Solution sheet 2

Introduction

Please note that there can be other solutions than those listed in this document.

This is a literate Haskell file which is available as PDF, as well as literate Haskell source code (.lhs extension). If you are looking at the PDF version, you should download the .lhs file, which can be loaded into ghci or compiled with ghc. This document uses ***Markdown*** syntax, compiled to PDF using *Pandoc*.

Everything that is not enclosed in a code block like below is treated as text.

```
-- This is code
main :: IO ()
main = undefined
```

Exercise 1 - Boolean functions

1 - NOT

Using pattern matching, we can replicate the behaviour of `NOT` as seen below.

```
notImpl :: Bool -> Bool
notImpl True = False
notImple False = True
```

Additionally, as the patterns are matched from top to bottom, we can simplify using wildcards (_)

```
notImpl' :: Bool -> Bool
notImpl' True = False
notImpl' _    = True
```

Generally, it is good practice to use wildcards where it does not complicate the other code.

2 - OR

We can replicate the behaviour of `OR` as seen below.

```
orImpl :: Bool -> Bool -> Bool
orImpl True _ = True
orImpl _ True = True
orImpl _ _    = False
```

3 - XOR

Pattern matching is not *free*. This is why it is preferred to only pattern match on the minimum pattern required, as seen in the `XOR` example below.

```
xorImpl :: Bool -> Bool -> Bool
xorImpl a False = a
xorImpl False a = a
xorImpl _ _     = False
```

4 - NAND

Finally, `NAND` can be implemented using only pattern matching, as seen below.

```
nandImpl :: Bool -> Bool -> Bool
nandImpl True True = False
nandImpl _ _       = True
```

Exercise 2 - Simple functions

1 - eeny

```
eeny :: Integer -> String
eeny x = if x `mod` 2 == 0
         then "eeny"
         else "meeny"
```

Quite simple implementation using if..then..else.. notation.

2 - fizzbuzz

If more than one check is desired, one option is to use Guards ("|"). To read more about Guards, check out the chapter "Syntax in functions" of Learn you a Haskell for great good!

```
fizzbuzz :: Integer -> String
fizzbuzz x | x `mod` 3 == 0 && x `mod` 5 == 0 = "FizzBuzz"
           | x `mod` 3 == 0                   = "Fizz"
           | x `mod` 5 == 0                   = "Buzz"
           | otherwise                        = ""
```

Exercise 3 - Recursive functions on integers

Recursion is a huge subject in Haskell, and allows for some really powerful functions.

1 - Sum of numbers

Many possible implementations, below is a simple one.

```
sumTo :: Integer -> Integer
sumTo 0 = 0
sumTo n = if n > 0
  then n + sumTo (n-1)
  else error "sumTo only intended for positive n."
```

Unless a hardware error occurs, the function will only throw the error if initially called with negative `n`.

2 - Binominal coefficients

Guards and pattern matching can be used in recursive functions too.

```
binominal :: Integer -> Integer -> Integer
binominal _ 0 = 1
binominal n k | n == k             = 1 -- Here we use guards
              | k < 0              = 0 -- to check value conditions
              | k > n              = 0
              | k >= 1 && k <= n-1 = binominal (n-1) k +
                                     binominal (n-1) (k-1)
```

We use the pattern `binominal _ 0` to catch all instances where k is 0. Remaining conditions are handled with guards.

3 - power

```
power :: Integer -> Integer -> Integer
power _ 0 = 1 -- Per definition, also base case
power n k = n * power n (k-1) -- Always remember parenthesis
```

Reminder: Any case in a recursive function that does **not** call itself recursively, is called a base case. You should generally always think about the base case before anything else when designing recursive functions.

4 - log2

Nothing too interesting, we can use `div` for integer division when the arguments are integer.

```
ilog2 :: Integer -> Integer
ilog2 1 = 0
ilog2 n | n < 1     = error "invalid argument"
        | otherwise = 1 + ilog2 (n `div` 2)
```

Exercise 4 - Recursive functions on lists

1 - dropZeros

Quite similar to the example given with `flattenMaybe`. Base case will pretty much always be the empty list

```
dropZeros :: [Int] -> [Int]
dropZeros []       = [] -- Base case
dropZeros (0 : xs) = dropZeros xs     -- Drop 0 and continue
dropZeros (x : xs) = x : dropZeros xs -- Keep x and continue
```

2 - flatten

As the type specifies that we will have at most one level of nested lists, we can pattern match on the inner list as well.

We can define flatten:

```
flatten :: [[a]] -> [a]
flatten []               = []
flatten ([] : xs)        = flatten xs -- Get rid of empty lists
flatten ((x : xs) : ys) = x : flatten (xs:ys)
```

This works by extracting elements from the inner list one by one, and then recursively calling flatten on the rest.

3 - twiceAll

Simply put an additional x infront of each x, and continue with the tail of the list.

```
twiceAll :: [a] -> [a]
twiceAll []       = []
twiceAll (x : xs) = x : x : twiceAll xs
```

4 - alternate

We introduce a helper function as this provides a simple way to alternate between two actions.

```
alternate :: [Int] -> [Int]
alternate []       = []
alternate (x : xs) = x : alternate' xs
  where
    alternate' []       = []
    alternate' (y : ys) = (-y) : alternate ys
```

5 - replicateAll

Here, we create a helper function which will keep track of how many times the head is yet to be replicated, before calling itself with original n, and the tail as the list.

```
replicateAll :: Int -> [a] -> [a]
replicateAll _ [] = []
replicateAll 0 _  = []
replicateAll n xs = replicateAll' n n xs
  where
    replicateAll' :: Int -> Int -> [a] -> [a]
    replicateAll' _ _ []       = []
    replicateAll' n 0 (_ : ys) = replicateAll' n n ys
    replicateAll' n k (y : ys) = y : replicateAll' n (k-1) (y : ys)
```

The following is equivalent to previous implementation, as helper functions defined in `where` inherits the variables available when they are called (Due to scopes).

```
_replicateAll :: Int -> [a] -> [a]
_replicateAll _ [] = []
_replicateAll 0 _  = []
_replicateAll n xs = replicateAll' n xs
  where
    replicateAll' :: Int -> [a] -> [a]
    replicateAll' _ []       = []
    replicateAll' 0 (_ : ys) = replicateAll' n ys
    replicateAll' k (y : ys) = y : replicateAll' (k-1) (y : ys)
```

6 - cumulativeSum

We create a helper function which will be able to keep track of the sum so far.

```
cumulativeSum :: [Int] -> [Int]
cumulativeSum []       = []
cumulativeSum (x : xs) = x : cumulativeSum' x xs
  where
    cumulativeSum' :: Int -> [Int] -> [Int]
    cumulativeSum' _ []             = []
    cumulativeSum' prevSum (y : ys) = prevSum + y : cumulativeSum' (y + prevSum) ys
```