eval x coefficient-sequence)
  (fold-right (lambda (this-coeff higher-terms)
                MISSING-PART )
              0
              coefficient-sequence))
```

Assume that the coefficients of the polynomial are arranged in a sequence, from a_0 through a_n. For example, to compute $1 + 3x + 5x^3 + x^5$ at x=2:

```
> (horner-eval 2 (list 1 3 0 5 0 1))
79
```

Q4 — Code in the Blank [10 points]

Using the definition of fold-right from Q?1: **Complete the *MISSING-PART* to compute the basic `length` procedure on a list**

```
(define (length sequence)
  (fold-right  MISSING-PART  0 sequence))
```

Q5 — Lambda Calculus [10 points]

Recall from the lambda calculus lecture that Booleans, natural numbers, and cons cells can *all* be represented in terms of single-argument functions!

```
   true = λx. λy. x
  false = λx. λy. y
     if = λp. λthn. λels. (p thn els)
   cons = λa. λb. λs. (s a b)
    car = λp. (p (λa. λb. a))
    cdr = λp. (p (λa. λb. b))
      0 = λs. λz. z
      1 = λs. λz. (s z)
   succ = λn. (λs. λz. (s (n s z)))
   plus = λn. λm. (n succ m)
iszero = λn. n (λi. false) true
```

Even though cons cells are available, full list structures are *not* supported because the empty list value and the ***null?*** predicate haven't been defined. However, supporting these operations requires different definitions for ***cons***, ***car***, and ***cdr***.

Write new definitions for ***cons***, ***car***, and ***cdr*** that support lists (name them ***list-cons***, ***list-car***, ***list-cdr***) in addition to defining two new values: ***list-null?*** and ***the-empty-list***.

You may use any of the existing definitions above if you wish, and you may use either Scheme or lambda calculus syntax in your answer.

A.) Write a definition for *list-cons*

B.) Write a definition for *list-car*

C.) Write a definition for *list-cdr*

D.) Write a definition for *list-null?*

E.) Write a definition for *the-empty-list*

Q6 — Continuation Passing Style [10 points]

Recall that, to rewrite our programs in continuation-passing style, we had to provide alternative definitions for some of the primitives.

A.) Using the definition for `+`, define a continuation-passing style `+/cb` function that takes in two numbers and sends their sum to the given callback. For example, it would begin as: `(define (+/cb n m cb)` ...

B.) Do the same for a two-argument version of `*`, to make `*/cb`

C.) Do the same for `cons`, to make `cons/cb`

D.) Do the same for the `abs` function to make `abs/cb`

Q7 — Ambiguous Grammars [10 points]

Consider the familiar if-then-else construct used in many languages:

S ::= `if` *E* `then` *S* `else` *S*
S ::= `if` *E* `then` *S*
S ::= *Identifier* `:=` *Number* `;`
E ::= Identifier `==` *Number*
Identifier ::= `a` | `b` | `c`
Number ::= `0` | `1` | `2`

Where *S* stands for *Statement*, *E* stands for *Expression*, and the tokens are: '`if`', '`then`', '`else`', '`:=`', '`;`', '`==`', '`a`', '`b`', '`c`', '`0`', '`1`', '`2`'.

The above grammar is considered an ambiguous grammar because it's possible for the *same* sequence of tokens to lead to *two different* parse trees.

By drawing parse trees, show how the sequence of tokens:

```
if a == 0 then if b == 1 then c := 1; else c:= 2;
```

can lead to two different parse trees

Q8 — Higher Order Functions [10 points]

Implement a function, named **`repeated`**, that takes in a procedure **`op`** and a non-negative number **`n`**, and returns a *procedure* that, once given a value, applies **`op`** to that value *n*-times.

For example, **`repeated`** could be used to define multiplication:

```
(define (mult n m)
  (let ([add-n (lambda (x) (+ n x))])
    ((repeated add-n m) 0)))
```

where

```
> (mult 5 7)
35
```

Implement `repeated`, start it with **`(define (repeated op n)`**...

Q9 — Typing [10 points]

Write in the most general terms the *type* of `repeated` (from **Q8**)

QX — List Processing [10 points]

Implement a function that *splits and flattens,* named `splat`, that takes a list of symbols and numbers (with the potential for lists to be arbitrarily nested) and returns a pair of *flattened* lists, one consisting only of the symbols and the other only of the numbers.

For example,

```
> (splat '(0 () (a d) (h o c) 1 (l o (c (o))) 3 1 4))
'((a d h o c l o c o) (0 1 3 1 4))
```