UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FEELT - FACULDADE DE ENGENHARIA ELÉTRICA
ENGENHARIA DE COMPUTAÇÃO

LUCAS ALBINO MARTINS

Matrícula: 12011ECP022

TRABALHO 02: 2ª. Lista de Exercícios: Processamento de listas em Haskell.

Disciplina: Programação Funcional

Uberlândia
2020

Faculdade de Engenharia Elétrica - Engenharia de Computação
Programação Funcional (EC) – 1° Período
2a. Lista de Exercícios: Processamento de listas em Haskell

1. **Mostre as listas geradas pelas seguintes expressões Haskell:**

> [n*n| n<-[1..10],even n]

```
Prelude> [n*n| n<-[1..10],even n]
[4,16,36,64,100]
```

> [7| n<-[1..5]]

```
Prelude> [7| n<-[1..5]]
[7,7,7,7,7]
```

> [(x,y)| x<-[1..5], y<-[4..7]]

```
Prelude> [(x,y)| x<-[1..5], y<-[4..7]]
[(1,4),(1,5),(1,6),(1,7),(2,4),(2,5),(2,6),(2,7),(3,4),(3,5),(3,6),(3,7),(4,4),(4,5),(4,6),(4,7),(5,
4),(5,5),(5,6),(5,7)]
```

> [(m,n)| m<-[1..3], n<-[1..m]]

```
Prelude> [(m,n)| m<-[1..3], n<-[1..m]]
[(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]
```

> [j| i<-[1,-1,2,-2], i>0, j<-[1..i]]

```
Prelude> [j| i<-[1,-1,2,-2], i>0, j<-[1..i]]
[1,1,2]
```

> [a+b | (a,b) <- [(2,2),(3,3),(4,4),(5,5)]]

```
Prelude> [a+b | (a,b) <- [(2,2),(3,3),(4,4),(5,5)]]
[4,6,8,10]
```

2. Defina uma função que dada uma lista de inteiros, retorna o número de elementos de valor superior a um número *n* qualquer.

> retornaSup 4 [3,2,5,6]
2

Código:

```
retornaSup n [ ] = 0
retornaSup n (h :t) = if ( n < h) then 1 + retornaSup n t
 else retornaSup n t
retornaSuplist n lista = retornaSupAux n lista 0
 where
retornaSupAux n [ ] res = res
retornaSupAux n (h : t) res = if ( n < h) then retornaSupAux n t (res+1)
 else retornaSupAux n t res
```

Execução:

*Main> retornaSup 4 [1,2,6,2,8,14,1]

3

3. Defina uma função que dada uma lista de inteiros, retorna outra lista que contém apenas de elementos de valor superior a um número *n* qualquer.

> retornaListaSup 4 [3,2,5,6]
[5,6]

Código

```
retornaListaSup::Int->[Int]->[Int]
retornaListaSup x [] = []
retornaListaSup x (a:t) = if a > x then a: retornaListaSup x t
                else retornaListaSup x t
```

Execução:

*Main> retornaListaSup 3 [1,2,3,4,5,6,7,8]
[4,5,6,7,8]
*Main> retornaListaSup 3 [1,2,3,4,5,6,7,8,3,1,2,7,8,9]
[4,5,6,7,8,7,8,9]

4. Escreva uma função que recebe duas listas de inteiros e produz uma lista de listas. Cada uma corresponde à multiplicação de um elemento da primeira lista por todos os elementos da segunda.

> mult_listas [1,2] [3,2,5]
[[3,2,5],[6,4,10]]

Código

```
multiplicaovalor :: Int -> [Int] -> [Int]
multiplicaovalor n list = [x*n | x <- list]

multiplicalistas :: [Int] -> [Int] -> [[Int]]
multiplicalistas list1 list2 = [multiplicaovalor x list2 | x <- list1]
```

Execução:

*Main> multiplicalistas [2,3] [4,5,6]
[[8,10,12],[12,15,18]]

5. Escreva uma função para verificar se os elementos de uma lista são distintos.

Código:

```
distintos :: (Eq a) => [a] -> Bool
distintos [] = True
distintos (x:xs) = if repetidos x xs then False else distintos xs

repetidos :: (Eq a) => a -> [a] -> Bool
repetidos a [] = False
repetidos a (x:xs) = if a == x then True else repetidos a xs
```

Execução:

*Main> distintos [1,2,3,4,5,6]
True
*Main> distintos [1,2,3,4,5,6,11,14,1]
False

6. Seja a função união abaixo, definida através da construção de listas por compreensão. Teste esta função e implemente a interseção utilizando a mesma estratégia.

uniao :: Eq t => [t] -> [t] -> [t]
uniao as bs = as ++ [b | b <- bs, not (pertence b as)]

Código:

```
intersecao :: (Eq a) => [a] -> [a] -> [a]
intersecao [] _ = []
intersecao _ [] = []
intersecao (x:xs) list
                        | x `elem` list = x : intersecao xs list
                        | otherwise = intersecao xs list
```

Execução:

*Main> intersecao [9,13,45,1,12,43,4,85] [14,13,13,1,3,6,26,39,45]
[13,45,1]
*Main> intersecao [1,2,3,4,5] [6,7,8,9,10]
[]

7. Use a construção de listas por compreensão para definir uma função que recebe uma lista e retorna uma lista contendo apenas os elementos negativos da lista original.

Código:

```
negativos :: [Int] -> [Int]
negativos [] = []
negativos (h:t) = if (h <= 0) then h : negativos t else negativos t
```

Execução:

*Main> negativos [-1,2,3,-4,6,5,-112,-16,-231,35,-32]
[-1,-4,-112,-16,-231,-32]

8. Seja a função abaixo que recebe uma lista de pontos no plano cartesiano e calcula a distância de cada ponto à origem:

distancias :: [(Float,Float)] -> [Float]
distancias [] = []
distancias ((x,y):xys) = (sqrt (x^2 + y^2)) : (distancias xys)

Escreva uma outra versão da função distâncias, utilizando a construção de listas por compreensão.

Código:

```
distancia :: Floating a => [(a, a)] -> a
distancia = sum . (dist <*> tail)
 where dist = zipWith (\p s -
> sqrt ((fst p - fst s)^2 + (snd p - snd s)^2))
```

Execução:

*Main> distancia [(1,2),(3,4)]
2.8284271247461903

9. Defina a função tabuada :: Int -> [(Int, Int, Int)] que dado um inteiro n produz uma lista da tabuada dos n.

> tabuada 5
[(5,1,5), (5,2,10),(5,3,15), … , (5,10,50)]

Código:

```
tabuada x = map (\i -> (x, i, x*i)) [1..10]

main = mapM_ putStrLn . map showTab . take 10 $ tabuada 5
 where
   showTab (a, b, c) = show a ++ "*" ++ show b ++ "=" ++ show c
```

Execução:

*Main> tabuada 8

[(8,1,8),(8,2,16),(8,3,24),(8,4,32),(8,5,40),(8,6,48),(8,7,56),(8,8,64),(8,9,72),(8,10,80)]

*Main> tabuada 9

[(9,1,9),(9,2,18),(9,3,27),(9,4,36),(9,5,45),(9,6,54),(9,7,63),(9,8,72),(9,9,81),(9,10,90)]

*Main>

Tela com editor Haskell