SET07106 - Exercise sheet 5
Maybe

NOTE: This practical introduces some new Haskell syntax, but in a very focussed way. It should be read in conjunction with one of the Haskell textbooks found on Moodle, or with this online version of Learn You a Haskell, in particular the sections on `Maybe`.

The type signatures of all of the functions you will need to define already exist inside the file `exercise5.hs`. You just need to fill in the gaps then uncomment the function. The gaps for some functions are larger than for others.

As we saw in the lecture, the `Maybe` type deals with speculative calculations, with the value of `Nothing` signifying that the calculation, in some sense, failed, whereas `Just x` signifies that the calculation succeeded and gave the value of `x`.

We also saw that it is useful to write an unsafe version of a function, and then use the `Maybe` type to wrap that function up. Most of what you do this week will involve using functions from previous weeks, and producing safe versions of them.

Exercise 1 *Write a function called `lastSafe`, which takes a list of type `[a]` and returns a value of type `Maybe a`. If the list is empty, then you should return `Nothing`. Otherwise, you should return the last element of the list inside the `Just` constructor.*

Exercise 2 *Write a function called `biggestPrime` which takes a list of type `[Int]` and returns a value of `Maybe Int`. It should return the biggest prime number in the list, if one exists. For this exercise, you may assume the input list is ordered from smallest to biggest.*

Exercise 3 *Write a function called `safeMax` which takes a list of type `[Int]` and returns a value of `Maybe Int`. It should return the biggest integer in the list, if the list is non-empty. You cannot assume that the list is ordered.*

Exercise 4 *Write a function called `biggestPrime2` which is the same as `biggestPrime` except that it can take an unordered list as an input.*

We saw in the lectures the safe function `find`, which returned an element from a list if that element existed. You are going to use a similar function to deal with the problem of witnesses and counterexamples from week 2 (predicate logic).

Exercise 5 *Write a function called `thereExistsWitness`, which takes a propositional function of type `a -> Bool`, a list of type `[a]`, and returns a value of type `Maybe a`. The behaviour of the function is that it should return a witness that makes the propositional function true. For example, `thereExistsWitness even [1..10]` should return `Just 2`, and `thereExistsWitness isPrime [8,9,10]` should return `Nothing`.*

Exercise 6 *Write a function called* `forAllCounterexample`*. It should be similar to the previous exercise, except that it returns an element which makes the propositional function false, if one exists.*

Exercise 7 *Write a safe version of* `findIndex`*. This takes a list* `ys` *of type* `[a]` *and an element* `x` *of type* `a`*, and returns the index of* `x` *in* `ys` *if it exists.*

We can also do pattern matching on a `Maybe` type itself. In lots of ways, the `Maybe` type is the same as the list type: each has a base constructor `[]` and `Nothing` which are the base cases for *every* kind of list, and each has a constructor `x:xs` and `Just x`. These two constructors *fix* the type, because `x` needs to be a particular kind of object. If we then wanted to, we can write functions which unpack `Just` types to get at the values.

```
weirdPlus ::  Maybe Int -> Maybe Int -> Int
weirdPlus Nothing _ = -1
weirdPlus _ Nothing = -1
weirdPlus (Just x) (Just y) = x+y
```

In practice, you would almost never do this. If an input to a function was of type `Maybe a`, then the output should be of type `Maybe b` for suitable `a` and `b`. Why? Because the input was generated by a safe process, so we do not want to introduce potentially harmful behaviour. Instead, we unpack and do recursion inside the function.

We saw in the lecture the pattern for unpacking `Maybe` types inside a recursive function:

```
safeSum ::  [Int] -> Maybe Int
safeSum [] = Nothing
safeSum [x] = Just x
safeSum (x:xs) = case (safeSum xs) of
    Just n -> Just (n+x)
    Nothing -> Nothing
```

Exercise 8 *Without using a helper function (i.e. using the pattern above), write the function* `safeLast`*, which is like* `lastSafe` *from exercise* 1*.*

Exercise 9 *(Harder) Without using a helper function (i.e. using the pattern above), write the function* `removeLastSafe`*, which takes a list of type* `[a]` *and returns a value of* `Maybe [a]`*. The function removes the last element from a non-empty list. If you are having trouble, try writing the non-safe version first to see how it works.*