Exercise 1: Insert

insert is a procedure which takes two parameters, a binary procedure and a list, and gives the result of applying the procedure to neighboring values. There are two ways to think about insert based on the way we interpret

\[(\text{insert } \text{proc} \ (\text{list } v_0 \ v_1 \ v_2 \ \ldots \ v_{n-1} \ v_n))\]

i. We could interpret the call to insert as

\[(\text{proc } v_0\\ \ (\text{proc } v_1\\ \ (\text{proc } v_2\\ \ \ldots\\ \ (\text{proc } v_{n-1}\\ \ v_n) \ldots)))\]

ii. We could interpret the call to insert as

\[(\text{proc}\\ \ (\text{proc}\\ \ \ldots\\ \ (\text{proc}\\ \ (\text{proc}\\ \ (\text{proc}\\ v_0\\ v_1)\\ v_2)\\ v_3)\\ \ldots\\ v_{n-1})\\ v_n)\]

That is, we can apply the operation in a rightmost manner (i) or in a leftmost manner (ii). For addition, the difference is between grouping like this

\[(v_0 + (v_1 + (v_2 + \ldots + (v_{n-1} + v_n) \ldots)))\]

or like this
\((\ldots (v_0 + v_1) + v_2) + \ldots + v_{n-1}) + v_n\)

a. Does it make a difference which way we do things? (Hint, consider subtraction as a binary operation.)

b. Implement the rightmost version of insert. You should find that this works like the standard version of sum.

c. Implement the leftmost version of insert. You should find that this works more like the version of sum we did in class.

**Exercise 2: Making Lists**

a. Define and test a generate-list procedure that takes two arguments: (1) a one-argument procedure, proc, that can be applied to a natural number and (2) n, a natural number. Your procedure should generate a list of length n whose ith element is the result of applying proc to i. For example,

```plaintext
> (generate-list (lambda (x) (* x x)) 6)
(0 1 4 9 16 25)
```

b. Define and test a generate-lister procedure that takes one argument -- a one-argument procedure, proc, that can be applied to a natural number -- and generates a new procedure that takes one parameter, a natural number, n, and returns a list of length n whose ith element is the result of applying proc to i.

**Exercise 3: Right section**

In the reading, you saw how we might define left-section, which fills in the left of the two arguments of a binary procedure. Define and test the analogous higher-order procedure right-section, which takes a procedure of two arguments and a value to drop in as its second argument, and returns the operator section that expects the first argument. (For instance, (right-section expt 3) is a procedure that computes the cube of any number it is given.)

**Exercise 4: Powers of Two**

Using the generate-lister procedure from you defined in a previous lab and an appropriate operator section, define a procedure powers-of-two that constructs and returns a list of powers of two, in ascending order, given the length of the desired list:

```plaintext
> (powers-of-two 7)
(1 2 4 8 16 32 64)
```

**Exercise 5: Removing Elements**

Here is an interesting list of natural numbers:
(define republican-voter-ids (list 1471 4270))

Define a Scheme procedure \textit{remove-republicans} that takes a list of non-empty lists as its argument and filters out of it the lists in which the first element is also an element of \texttt{republican-voter-ids}.

**Exercise 6: Intersection**

Define \textit{intersection} (given two lists with no duplicates, return the list containing only elements that appear in both lists) using \textit{remove}, \textit{complement}, \textit{member}, and \textit{right-section}.

**Exercise 7: Filtering**

The filters constructed by \textit{remove} are designed to \textit{exclude} list elements that satisfy a given predicate. Define a higher-order procedure \textit{make-filter} that returns a filtering procedure that \textit{retains} the elements that satisfy a given predicate (excluding those that \textit{fail} to satisfy it). For instance, applying the filter (\textit{make-filter even?}) to a list of integers should return a list consisting of just the even elements of the given list.