1. Define a SCHEME function `(ins x l)` which takes a value x (an integer) and a *sorted* list ℓ (in increasing order) and inserts x at the right location within ℓ so that the list remains sorted. Note that `ins` produces a new list ℓ' identical to ℓ except for the addition of x at the right spot. For instance, the call

   ```
   (ins 5 (list 1 2 4 6 7))
   ```

   produces the list

   ```
   (1 2 4 5 6 7)
   ```

   As before, this leaves the input list ℓ in pristine condition.

2. Armed with `ins`, you are now ready to implement another sorting algorithm knows as *insertion sort*. The idea of the algorithm is straightforward. Given an unsorted list ℓ, it proceeds by peeling off elements from the front of ℓ and inserting them (one at a time of course) at their "sweet spot" within a sorted list ℓ' that starts off as an empty list. For instance the call

   ```
   (insSort (list 3 5 1 6 9 0 2 7))
   ```

   produces the list

   ```
   (0 1 2 3 5 6 7 9)
   ```

   Unsurprisingly, *insertion sort* is closely related to *selection sort* which we covered in class. Write a SCHEME function `(insSort l)` which, given a list ℓ of integers, produces a sorted permutation of ℓ (in increasing order).

3. (a) Define a SCHEME procedure, named `(fold-right op initial sequence)` which accumulates the values in the list `sequence` using the function/operator `op` and initial value `initial`. `fold-right` should start with the initial value and accumulate the result from the last item in the list to the first. The procedure is named "fold-right" because it combines the first element of the sequence with the result of combining all the elements to the right. For example:

   ```
   (fold-right + 0 (list 1 2 3 4 5))
   15
   (fold-right * 1 (list 1 2 3 4 5))
   120
   (fold-right cons '() (list 1 2 3 4 5))
   (1 2 3 4 5)
   ```

   (b) Define a SCHEME function, named `(fold-left op initial sequence)`, which is another accumulate procedure except that `fold-left` applies the operator to the first element of the list first and then the next until it reaches the end of the list. That is, `fold-left` combines elements of `sequence` working in the opposite direction from `fold-right`.

```
(fold-left + 0 (list 1 2 3 4 5))
15
(fold-left * 1 (list 1 2 3 4 5))
120
(fold-left cons '() (list 1 2 3 4 5))
(5 4 3 2 1)
```

(c) Complete the following definition of my-map below which implements the `map` function on lists using only the `fold-right` function.

```
(define (my-map p sequence)
(fold-right (lambda (x y) <??>) '() sequence))
```

(d) Complete the following definition of my-append below which implements the `append` function on lists using only the `fold-right` function.

```
(define (my-append seq1 seq2) (fold-right cons <??> <??>))
```

(e) Complete the following definition of my-length below which implements the `length` function on lists using only the `fold-right` function.

```
(define (my-length sequence) (fold-right <??> 0 sequence))
```

(f) Complete the following definition of `reverse-r` below which implements the `reverse` function on lists using only the `fold-right` function.

```
(define (reverse-r sequence)
(fold-right (lambda (x y) <??>) '() sequence))
```

(g) Complete the following definition of `reverse-l` below which implements the `reverse` function on lists using only the `fold-left` function.

```
(define (reverse-l sequence)
(fold-left (lambda (x y) <??>) '() sequence))
```

(h) [SICP EXERCISE 2.34] 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

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

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 . Use the `fold-right` function to define a SCHEME procedure, named `(horner-eval x coefficient-list)` which evaluates a polynomial using Horner's rule. Assume that the coefficients of the polynomial are arranged in a list, from a_0 through a_n .
For example, to compute $1 + 3x + 5x^3 + x^5$ at $x = 2$ you would evaluate

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

4. **Truncatable Primes** You may enjoy this Numberphile video on Truncatable Primes.
   (a) **Left Truncatable Primes** In number theory, a left-truncatable prime is a prime number which, in a given base, contains no 0, and if the leading ("left") digit is successively removed, then all resulting numbers are prime. For example, 9137, since 9137, 137, 37 and 7 are all prime.
      i. Define a SCHEME procedure, named `(left-truncatable-prime? p)`, that takes one integer argument, `p`, and evaluates to true (`#t`) if the integer `p` is a left-truncatable prime and false (`#f`) otherwise.
      ii. Define a SCHEME procedure, named `(nth-left-trunc-prime n)`, that takes one argument, `n`, and uses the `find` function you write in Lab 6 and `(left-truncatable-prime? p)` to return the n^{th} left-truncatable prime number.
   (b) **Right Truncatable Primes** A right-truncatable prime is a prime which remains prime when the last ("right") digit is successively removed. 7393 is an example of a right-truncatable prime, since 7393, 739, 73, 7 are all prime.
      i. Define a SCHEME procedure, named `(right-truncatable-prime? p)`, that takes one integer argument, `p`, and evaluates to true (`#t`) if the integer `p` is a right-truncatable prime and false (`#f`) otherwise.
      ii. Define a SCHEME procedure, named `(nth-right-trunc-prime n)`, that takes one argument, `n`, and uses the `find` function you write in Lab 7 and `(right-truncatable-prime? p)` to return the n^{th} right-truncatable prime number.
   (c) **Two-Sided Primes** There are 15 primes which are both left-truncatable and right-truncatable.
      i. Define a SCHEME procedure, named `(two-sided-prime? p)`, that takes one integer argument, `p`, and evaluates to true (`#t`) if the integer `p` is both a left-truncatable prime and a right-truncatable prime, and false (`#f`) otherwise.
      ii. Define a SCHEME procedure, named `(nth-two-sided-prime n)`, that takes one argument, `n`, and uses the `find` function you write in Lab 7 and `(two-sided-prime? p)` to return the n^{th} two-sided prime number.

Recall the conventions we have adopted in class for maintaining trees. We represent the empty tree with the empty list `()`; a nonempty tree is represented as a list of three objects

```
(value left-subtree right-subtree)
```

where `value` is the value stored at the root of the tree, and `left-subtree` and `right-subtree` are the two subtrees. We introduced some standardized functions for maintaining and accessing this structure, which we encourage you to use in your solutions below.

```
(define (make-tree value left right) (list value left right))
(define (value tree) (car tree))
(define (left  tree) (cadr tree))
(define (right tree) (caddr tree))
```

5. *SICP Exercise 2.31.* Define a SCHEME procedure, named `tree-map`, which takes two parameters, a tree, `T`, and a function, `f`, and is analogous to the `map` function for lists. Namely, it returns a new tree T' with a topology identical to that of `T` but where each node $n \in T'$ contains the image under f of the value stored in the corresponding node in T. For instance, if the input tree is shown in Example 1 and the function `f` is $f(x) = x^2$. then the tree returned by `tree-map` is shown in Example 2.

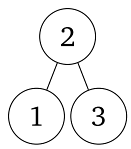

Figure 1: `T`, the tree passed to `tree-map`

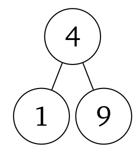

Figure 2: T', the tree returned by `tree-map` when passed `T` and the squaring function as parameters.

6. Define a SCHEME procedure, named `tree-equal?`, which takes two trees as parameters and returns `#t` if the trees are identical (same values in the same places with exactly the same structure) and `#f` otherwise. Use the SCHEME `eq?` function to test equality for the values of the nodes.

7. Define a SCHEME procedure, named `(tree-sort l)`, which takes a list of numbers and outputs the same list, but in sorted order. Your procedure should sort the list by

   1. inserting the numbers into a binary search tree and, then,
   2. extracting from the binary search tree a list of the elements in sorted order.

   To get started, write a procedure called `(insert-list L T)` which takes a list of numbers L and a binary search tree T, and returns the tree that results by inserting all numbers from L into T. (Place the argument L first, so a call to your function should have the form `(insert-list L T)`, where L is a list and T is a (perhaps empty) binary search tree.)

   Then write a function called `sort-extract` which takes a binary search tree and outputs the elements of the tree in sorted order. (We did this in class!)

   Finally, put these two functions together to achieve `(tree-sort l)`. (Note, all three of these functions will be graded, so your solutions must consist of three top-level functions, `insert-list`, `sort-extract`, and `tree-sort`.)