Exercise File 1

```
module E1 where

import Data.List
```

Exercise 1.1

Define your own versions of `map`.

First, in two lines with pattern matching:

```
myMap :: (a -> b) -> [a] -> [b]
myMap f []     = undefined
myMap f (x:xs) = undefined
```

Second, in one line with a list comprehension:

```
myOtherMap :: (a -> b) -> [a] -> [b]
myOtherMap f xs = undefined
```

Do the same for `filter`. Look up `zip` in hoogle and implement `myZip`.

Exercise 1.2

Find the larger number in a tuple. There are at least three ways to do this: with guards, with `case compare ... of` and with `if ... then ... else`.

```
getLarger :: (Integer,Integer) -> Integer
getLarger (x,y) = case compare x y of
                    LT -> y
                    EQ -> y
                    GT -> x
```

Interruption: What is (the type of) `undefined`? Did you try using/evaluating it? Also look up `error` on hoogle and in your Haskell book of choice.

Exercise 1.3

By natural number we mean elements of `[0..]`. Your task is to translate the following definition of being a prime number to Haskell:

An integer n is prime iff (i) it is a natural number and (ii) for all natural numbers d with $1 < d < n$ we have that d does not divide n.

First we need the `divide` relation. Hint: Look up `rem` on hoogle or with `:t rem` in ghci.

```
divide :: Integer -> Integer -> Bool
divide n m = undefined
```

Next, try to translate the definition, using the functions `&&`, `all` and `not`.

```
isPrime :: Integer -> Bool
isPrime n = undefined
```

Exercise 1.4

Recall the following definitions for propositional logic.

```
data Form = P Integer | Neg Form | Conj Form Form
  deriving (Eq,Ord,Show)
type Assignment = Integer -> Bool

satisfies :: Assignment -> Form -> Bool
satisfies v (P k)      = v k
satisfies v (Neg f)    = not (satisfies v f)
satisfies v (Conj f g) = satisfies v f && satisfies v g
```

Your task is to add disjunction, implication, and $\top$ to this implementation. Can you think of different approaches?

Can you make conjunction and disjunction multi-ary, so that we can for example write `Conj [f,g,h]`?

Our next goal is to check whether a formula is valid, i.e. true in all assignments. First, write a function that collects all variables in a given formula:

```
variablesIn :: Form -> [Integer]
variablesIn = undefined
```

It is slightly annoying that `Assignment`s are functions, because we do not have a way to `show` them. But intuitively we know that instead of functions from numbers to booleans we can use sets of numbers. Can you change the code to use `type Assignment = [Integer]` instead?

Then write a function that generates all assigments for a given list of variables:

```
allAssignmentsFor :: [Integer] -> [Assignment]
allAssignmentsFor = undefined
```

Given this, how can we check the validity of a formula?

```
isValid :: Form -> Bool
isValid f = undefined
```

Here are some tests that your implementation should pass. Add some more!

```
tests :: [Bool]
tests = [ not . isValid $ P 1
        ,       isValid $ Neg (Conj (P 1) (Neg (P 1))) ]
```

Bonus questions How would you implement `allAssignmentsFor` for the original `type Assignment = (Integer -> Bool)`? Which type is easier to use?

More Exercises

You can find more introductory Haskell exercises in the books The Haskell Road and Programming in Haskell, on the website exercism.io and at Project Euler (try problems 9, 10 and 49).

Homework

1. Go through the lecture slides and find at least one thing you did not understand or would like to know more about. Ask it at the next lecture.
2. Finish this file as far as you can.
3. Get some popcorn ready and watch this talk:

   Simon Peyton-Jones: *Escape from the ivory tower: the Haskell journey*
   https://www.youtube.com/watch?v=re96UgMk6GQ