Lista de Exercícios em Haskell - Lista 2

Alunos:

- Gabriel Rocha Souza Silva
- Lucas Araujo Bourguignon

1. Filtrar elementos de uma lista com base em uma função

Dada uma função para classificar uma lista de `a`, se for `Just a`, `a` entra na lista final; se for `Nothing`, `a` não entra.

```haskell
testFunc :: Int -> Maybe Int
testFunc x
    | x >= 2    = Just x
    | otherwise = Nothing

mapMaybe :: (a -> Maybe b) -> [a] -> [b]
mapMaybe _ [] = []
mapMaybe g (x:xs) =
    case g x of
        Just y  -> y : mapMaybe g xs
        Nothing -> mapMaybe g xs
```

Exemplo de uso:

```haskell
main :: IO ()
main = do
        let m = mapMaybe testFunc [2,3,1,0,-1,-232]
        print m
```

2. Classificar elementos de uma lista em dois grupos

Recebendo uma lista de valores, se for `Right x`, `x` irá para a lista da direita; se for `Left y`, `y` irá para a lista da esquerda.

```haskell
classifica :: [Either a b] -> ([a], [b])
classifica [] = ([], [])
classifica (Left x : xs) = (x:ls, rs)
        where (ls, rs) = classifica xs
classifica (Right y : ys) = (ls, y:rs)
        where (ls, rs) = classifica ys
```

Exemplo de uso:

```haskell
main :: IO ()
main = do
        let c = classifica [Left 6, Left 5, Right True, Left 3]
        print c
```

3. Descobrir como duas listas diferem entre si

- Se tiverem comprimentos diferentes, retorna `Just "<comprimento da lista 1> /= <comprimento da lista 2>"`.
- Se tiverem o mesmo comprimento, encontra o primeiro índice onde os elementos são diferentes e retorna `Just "<índice da lista 1> /= <índice da lista 2>"`.
- Se as listas forem iguais, retorna `Nothing`.

```haskell
findDifference :: (Eq a, Show a) => [a] -> [a] -> Maybe String
findDifference [] [] = Nothing
findDifference xs ys
        | length xs /= length ys = Just (show (length xs) ++ " /= " ++ show
(length ys))
        | otherwise = findDifference' xs ys 0
        where
                findDifference' [] [] _ = Nothing
                findDifference' (x:xs) (y:ys) i
                        | x /= y = Just (show i ++ " /= " ++ show i)
                        | otherwise = findDifference' xs ys (i+1)
```

Exemplos de uso:

```haskell
main :: IO ()
main = do
        print $ findDifference [1,2,3] [1,2,3] -- Nothing
        print $ findDifference [1,2,3] [1,2,4] -- Just "2 /= 2"
        print $ findDifference [1,2,3] [1,2,3,4] -- Just "3 /= 4"
        print $ findDifference [1,2,3] [1,2] -- Just "3 /= 2"
```

4. Implementar uma instância de `Eq` para um vetor 3D

Este é um tipo para um vetor 3D. A instância de `Eq` verifica se todos os elementos dos vetores são iguais.

```haskell
data Vetor = Vetor Integer Integer Integer
        deriving (Show)

instance Eq Vetor where
        (Vetor x1 y1 z1) == (Vetor x2 y2 z2) = x1 == x2 && y1 == y2 && z1 == z2
```

Exemplos de uso:

```
main :: IO ()
main = do
        let v1 = Vetor 1 2 3
        let v2 = Vetor 1 2 3
        putStrLn $ "Teste 1 - Vetores iguais: " ++ show (v1 == v2) -- True

        let v3 = Vetor 1 2 3
        let v4 = Vetor 4 5 6
        putStrLn $ "Teste 2 - Vetores diferentes: " ++ show (v3 == v4) -- False

        let v5 = Vetor 1 2 3
        let v6 = Vetor 1 2 4
        putStrLn $ "Teste 3 - Vetores com um elemento diferente: " ++ show (v5 ==
v6) -- False
```

5. Implementar uma instância de `Num` para Vetor

As operações são feitas elemento a elemento.

Exemplos:

- `Vetor 1 2 3 + Vetor 0 1 1 == Vetor 1 3 4`
- `Vetor 1 2 3 * Vetor 0 1 2 == Vetor 0 2 6`
- `abs (Vetor (-1) 2 (-3)) == Vetor 1 2 3`
- `signum (Vetor (-1) 2 (-3)) == Vetor (-1) 1 (-1)`

```
data Vetor = Vetor Integer Integer Integer
        deriving (Show)

instance Eq Vetor where
        (Vetor x1 y1 z1) == (Vetor x2 y2 z2) = x1 == x2 && y1 == y2 && z1 == z2

instance Num Vetor where
        (Vetor x1 y1 z1) + (Vetor x2 y2 z2) = Vetor (x1 + x2) (y1 + y2) (z1 + z2)
        (Vetor x1 y1 z1) * (Vetor x2 y2 z2) = Vetor (x1 * x2) (y1 * y2) (z1 * z2)
        abs (Vetor x y z) = Vetor (abs x) (abs y) (abs z)
        signum (Vetor x y z) = Vetor (signum x) (signum y) (signum z)
        fromInteger n = Vetor n n n
        negate (Vetor x y z) = Vetor (negate x) (negate y) (negate z)
```

Exemplos de uso:

```
main :: IO ()
main = do
        let v1 = Vetor 1 2 3
```

```
        let v2 = Vetor 0 1 1
        putStrLn $ "Teste 1 - Soma de vetores: " ++ show (v1 + v2) -- Vetor 1 3 4

        let v3 = Vetor 1 2 3
        let v4 = Vetor 0 1 2
        putStrLn $ "Teste 2 - Mult de vetores: " ++ show (v3 * v4) -- Vetor 0 2 6

        let v5 = Vetor (-1) 2 (-3)
        putStrLn $ "Teste 3 - Valor absoluto de um vetor: " ++ show (abs v5) --
Vetor 1 2 3

        let v6 = Vetor (-1) 2 (-3)
        putStrLn $ "Teste 4 - Signum de um vetor: " ++ show (signum v6) -- Vetor
(-1) 1 (-1)
```