Exercise 1

NTNU

TDT4165 høst 2018

1 Collaboration

The exercises and project will be carried out as group assignments.
The reason for this is:

- Increase the learning effect, because you learn faster when you have someone to discuss problems with.
- Ensure completion, because it is more committing to cooperate.
- Increase course efficiency, because a learning assistant can spend more time on groups than individuals.

In order to ensure that group participants work together to get the most out of individual and shared learning outcomes, we want you to write a short answer to the following questions:

- How do you plan your work?
- How do you distribute tasks?
- How do you ensure that an effort is being made?
- What do you do if your cooperation does not work?

2 Installation

Download and install the Haskell platform from `https://www.haskell.org/platform/`. Your operating system will be automatically chosen for you, but if it isn't, you can find it in the list.

2.1 Linux

```
$ sudo apt-get install haskell-platform
```

2.2 Windows

Download and install the .exe file from the website above. Choose "Download Core 64-bit" (or 32-bit, if you're on a 32-bit system).

3 The Haskell Tool Stack

In this course we will be using Stack to build our projects. This is because Stack offers an easy way to control dependencies and versions of the GHC compiler, making sure our builds go as smoothly as possible.

3.1 Using the REPL

To create your first "Hello World!" program in Haskell, open up your favorite text editor and copy the code below.

```
main :: IO ()
main = putStrLn "Hello World!"
```

Save the text file as "hello.hs". Then open a terminal window, navigate to the folder you saved your code in and type

```
$ stack ghci
```

If you now get a warning that you don't have any package.yaml / *.cabal files, that's perfectly fine. We don't need that at this point.

GHCi, the Glasgow Haskell compiler interactive environment, will now be loaded. You should see something like this:

```
Loaded GHCi configuration from /tmp/ghci3824/ghci-script
Prelude>
```

To compile and load our program, type:

```
Prelude> :l hello
```

If the compilation goes well, which it should with our simple "Hello World!", the compiler will let you know.

```
[1 of 1] Compiling Main             ( hello.hs, interpreted )
Ok, one module loaded.
*Main>
```

Now we call the function we've written.

```
*Main> main
Hello World!
```

Our "Hello World!" was a success!

3.2 Creating a new project

The approach above works wonders for small programs, but if we want to write a bigger project, introduce dependencies and share our code with others, we are better off creating a new project. If you are still in interactive mode, run

```
*Main> :q
```

GHCi will exit and we're back in our folder. We will now tell Stack that we want a new project.

```
$ stack new hello
```

Stack will create a new project in the directory we are presently at, in a folder named "hello". If we navigate into the folder, we see that several files and folders have been created, but the two files we will focus on, for now, is app/Main.hs and src/Lib.hs. Opening src/Lib.hs, we will see something like this

```
module Lib
  ( someFunc
  ) where

someFunc :: IO ()
someFunc = putStrLn "someFunc"
```

This means we have a **module** called Lib, where we export a function *someFunc*, so other modules are able to import and use it. We can create other functions or types and export them too, separating each export by a comma.

```
module Lib
  ( someFunc
  , hello
  ) where

someFunc :: IO ()
someFunc = putStrLn "someFunc"

hello :: IO ()
hello = putStrLn "Hello World!"
```

In app/Main.hs our code looks like this:

```
module Main where

import Lib
```

```
main :: IO ()
main = someFunc
```

We have a module named Main that imports our module Lib and a main function. This is the entry point to our project. Since we would like our project to display "Hello World!" and not "someFunc", we replace the occurrence of someFunc in main with *hello*. We can now tell stack to build our project.

```
$ stack build
```

After a successful build, we can enter the interactive environment and call our main function, like we did in 2.1.

4 First examples

4.1 Build the exercise project

Navigate to the folder **ex1code/** and tell Stack to build the pre-written project. Stack will then download dependencies and the right version of GHC according to the specifications in our *.yaml and *.cabal files.
NOTE: This will take a while the first time you do it, but will be faster the next time you build the project. It will also take less time for the rest of the exercises in the course. If you have already downloaded and installed an image, for example lts-11.15, it will not get downloaded again, Stack will reuse the old copy for the new project. Isn't it amazing?

4.2 Complete the functions

The exercise project has several functions, most of their type signatures are defined. Do not change them. Complete the tasks in the project skeletons, located in the file **src/Lib.hs**. Wherever you encounter "undefined", you replace it with valid code. Test your functions by running this command:

```
\$ stack test
```

4.3 Fizzbuzz

Fizzbuzz is an iconic programming exercise intended to demonstrate basic knowledge of a language. It has the following rules:

- Print the integers from 1 to 100, each number on a new line.
- If the number is divisible by 3, print "Fizz".
- If the number is divisible by 5, print "Buzz".
- If the number is divisible by 3 and 5, print "FizzBuzz"

- If the number is not divisible by 3 or 5 (or both), print the number.

Create a list **fizzbuzz :: [String]** where the four last rules are applied. A function that prints each element on a new line, is already provided in src/Lib.hs. Your assignment is to create the list that *printFizz* prints.
Hint: Create a helper function, either inside of **fizzbuzz** using the **keyword where**, or outside of it.
Hint 2: Prelude is a set of modules that gets imported by default, unless we explicitly say otherwise. In Prelude, you will find the function **show :: Int -> String**, which will come in handy to complete this task.