CSE1729 - Introduction to Programming	March 21, 2022
Problem Set 7	

1. Define a SCHEME function, `merge`, which takes two lists ℓ_1 and ℓ_2 as arguments. Assuming that each of ℓ_1 and ℓ_2 are *sorted* lists of integers (in increasing order, say), `merge` must return the sorted list containing all elements of ℓ_1 and ℓ_2.

   To carry out the merge, observe that since ℓ_1 and ℓ_2 are already sorted, it is easy to find the smallest element among all those in ℓ_1 and ℓ_2: it is simply the smaller of the first elements of ℓ_1 and ℓ_2. Removing this smallest element from whichever of the two lists it came from, we can recurse on the resulting two lists (which are still sorted), and place this smallest element at the beginning of the result.

2. There is a sorting algorithm that one can build from the SCHEME function `merge`. It is aptly named *merge sort*. Its architecture is based on the simple principle known as "divide and conquer" (like `quickSort`, covered in class). It works as follows: given a list ℓ, split the list into two sub-lists ℓ_1 and ℓ_2 of approximately the same length (a difference of 1 at most) such that all elements of ℓ appear in either ℓ_1 or ℓ_2. For instance, a list $\ell =$`(1 6 7 3 9 0 2)` could be split into $\ell_1 =$`(1 7 9 2)` and $\ell_2 =$`(6 3 0)`. Then one can recursively sort ℓ_1 and ℓ_2 to obtain sorted versions ℓ_1' and ℓ_2' and *merge* them to recover a fully sorted list. Write a SCHEME function `(mergeSort l)` which, given an unsorted list ℓ of integers, returns a sorted version of ℓ's content. **Hint:** writing a helper function to carry out the splitting would be a wise first step. Hiding that helper function in the bowels of `mergeSort` would be even better! A whimsical illustration of mergeSort is shown in Figure 1.

3. 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.

4. Armed with `ins`, you are now ready to implement another sorting algorithm known 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)
   ```

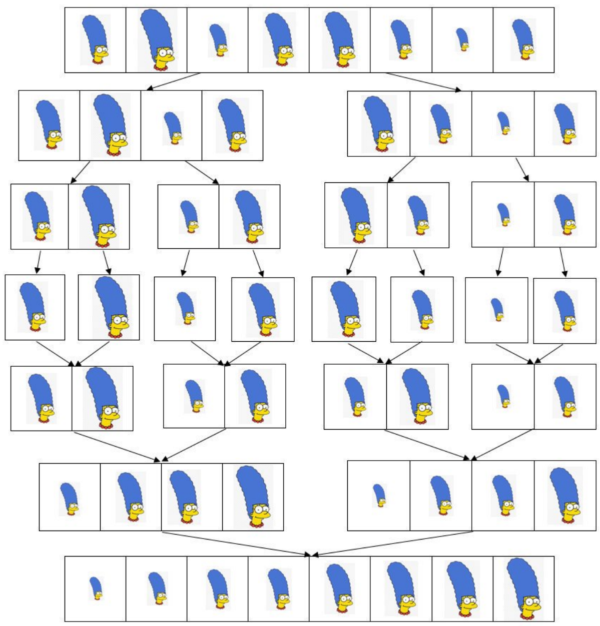

Figure 1: `mergeSort` at work!

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).

5. (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 (lambda (x y) (cons y x)) '() (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))
```

6. **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 wrote 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 wrote in Lab 6 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 wrote in Lab 6 and (`two-sided-prime? p`) to return the n^{th} two-sided prime number.