Week 1 - Preparation

Introduction

Starting & using GHCi

The interactive Haskell environment GHCi can be started by executing the `ghci` command in a terminal window. Haskell programs are stored in files with the `.hs` extension. Familiarise yourself with the following GHCi *directives*.

- To load a program type `:l filename.hs` in GHCi.
- To reload the last file, type `:r` in GHCi.
- To find out the type of an expression, type `:t expr` with `expr` being an expression, e.g. `not True`.
- To find out more information about a type, use `:i type` with `type` being a type, e.g. `Bool` or `[]`.
- To exit GHCi, type `:q`.
- Haskell is indentation sensitive. Use spaces (not tabs!) to indent your program.

HLint

HLint is a tool that helps to improve your code style. It suggests changes for making your code better by indicating redundant brackets, better usage of built-in functions, eta reductions, ... It is a good idea to run HLint on your file after you are done with an exercise. Remember that we pay attention to style when correcting your exam!

```
$ hlint example.hs
example.hs:10:15: Error: Redundant bracket
Found:
  (x)
Why not:
  x

1 suggestion
```

1. Haskell 101

Implement the functions below.
Note that HLint (on E-Systant) may generate warnings, you can ignore these.

- Write a function `double :: Int -> Int`, which doubles its argument.

```
Main> double 3
6

Main> double 21
42
```

- Write a function `myAbs :: Int -> Int`, which computes the absolute value of its argument. Positive numbers thus remain unchanged, while negative numbers become positive.

```
Main> myAbs 0
0

Main> myAbs 42
42

Main> myAbs (-42)
42
```

- Write a function `toFahrenheit :: Float -> Float`, which converts a decimal number from degrees Celsius to degrees Fahrenheit. Use the formula $F = 1.8C + 32$.

```
Main> toFahrenheit 20.0
68.0

Main> toFahrenheit (-3.0)
26.6
```

- Write a function `fizzbuzz :: Int -> String`, which returns `fizz` if its argument is a multiple of 3, `buzz` if its argument is a multiple of 5 and `fizzbuzz` if its argument is a multiple of both 3 and 5. Alternatively, if it's argument is not a multiple of 3 or 5, it should return its argument in text form. Use the function `show` to convert an `Int` value to a `String`.

```
Main> fizzbuzz 2
"2"

Main> fizzbuzz 3
"fizz"

Main> fizzbuzz 4
"4"
```

```
Main> fizzbuzz 5
"buzz"

Main> fizzbuzz 15
"fizzbuzz"
```

2. List Operations - Part 1

Implement the functions below. Note that many of these functions are available in the standard library,[1] but the goal of this exercise is to practice by implementing them from scratch. When writing a recursive function involving lists, put some thought into choosing the right base case.

Recall that the syntax of pattern-matching on a list is as follows (where `x` is the head of the list and `xs` is the tail):

```
function :: [...] -> ...
function []     = ...
function (x:xs) = ...
```

Note that HLint (on E-Systant) may generate warnings, you can ignore these.

- Write a function `count :: [Int] -> Int`, which counts the number of elements in a list.

```
Main> count [1, 2, 3]
3

Main> count []
0
```

- Write a function `myAnd :: [Bool] -> Bool`, which takes as argument a list of booleans and evaluates to `True` if all the elements of the list are `True` and `False` otherwise.

```
Main> myAnd [True, False]
False

Main> myAnd [True, True, True]
True

Main> myAnd []
True
```

[1] For example, see module `Data.List`, which can be found at `http://downloads.haskell.org/~ghc/7.6.3/docs/html/libraries/base/`.

- Write a function `myOr :: [Bool] -> Bool`, which takes as argument a list of booleans and evaluates to `True` if at least one element in the list is `True` and `False` otherwise.

```
Main> myOr [True, False]
True

Main> myOr [False, False, False, False]
False

Main> myOr []
False
```

- Write a function `append :: [Int] -> [Int] -> [Int]`, which takes two lists and computes their concatenation.

```
Main> append [1,2] [3,4,5]
[1,2,3,4,5]

Main> append [] [1,2,3]
[1,2,3]

Main> append [1,2,3] []
[1,2,3]
```

3. Warm up: Algebraic Datatypes

Haskell is famous for its type system, its type checker and its strongly statically-typed compilation process. However, up until now you've only encountered predefined types. Using *algebraic data types* it is possible to define new types yourself.

Defining Algebraic Datatypes

A newly created type has to be *defined* by specifying all possible *(data) constructors*. Each constructor is a function that can be used to create a value of this type. The different constructors are separated by the `|` symbol. The `deriving`-clause is optional. Syntactically, this is done in the following manner:

```
data TypeName = Constructor1 ArgType1 ArgType2 ...
              | Constructor2 ArgType1 ArgType2 ...
              | ...
              | ConstructorN ArgType1 ArgType2 ...
```

For example, a boolean can be either true or false:

```
data Bool = True | False
```

Note: to avoid confusion, we advise you to always pick a different name for the constructor than for the data type. For example:

BAD: `data Age = Age Int`

GOOD: `data Age = MkAge Int`

Define algebraic datatypes (ADTs) to represent the following concepts:

- `Name`: a name is just a `String`.
- `Pair`: a pair consists of two integers (`Int`).
- `Gender`: a gender is either male, female, or other.
- `Person`: a person consists of a name (`Name`), an age (`Int`), and a gender (`Gender`).
- `TestResult`: a result of a test is either a *pass*, along with a grade (`Int`) or a *fail*, along with a list of comments from the teacher. You can use a `String` to represent a comment.

Don't forget to add "`deriving (Show)`" at the end of the datatype definition! The error "No instance for (Show ...) arising from ..." means that you have forgotten to add it.

Using Algebraic Datatypes

- Write a function `stringToGender :: String -> Gender` that returns the correct gender for the given string. If the string is "Male" or "Female" (correctly capitalised), the right constructor of `Gender` should be picked. All other strings are considered to be "Other".
- Write a function `genderToString :: Gender -> String` that converts a gender to a string: "Male", "Female", or "Other".

Examples

```
Main> genderToString (stringToGender "Male")
"Male"
Main> genderToString (stringToGender "Hamster")
"Other"
```

- Write a function `passing :: Int -> TestResult` that creates a passing `TestResult` with the given grade.
- Write a function `failing :: [String] -> TestResult` that creates a failed `TestResult` with the given comments.
- Write a function `grade :: TestResult -> Int` that returns the grade of a `TestResult`. A fail results in 0.
- Write a function `comments :: TestResult -> [String]` that returns the comments of a `TestResult`. A passing result has no comments.

Examples

```
Main> grade (passing 10)
10
Main> grade (failing ["Incorrect datatype syntax"])
0
Main> comments (passing 10)
[]
Main> comments (failing ["Incorrect datatype syntax"])
["Incorrect datatype syntax"]
```