1 Problem 2 - Higher-Order functions

(a) Given the function `foldr` as below

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

Define a new function

```
append ::  [a] -> [a] -> [a]
```

That takes two lists as input and returns their concatenations

Svar.

```
append :: [a] -> [a] -> [a]
append = foldr (:)
```

(b) Given the function `foldl` as below:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f v [] = v
foldl f v (x:xs) = foldl f (f v x) xs
```

What does the following expression return?

```
foldl (\x -> \y -> x/2 + y) 4 [2,4,6]
```

Svar. This function would return `9.0`. This is because `foldl` works in the following manner

```
foldl (\x -> \y -> x/2 + y) 4 [2,4,6]
= foldl (\x -> \y -> x/2 + y) 4 [2] = 4
= foldl (\x -> \y -> x/2 + y) 4 [4] = 6
= foldl (\x -> \y -> x/2 + y) 6 [6] = 9
```

2 Problem 3 - Phone Book

In this task, you are asked to implement a phone book in Haskell. Use the following type declarations for the implementation.

```
type Name = String
type Pnum = Integer
type Pbook = Name -> Maybe Pnum
```

Where `Pbook` is the data type of the phone book, `Name` and `Pnum` are the types of names and phone numbers, respectively.

With this phone book, one can *insert* a new entry by providing a name and an 8-digit number, *lookup* the phone number with a given name, and *delete* an entry with a given name. Each name in the phone book should be unique.

(a) Define the function `lookup :: Pbook -> Name -> Maybe Pnum` which returns the corresponding 8-digit number if the input (of type `Name`) exists in the phone book (of type Pbook); otherwise, the function should return an appropriate value.

Svar.

```
lookup :: Pbook -> Name -> Maybe Pnum
lookup b = b
```

(b) Define the function `insert :: Pbook -> Name -> Pnum -> Pbook` which adds an entry associating the input name (of type `Name`) with the input 8-digit number (of type `Pnum`) in the phone book (of type Pbook). If the length of the number is incorrect, the function `insert` should call the function `error :: [Char] -> a`. You do not have to implement the `error` function.

Svar.

```
insert :: Pbook -> Name -> Pnum -> Pbook
insert b n p
  | length (show p) > 8 = error "Invalid phone number format"
  | otherwise = \x -> if x == n then Just p else b x
```

(c) Define the function `delete :: Pbook -> Name -> Pbook` which removes the entry of the given name (of type `Name`) and the corresponding 8-digit number from the phone book (of type `Pbook`).

3 Problem 4 - The game Nim

We consider in this task the simplified version of Nim, a mathematical game of strategy in which two players take turns to remove either 1, 2, or 3 objects from a heap. The player who empties the heap wins. The following example is an illustration that shows how the game, which starts with a heap of 10 objects, is played between two players: Player 1 and Player 2.

A winner in Nim: We assume the players in this game are smart, and therefore Player 1 in the last round in the illustrating example (*) will take two objects but not one from the heap. That is, the player who is going to remove objects from a heap containing only 1, 2, or 3 objects wins the game by default.

3.1 Implementation of Nim

You are going to implement the game Nim with the corresponding interface. The game should start with a heap containing a random number of objects. This random number should be of type Integer and be within the range of 10 to 20. After generating the number, the interface of the game will (i) display the number of objects in the heap, and (ii) ask a player to input the number of objects to be removed from the heap, then (iii) deduct the input number of objects from the current heap. The implementation has to ensure that the player inputs a number between 1 and 3; if the input is out of range, the interface should warn the player about the wrong input and ask for the input again without aborting. The game will continue by repeating (i) – (iii), with an alternating player for each round, until a winner is found (see the winner definition above). The following figure shows how the interface should interact with the players and ultimately find the winner for the illustrating example (*). Note that those lines with a single number are inputs from the players.

By following the given structure of the implementation, you are asked to

(a) Complete the `main` function, in which you have to generate the random number (`randNum`) as the initial number of objects for the heap in the game;

Svar.

```
main :: IO ()
main = do
  gen <- newStdGen
  let (num, gen') = randomR (10, 20) gen :: (Integer, StdGen)
  play num 1

play :: Integer -> Integer -> IO ()
play n p =
  if n < 3
    then do
      putStrLn $ "Player " ++ show p ++ " wins"
    else do
      putStrLn $ "Player " ++ show p ++ "'s turn"
      input <- getLine
      let remove = read input
      if remove > 0 && remove < 4
        then do
          putStrLn $ "Player " ++ show p ++ " removed " ++ show remove ++ " numbers from
              the heap"
          putStrLn $ "There are now " ++ show (n - remove) ++ " numbers in the heap"
          if p == 1
            then play (n - remove) 2
            else play (n - remove) 1
        else do
          putStrLn "Wrong input"
          play n p
```