Problem 1. Work through the file *refresher.hs*, which revisits the basic syntax of the Haskell language.

Problem 2. Open the file *basicFuncs.hs*.

1. Write a function

   `palindrome :: String -> Bool`

   that determines if a given string is a palindrome (a palindrome is a string that is the same if we reverse the order of its letters. The function

   `reverse :: [a] -> [a]`

   reverses a list. Recall that `String` is a synonym for `[Char]`

2. Write a function `capitalizeSentence :: String -> String` which capitalizes each word in a string. You may find the following functions helpful:

   `capitalize :: String -> String` (Takes as input a string and returns the same string with the first character written in upper case.)

   `words :: String -> [String]` (Splits a string into its constituent words, returned as a list of strings.)

   `unwords :: [String] -> String` (Takes a list of words and returns them as a single whitespace-separated String)

   `map :: (a -> b) -> [a] -> [b]` (Takes as input a function and a list and returns the result of applying the function to each element of the list. Take some time to understand the type signature.)

3. Write a function `removePunctuation :: String -> String` that removes punctuation and any other non-alphanumeric characters from a string. You will need to use the function

   `filter :: (a -> Bool) -> [a] -> [a]`

   This function takes a predicate (a function which returns a boolean value) and a list and returns a new list containing only the elements of the original list for which the predicate returns `True`. You can test if a character is alphanumeric with the function `isAlphaNum :: Char -> Bool`

4. Write a function `maxInt :: [Int] -> Int` which returns the maximum element of a list of positive Ints. Use the function

   `foldr :: (a -> b -> b) -> b -> [a] -> b`

   which takes a binary function(a function with two arguments), a default value, and a list and returns the result of traversing the list, sequentially applying the binary function. As an example, consider a function that sums the elements of a list of integers

```
sum ::  [Int] -> Int
sum xs = foldr (+) 0 xs
sum [1,2,3] = foldr (+) 0 [1,2,3] = 1 + (2 + (3 + 0))
```

Note that the default value is used on the right hand side of the final addition at the end of the list.

To find the maximum element in your list you should fold the function

`max ::  Ord a => a -> a -> a` over the list. You can use `0` as the default value, since it is smaller than any positive integer.

5. Write a function that enumerates a list - that appends to each element its position in the list. Haskell has a 2-tuple type `(a,b)` which can store pairs of values. Use the functions

   `zip ::  [a] -> [b] -> [(a,b)]` which takes two lists and pairs up corresponding elements as follows.

   ```
   zip ['a', 'b', 'c'] [1,2,3] = [('a', 1), ('b', 2), ('c', 3)]
   ```

   If the lists have different lengths, `zip` terminates after the shorter list has been exhausted.

   To get the list of integers from a to b, we can use the shorthand `[a..b]`. You may find it useful to get the length of a list with the function `length ::  [a] -> Int`. (Optionally, write your own length function). Alternatively, you may be able to avoid using `length` altogether (Hint: Haskell lists can have infinite length).

6. Write a function to encrypt a binary message `encryptBits ::  [Int] -> [Int] -> [Int]` that takes a message and key (as lists of 0s and 1s) and encrypts the message by XORing the first bit of the message with the first bit of the key, the second bit of the message with the second bit of the key, and so on.

   A function `xorInt ::  Int -> Int -> Int` is defined for you. Use the function

   `zipWith ::  (a -> b -> c) -> [a] -> [b] -> [c]` which acts like zip, but instead of constructing tuples takes a function as its first argument and applies it to each pair of elements.

   Write a function to decrypt messages encrypted in this way.

Problem 3. Open the file *dataX.hs*. This file introduces some basic functional Data Structures.

List

The most frequently used of these is the List. A List is an ordered collection of elements. Lists are homogeneous (all elements of a list are the same type). Like all Haskell values, Lists are immutable (cannot be changed). The List datatype is parametric, meaning it takes a parameter, which can be any type. This means we can have lists of integers `[Int]`, booleans(`[Bool]`, characters `[Char]` a.k.a. `String`. We can take this even further - we can have lists of lists of characterws `[[Char]]` a.k.a. a list of Strings `[String]`.

There are two types of List:

1. The empty List `[]`
2. Nonempty Lists - these take the form `x : xs`, read as `x` cons `xs`. Think of cons as being like a function `(:) :: a -> [a] -> [a]` that takes a value and a list, and returns a new list with the value tacked on to the front of the old list.

Our usual notation for list is actually syntactic sugar on top of this definition. For example, `[1,2,3]` is syntactic sugar for `1 : 2 : 3 : []` and `"Hello` is syntactic sugar for `'H' : 'e' : 'l' : 'l' : 'o' : []`. For obvious reasons we shall stick with the sugar. By convention variables denoting lists have the suffix -s, variables denoting lists of lists have the suffix -ss and so on.

When writing functions on Lists, we can thus pattern match on these two cases. Consider the following function which determines if a List is empty:

```
empty :: [a] -> Bool
empty [] = True
empty (x:xs) = False
```

To define a function on the Lists, it suffices to cover these 2 cases. For lists which are not empty, we can return the first element with the function `head :: [a] -> a` and the rest of the list with the function `tail :: [a] -> a`. What happens if we try to call `head` or `tail` on the empty list?

Lists can be appended together with the function `(++) :: [a] -> [a] -> [a]` (pronounced append). You have already seen other operations on Lists(reverse, length, map, filter, fold, zip). If you need to access specific elements or sublists, you can use

`(!!) :: [a] -> Int -> a` (called index) (`xs !! n`) returns the nth element of xs

`take :: Int -> [a] -> [a]`
`take n xs` returns the first `n` elements of `xs`

`drop :: Int -> [a] -> [a]`
`drop n xs` returns `xs` without its first `n` elements

You can split a list at its *n-th* element using the function
`splitAt :: Int -> [a] -> ([a],[a])`.
Write your own definition of this function (call it splitAtX) using `take` and `drop`

The function `concat :: [[a]] -> a` takes a lists of lists and combines them into a single list, e.g.

`concat [[1,2],[3,4] = [1,2,3,4]`
`concat [['a','b'],['c','d'] = ['a','b','c','d']`
`concat ["Hello ", "world", "!"]` = "Hello world!"
Write your own version of `concat`, (call it `concatX`).

Maybe

Computations often fail. A search does not find its target. Validators receive invalid data. Situations like this are not serious enough that we need to throw an exception, but they do impede the proper flow of data. Functions like this, which may return a value or not have type `Maybe`. Let's look at the type's source code:

```
data Maybe a = Nothing | Just a
  deriving (Eq, Ord)
```

Like `List`, `Maybe` is parametric, meaning it we can have `Maybe Int`, `Maybe Bool`, `Maybe [Int]`, `Maybe (Maybe Int)` or any other `Maybe` type we want. A variable of type `Maybe a` be either `Nothing` or `Just a`, e.g.

```
maybe5 :: Maybe Int
maybe5 = Just 5

maybe6 :: Maybe Int
maybe6 = Nothing
```

To see where this might be useful, consider the `maxInt` function you wrote earlier. One drawback was that it returned 0 on the empty list, even though 0 was not actually present. We could write another function

```
maxIntMaybe :: [Int] -> Maybe Int
maxIntMaybe xs = case (maxInt xs) of | 0 -> Nothing
                                     | x -> Just x
```

which more accurately reflects the behavior we wanted - finding the maximum element in the list.

Write a function `addMaybe :: Maybe Int -> Maybe Int -> Maybe Int` which adds two integers, but returns `Nothing` if either of its arguments is `Nothing`. Use pattern matching.

(Optional) There are certain library functions that operate on the `Maybe` type. Look up the behavior of the following: `maybe, isJust, isNothing` and implement them, and any other such functions you may find.

Stack

You now have all the ammunition you need to implement the `Stack` ADT in Haskell.

Uncomment out the type signature for `Stack`. It is parametric, like `List` and `Maybe`, so we can have a stack of any type, and has one constructor `Stack` which contains an `Int` (which denotes the current size of the stack) and a `List` of the same type as the stack.

Define the following functions:
`newStack` which returns a new, empty stack.
`push` which returns the result of adding an element to the stack
`pop` which returns a tuple consisting of (i)the stack resulting from removing the most recently pushed elementand (ii) that element. Since this operation is invalid if the stack is empty, this should have a `Maybe` type.
`size` which returns the size of the stack.
`empty` which returns true if the stack is empty and false otherwise.
`peek` which returns the element at the top of the stack (infer an appropriate type from the information above).

Problem 4. Computer Science is the study of the recursive functions - those functions whose outputs can be calculated from their inputs by following a finite procedure. However even a finite procedure may reference itself:

1. Walk down the road
2. Take a right on Main Street
3. Walk down the Avenue and onto the High Street
4. If you get lost, retrace your steps and start this procedure again from the beginning.
5. The Benab will be on your left.

These five steps may take a long time to complete if I were constantly getting lost on Main Street and having to retrace my steps and start the procedure over. In this way following a finite procedure may result in the execution of an arbitrarily large or potentially infinite set of instructions.

Note that on line 4, the procedure references itself. This is a defining feature of recursive procedures, and means they can offer very simple descriptions of how to solve a problem. Consider the function `length :: [a] -> Int` we have been using - a procedure for computing the length of a `List` using pattern matching can be described as follows:

- The length of the empty list is 0.
- The length of a nonempty list is 1 + the length of its tail.

We can translate this description directly into code (see *recursion.hs*

```
length :: [a] -> Int
length [] = 0
length (x : xs) = 1 + (length xs)
```

Translate the following description of the `reverse` function to code:

- The reverse of the empty list is the empty list.
- To reverse a nonempty list, reverse its tail, then append its head to the end.

Implement the insertion sort algorithm from the following description:

- An empty list is sorted.
- A list with only one element is sorted.
- If a list has more than one element, sort its tail, then insert the element at the head into the appropriate position.

We leave it to you to deduce how to insert the element at the appropriate position (Hint: Take advantage of the fact that the tail is sorted).

Note that when reasoning about recursion, we take advantage of properties of data that does not yet exist. Why can we do this?

Note that the above examples have several things in common

- Our function is defined non-recursively for at least one value, called the base case.
- Recursive calls are made on data more similar to the base case than our input.

Neither of these conditions is necessary for recursion, but together they guarantee that a recursive function will terminate. Every time a recursive fall is made, the input becomes more similar to the base case, until finally the input is the base case, and no further recursive calls need to be made.

Implement the remaining functions on lists from *basicFuncs.hs* - append, concat, drop, take, map, filter, zip, zipWith using recursion and pattern matching.

Bonus Exercise:

Implement the library functions

```
and :: [Bool] -> Bool
or :: [Bool] -> Bool
any :: (a -> Bool) -> [a] -> Bool
all :: (a -> Bool) -> [a] -> Bool
```

using recursion. Then implement them using your basicFuncs (map, foldr, etc.)

Observe the relationship between foldr and recursion, and use that observation to implement foldr using recursion.

Recursion is not limited to lists. Consider the parity problem - how do we determine whether a nonnegative integer is odd or even? Plainly if an Int is odd, then it is not even. For our base case, we can safely assume that 0 is even. Finally, for our recursive case, we notice that if the number preceding an even number is odd. Use the above information to write a recursive function

`even :: Int -> Bool` that determines whether a nonnegative integer is even.

Write a recursive definition of the factorial function.