

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

LUCAS ALBINO MARTINS

Matrícula: 12011ECP022

**TRABALHO 03: 3ª. Lista de Exercícios: Listas, tipos, classes e
lambda calculus em Haskell.**

Disciplina: Programação Funcional

Uberlândia
2020

Faculdade de Engenharia Elétrica - Engenharia de Computação
Programação Funcional (EC) – 1º Período
3a. Lista de Exercícios: Listas, tipos, classes e lambda calculus em Haskell

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

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

8

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

17

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

[8,64,216]

```
deriv :: Fractional a => (a -> a) -> a -> a -> a
```

```
deriv f dx = \x -> (f(x + dx) - f(x)) / dx
```

```
Prelude> :l deriv.hs
```

```
[1 of 1] Compiling Main      ( deriv.hs, interpreted )
```

Ok, one module loaded.

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

3.0003000099987354

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

48

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

48

```
*Main> map (\(x,y,z) -> x + y + z) [(3,4,2),(1,1,2),(0,0,4)]  
[9,4,4]
```

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

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

```
*Main> map (\xs -> zip xs [1..]) [[4,6],[5,7]]  
[[ (4,1),(6,2) ], [ (5,1),(7,2) ]]
```

```
*Main> foldr1 (+) [1,2,3]  
6
```

```
*Main> foldr1 (\x -> \y -> x + y + 7) [1,2,3,4,5]  
43
```

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")
```

Inicialmente o uso `data` para determinar um novo tipo de dado. A cláusula `deriving` permite declarar as classes das quais o novo tipo será instância, automaticamente. Logo, segundo a declaração dada, o tipo `NomeP` é uma instância da classe `Show`, e a função `show` é sobrecarregada para o tipo `NomeP`, o mesmo procedimento ocorre com o `SobrenomeP`.

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)
```

Resposta:

Declarado a função `avalia`, a função `data` para um novo tipo de dado, no caso `Val`.

```
*Main> avalia (Mul (Val 10) (Val 20))
200.0
*Main> avalia (Add (Val 10) (Val 20))
20.0
```

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)
```

Resposta:

```
c1,c2::PosicaoLocal
```

Declarado que c1 e c2 são do tipo PosicaoLocal.

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

Então são atribuídos os valores do tipo PosicaoLocal para c1 e c2 baseado no modelo de entrada aceito na função LL.

eLat::PosicaoLocal->(String,LL)

É declarado que eLat e do tipo PosicaoLocal, no qual e do tipo String e que recebe inteiros pois os dados de LL foi declarado para receber inteiros.

eLat (p,(Latitude a b c), (Longitude x y z)) = (p,(Latitude a b c))

A chamada para execução pega os valores de Latitude e Longitude e imprime a Latitude como explicito no corpo da função LL.

Resultado da execução:

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

```
("Torres",Lat -29°20'7")
```

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

```
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]

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,PosicaoLocal)->(String,LL)
eLat ((p,(Latitude a b c),(Longitude x y z)),(q,(Latitude d e f),(Longitude r s t))) = (p,(Latitude a b c))

norteDe::(PosicaoLocal,PosicaoLocal)->Bool
norteDe ((p,(Latitude a b c),(Longitude x y z)),(q,(Latitude d e f),(Longitude r s t))) = a > d
```

Saída:

```
*Main> norteDe (("Brasilia", Latitude (-15) 46 47, Longitude 47 55 47),("Torres",  
Latitude (-29) 20 07, Longitude 49 43 37))
```

True

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

```
lidades::Cidades lidades = [("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
```

Talvez é um construtor de tipo, no caso tipo polimórfico, se alguma função precisar de um Talvez float como parâmetro, podemos passar o primeiro “Valor a” (Quando e possível dividir pelo valor) ou o segundo “Nada” (Quando o valor está sendo dividido por 0).

```
*Main> divisaoSegura 5 0
```

Nada

```
*Main> divisaoSegura 5 4
```

Valor 1.25

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]
```

Resolução:

Função alterada.

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

Execução:

```
*Main> addParesT [(2,3),(2,1),(3,4)]
```

```
[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
```

A função mp definida no módulo Prelude padrão tem a finalidade de criar pares a partir dos elementos de duas listas iniciais.

```
*Main> mp ['a','b','c'] [1,2,3]
```

```
[('a',1),('b',2),('c',3)]
```

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

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

```
Main> somaQuad 15 1240
```

```
somaquadrados :: Int -> Int
somaquadrados 0 = 0
somaquadrados 1 = 1
somaquadrados n = foldr1 (+) (map (\x -> x^2) [1..n])
```

```
*Main> somaquadrados 5
```

```
55
```

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

```
quadrados :: [Int] -> Int
quadrados [] = 0
quadrados n = foldr1 (+) (map (\n -> n^2) n)

elemento_positivo :: [Int] -> [Int]
elemento_positivo [] = []
elemento_positivo (n:ns) = if (n > 0) then n : elemento_positivo ns
                           else elemento_positivo ns

somaquadrados :: [Int] -> Int
somaquadrados [] = 0
somaquadrados n = quadrados (elemento_positivo n)
```

*Main> somaquadrados [2,0,-2,6,-7,4]

56

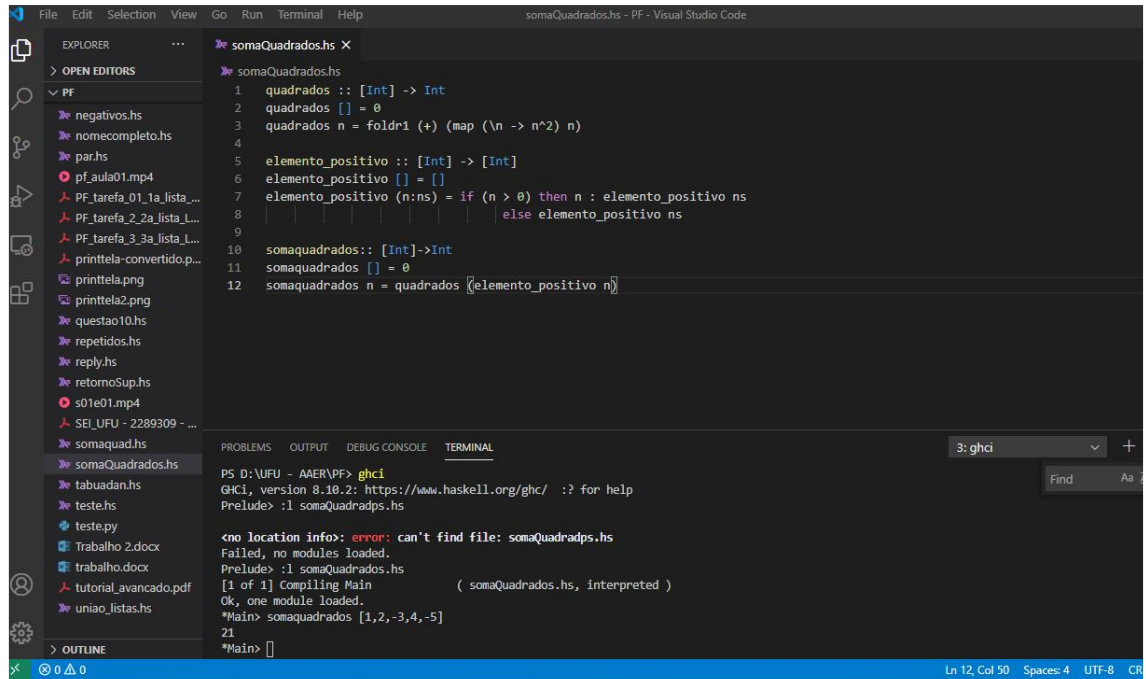
10. Como a função `misterio` se comporta, considerando que `fun x = [x]` ? Dê um exemplo e mostre os passos da execução.

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

A função `misterio` atua como um concatenador de uma lista mapeada dos valores começando da direita para esquerda.

```
concatenacao :: [String] -> String
concatenacao ls = foldr (++) "" ls
```

Anexo:



The image shows a screenshot of the Visual Studio Code editor. The Explorer sidebar on the left lists several files, including `somaQuadrados.hs`. The main editor window displays the content of `somaQuadrados.hs`, which defines functions for calculating squares and their sum.

```
1 quadrados :: [Int] -> Int
2 quadrados [] = 0
3 quadrados n = foldr1 (+) (map (\n -> n^2) n)
4
5 elemento_positivo :: [Int] -> [Int]
6 elemento_positivo [] = []
7 elemento_positivo (n:ns) = if (n > 0) then n : elemento_positivo ns
8 | | | | | else elemento_positivo ns
9
10 somaquadrados :: [Int] -> Int
11 somaquadrados [] = 0
12 somaquadrados n = quadrados (elemento_positivo n)
```

The bottom panel shows the TERMINAL output, indicating that the file `somaQuadrados.hs` was successfully compiled and interpreted, resulting in the output `*Main> somaquadrados [1,2,-3,4,-5]`.

```
PS D:\UFU - AAER\PF> ghci
Ghci, version 8.10.2: https://www.haskell.org/ghc/ :? for help
Prelude> :l somaQuadrados.hs
<no location info>: error: can't find file: somaQuadrados.hs
Failed, no modules loaded.
Prelude> :l somaQuadrados.hs
[1 of 1] Compiling Main             ( somaQuadrados.hs, interpreted )
Ok, one module loaded.
*Main> somaquadrados [1,2,-3,4,-5]
21
*Main>
```