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)
```

Using the function `foldr`, define a function

`lengthsum :: (Num a, Num b) => [a] -> (b, a)`

That takes a list of numbers as input, then return the length and the sum of the list as a pair.

Svar.

```
lengthSum :: (Num a, Num b) => [a] -> (b, a)
lengthSum = foldr (\n (x, y) -> (1 + x, n + y)) (0, 0)
```

(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
```

Using the function `foldl` define a function

`inList :: (Eq a) => a -> [a] -> Bool`

that takes a value and a list of the same type as inputs, then checks wheter the value is an element of the list.

Svar.

```
inList :: (Eq a) => a -> [a] -> Bool
inList x = foldl (\acc y -> (x == y) || acc) False
```

(c) What do the following expressions return?

(i) `foldr (:) "hello" "world!"` = "world!hello"

(ii)

```
foldl (\xs -> \x -> x:xs) "INF122" "exam" ->
```

2 Problem 3 - An evaluator

Consider the following type declaration:

`data Expr = V Int | M Expr Expr | D Expr Expr`

You are asked to implement an evaluator

```
eval :: Expr -> Maybe Int
```

which evaluates an expression of type Expr defined above

Svar.

```
data Expr = V Int | M Expr Expr | D Expr Expr

div' :: Int -> Int -> Maybe Int
div' _ 0 = Nothing
div' x y = Just (x `div` y)

eval :: Expr -> Maybe Int
eval (V x) = Just x
eval (M r l) =
  case eval r of
    Nothing -> Nothing
    Just x -> case eval l of
      Nothing -> Nothing
      Just y ->
        if x >= 0 && y >= 0
          then Just (x * y)
          else Nothing
eval (D r l) =
  case eval r of
    Nothing -> Nothing
    Just x -> case eval l of
      Nothing -> Nothing
      Just y -> div' x y
```

3 Problem 5 - Input and Output

In this problem, you are not supposed to use any build-in functions in Haskell except `return` and list operators

(a) Given a lsit of `IO`-actions, implement a function

```
toDoList :: [IO a] -> IO [a]
```

which excecutes each of the `IO`-actions in the list and gives back an `IO`-action that returns a list containing the corresponding results in the same order as output.

Svar.

```
toDoList :: [IO a] -> IO [a]
toDoList [] = return []
toDoList (x : xs) = do
  action <- x
  list' <- toDoList xs
  return (action : list')
```

(b) Implement a map function for `IO`-actions

```
mapActions ::  (a -> IO b) -> [a] -> IO [b]
```

which takes a functions of type `a -> IO b`, and then applies this function to each item in an input list of type `[a]`. The `mapActions` function finally gives bac an `IO`-action that returns a list.

Svar.

```
mapActions :: (a -> IO b) -> [a] -> IO [b]
mapActions f [] = return []
mapActions f (x : xs) = do
  y <- f x
  ys <- mapActions f xs
  return (y : ys)
```