Exercises 1

```
module E1 where

import Data.List
```

Exercise 1.0

If not done already, set up Haskell on your computer. See the course website and email for links to the tools. If you are unsure what to use, the recommended editor is Visual Studio Code (also called VS Code). Make sure to also install the Haskell exteions to get compiler errors and warnings directly in the editor. For larger projects later you should also install `stack`.

Exercise 1.1: Types

What is the type of the following expressions? Write down your own answer, then check it using `:t` in GHCI.

```
"Hello"

("Hello", 42::Int, "Byebye")

["Hello", 42::Int, "Byebye"]

(+ (1::Int))

show . (+(100::Int)) . (*(2::Int))

( \x y -> x ++ y ++ x )
```

Exercise 1.2: map, filter, zip

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 | undefined ]
```

Do the same for `filter`.

```
myFilter :: (a -> Bool) -> [a] -> [a]
myFilter _ []     = undefined
myFilter p (x:xs) = undefined
```

Hint: You might want to split the last line into two cases, using `if ... then ... else ...` or guards.

```
myOtherFilter :: (a -> Bool) -> [a] -> [a]
myOtherFilter p xs = [ undefined | undefined ]
```

Look up `zip` at https://hoogle.haskell.org/ and implement `myZip`. Hint: Use three lines with different patterm matching.

```
myZip = undefined
```

Note that on Hoogle you can also search by types. For example, search for "`(b -> c) -> (a, b) -> (a, c)`". What do you expect this function to do? implement it yourself!

```
mystery :: (b -> c) -> (a, b) -> (a, c)
mystery = undefined
```

Exercise 1.3

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) = undefined
```

Interruption

What is (the type of) `undefined`? Did you try using/evaluating it? Also look up the type of `error`. What is its type and why?

Recall that you can look up types with `:t something` in `ghci`. Note: `error` on https://hoogle.haskell.org/ has a much scarier type which we ignore here.

Exercise 1.4: Primes

By natural number we mean elements of `[1..]`. We now 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 $2 \leq d \leq n-1$ we have that d does not divide n.

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

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

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

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

If your function is correct then you can get the first 20 primes using this query in GHCI:

```
GHCI> take 20 (filter isPrime [1..])
```

Note that we can also write this as:

```
GHCI> take 20 $ filter isPrime [1..]
```

Exercise 1.5: Propositional Logic

Recall the following definitions for propositional logic from the lecture.

```
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 we need a function to collect all variables in a given formula:

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

Now it is slightly annoying that `Assignments` 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. Change the code to use `type Assignment = [Integer]` instead, and repair all the resulting errors! Hint: hoogle `elem`.

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)))
        -- add your tests here
        ]
```

Bonus question

How would you implement `allAssignmentsFor` for the original `type Assignment = (Integer -> Bool)`?

Which definition of `Assignment` is easier to use?

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

Even More Exercises

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