Faculdade de Computação
Programação Funcional (BCC/BSI) - 1° Período
3a. Lista de Exercícios: Listas, Classes e Cálculo Lambda

1. Mostre o resultado obtido pela execução das expressões Haskell:

```
> (\x -> x + 3) 5

> (\x -> \y -> x * y + 5) 3 4

> map (\x -> x ^3) [2,4,6]

deriv :: Fractional a => (a -> a) -> a -> a -> a
deriv f dx = \x -> (f(x + dx) - f(x)) / dx

> deriv (\x -> x*x*x) 0.0001 1

> (\(x,y) -> x * y^2) (3,4)

> (\(x,y,_) -> x * y^2) (3,4,2)

> map (\(x,y,z) -> x + y + z) [(3,4,2),(1,1,2),(0,0,4)]

> filter (\(x,y) -> x `mod` y == 0) [(4,2),(3,5),(6,3)]

> (\xs -> zip xs [1,2,3]) [4,5,6]

> map (\xs -> zip xs [1..]) [[4,6],[5,7]]

> foldr1 (+) [1,2,3]

>  foldr1 (\x -> \y -> x + y + 7) [1,2,3,4,5]
```

2. Seja a criação de um novo tipo, denominado *NomeCompleto*. Defina uma função que compara dois valores do tipo *NomeCompleto* e retorna verdadeiro se ambos são iguais (caso contrário, falso). Faça os testes abaixo e explique os resultados obtidos. Caso necessário, modifique ou inclua novas instruções ao programa.

```
data NomeP = Nome String  deriving (Show)
data SobreNomeP = SobreNome String  deriving (Show)

type NomeCompleto = (NomeP,SobreNomeP)
```

```
> compara (Nome "Ana", SobreNome "Lima") (Nome "Caio", SobreNome "Silva")

> (Nome "Ana", SobreNome "Lima") ==(Nome "Caio", SobreNome "Silva")

> (Nome "Cris", SobreNome "Dias") > (Nome "Cris", SobreNome "Dias")
```

3. Dada a definição de tipos abaixo, teste a função *avalia* para duas expressões quaisquer e forneça o resultado 'passo a passo'.

```
data Exp a = Val a                  -- um numero
           | Neg (Exp a)
           | Add (Exp a) (Exp a)    -- soma de duas expressoes
           | Sub (Exp a) (Exp a)    -- subtracao
           | Mul (Exp a) (Exp a)    -- multiplicacao
           | Div (Exp a) (Exp a)    -- divisao

avalia :: Fractional a => Exp a -> a
avalia (Val x)          = x
avalia (Neg exp)        = - (avalia exp)
avalia (Add exp1 exp2)  = (avalia exp1) + (avalia exp2)
avalia (Sub exp1 exp2)  = (avalia exp1) - (avalia exp2)
avalia (Mul exp1 exp2)  = (avalia exp1) * (avalia exp2)
avalia (Div exp1 exp2)  = (avalia exp1) / (avalia exp2)
```

4) Seja o código abaixo em Haskell para a manipulação da latitude e longitude de uma posição geográfica:

- A latitude é a distância ao Equador medida ao longo do meridiano de Greenwich e varia de 0 a 90° para Norte ou Sul.
- A longitude é a distância ao meridiano de Greenwich medida ao longo do Equador e varia de 0 a 180° para Leste ou Oeste.

```
data LL = Latitude Int Int Int
        | Longitude Int Int Int    deriving (Eq)

instance Show LL where
     show (Latitude a b c) =
     "Lat " ++ show a ++"°" ++ show b ++"'"++ show c ++"''"

type PosicaoLocal = (String, LL, LL)
type Cidades = [PosicaoLocal]
```

A) Explique as definições abaixo e informe o resultado das chamadas:

```
c1,c2::PosicaoLocal
c1 = ("Brasilia", Latitude (-15) 46 47, Longitude 47 55 47)
c2 = ("Uberlandia", Latitude (-18) 55 07, Longitude 48 16 38)

eLat::PosicaoLocal->(String,LL)
eLat (p,(Latitude a b c),
        (Longitude x y z)) = (p,(Latitude a b c))

> eLat c1

> eLat ("Torres", Latitude (-29) 20 07, Longitude 49 43 37)
```

B) Faça uma função denominada *NorteDe* que, dadas duas cidades (tipo PosicaoLocal) devolve verdadeiro se a primeira está ao Norte da segunda. A função deve comparar as latitudes das duas cidades.

```
NorteDe::PosicaoLocal->PosicaoLocal->Bool
```

C) Dada uma lista de cidades, faça funções para:

```
lcidades::Cidades
lcidades =
   [("Rio Branco", Latitude 09 58 29, Longitude 67 48 36),
    ("Brasilia", Latitude (-15) 46 47, Longitude 47 55 47),
    ("Torres", Latitude (-29) 20 07, Longitude 49 43 37),
    ("Joao Pessoa", Latitude (-07) 06 54, Longitude 34 51 47),
    ("Uberlandia", Latitude (-18) 55 07, Longitude 48 16 38)]
```

C.1) Retornar quantas estão abaixo da linha do Equador
C.2) Retornar os nomes das cidades com longitude entre 40 e 50 graus

5) Seja o código abaixo. Explique o tipo Talvez e mostre o resultado para as chamadas:

```
data Talvez a = Valor a | Nada deriving (Show)

divisaoSegura :: Float -> Float -> Talvez Float
divisaoSegura x y = if y == 0 then Nada else Valor (x/y)
```

```
> divisaoSegura 5 0

> divisaoSegura 5 4
```

6) A função *addPares* pode ser definida através de lista por compreensão:

```
addPares :: [(Int,Int)] -> [Int]
addPares lista :: [ m+n | (m,n) <- lista ]
```

Neste caso, todos os pares encontrados na lista serão avaliados. Os componentes de cada par devem ser somados e fornecidos como resposta numa nova lista:

```
> addPares [(2,3),(2,1),(5,4)]
     m =   2  2  3
     n =   3  1  4
     m+n= 5  3  7          ↳  [5,3,7]
```

A) Mude a função addPares para que os componentes de cada par sejam somados apenas se o primeiro for menor que o segundo:

```
> addParesT [(2,3),(2,1),(5,4)]
     m =   2  2  3
     n =   3  1  4
     m+n=  5     7          ↳  [5,7]
```

B) Escreva uma nova versão da função `addPares` usando uma expressão lambda.

C) Refaça a função `addParesT` usando uma expressão lambda e funções genéricas (como map e filter).

7) A função mp dada abaixo recebe uma função e duas listas. O que a função retorna?

```
mp f [] ys = []
mp f xs [] = []
mp f (x:xs) (y:ys) = f x y : mp f xs ys
```

8) Defina uma função para calcular a soma dos quadrados dos números naturais de 1 a n utilizando as funções `map` e `foldr1`.

$1^2 + 2^2 + 3^2 + \ldots + n^2 = ?$

```
Main> somaQuad 15
1240
```

9) Ainda utilizando funções genéricas, defina uma função que retorna a soma dos quadrados dos números positivos numa lista de inteiros.

$[2,0,-2,6,-7,4] = 2^2+0^2+6^2+4^2$

```
Main>  somaQuadPos [2,0,-2,6,-7,4]
56
```

10) Como a função mistério se comporta, considerando que `fun x = [x]` ? Dê um exemplo e mostre os passos da execução.

```
misterio xs = foldr (++) [] (map fun xs)
```