# Paradigmas de Programação

Fabrício Olivetti de França

19 de Junho de 2018

# Funções no Haskell

# Atividades

Para as atividades de hoje, crie um arquivo chamado *atividade03.hs* para escrever os códigos respostas. Teste as respostas carregando o arquivo no *ghci*.

# Funções

O Haskell é baseado no $\lambda$-cálculo. Como vimos na aula passada, toda função recebe uma entrada e uma saída:

```
somaUm :: Integer -> Integer
somaUm x = x + 1
```

`somaUm`

Nome da função deve começar com caixa baixa. O estilo padrão é o *camelCase*.

```
somaUm ::
```

Definição dos tipos de entrada e saída.

```
somaUm :: Integer
```

Recebe um valor inteiro.

```
somaUm :: Integer -> Integer
```

Retorna um valor inteiro.

```
somaUm :: Integer -> Integer
somaUm x
```

A função, dado um valor x...

```
somaUm :: Integer -> Integer
somaUm x =
```

...é definida como...

```haskell
somaUm :: Integer -> Integer
somaUm x = x + 1
```

...a expressão x + 1.

# Mais de uma variável

Funções de mais de uma variável, na verdade são composições de funções:

```
soma x y = x + y
```

Na verdade é:

```
-- resultado de soma x aplicado em y
soma :: Integer -> (Integer -> Integer)
soma x y = (soma x) y
soma = \x -> (\y -> x + y)
```

Isso é denominado **currying**

# Exercício

Qual a única função possível com a seguinte assinatura?

```
f :: a -> a
```

# Exercício

Qual a única função possível com a seguinte assinatura?

```
f :: a -> a
f x = x
```

# Exemplos de funções

Funções devem ser escritas em forma de expressões combinando outras funções, de tal forma a manter simplicidade:

```
impar :: Integral a => a -> Bool
impar n = n `mod` 2 == 1
```

# Exemplos de funções

Funções devem ser escritas em forma de expressões combinando outras funções, de tal forma a manter simplicidade:

```
quadrado :: Num a => a -> a
quadrado n = n*n
```

# Exemplos de funções

Funções devem ser escritas em forma de expressões combinando outras funções, de tal forma a manter simplicidade:

```
quadradoMais6Mod9 :: Integral a => a -> a
quadradoMais6Mod9 n = (n*n + 6) `mod` 9
```

# Exercício 01

Escreva uma função que retorne a raíz de uma equação do segundo grau:

```
raiz2Grau :: Floating a => a -> a -> a -> (a, a)
raiz2Grau a b c = ( ???, ??? )
```

Teste com `raiz2Grau 4 3 (-5)` e `raiz2Grau 4 3 5`.

# Cláusula where

Para organizar nosso código, podemos utilizar a cláusula **where** para definir nomes intermediários:

```
f x = y + z
  where
    y = e1
    z = e2
```

# Exemplo

```haskell
euclidiana :: Floating a => a -> a -> a
euclidiana x y = sqrt diffSq
  where
    diffSq = diff^2
    diff   = x - y
```

# Exercício 02

Reescreva a função `raiz2Grau` utilizando `where`.

# Condicionais

A função `if-then-else` nos permite utilizar desvios condicionais em nossas funções:

```
abs :: Num a => a -> a
abs n = if (n >= 0) then n else (-n)
```

ou

```
abs :: Num a => a -> a
abs n = if (n >= 0)
        then n
        else (-n)
```

# Condicionais

Também podemos encadear condicionais:

```
signum :: (Ord a, Num a) => a -> a
signum n = if (n == 0)
           then 0
           else if (n > 0)
                then 1
                else (-1)
```

# Exercício 03

Utilizando condicionais, reescreva a função `raiz2Grau` para retornar (0,0) no caso de delta negativo.

Note que a assinatura da função agora deve ser:

```
raiz2Grau :: (Ord a, Floating a) => a -> a -> a -> (a, a)
```

# Expressões *guardadas* (Guard Expressions)

Uma alternativa ao uso de `if-then-else` é o uso de *guards* (`|`) que deve ser lido como *tal que*:

```
signum :: (Ord a, Num a) => a -> a
signum n | n == 0     =  0  -- signum n tal que n==0
                            -- é definido como 0
         | n > 0      =  1
         | otherwise = -1
```

`otherwise` é o caso contrário e é definido como `otherwise = True`.

# Expressões *guardadas* (Guard Expressions)

Note que as expressões guardadas são avaliadas de cima para baixo, o primeiro verdadeiro será executado e o restante ignorado.

```
classificaIMC :: Double -> String
classificaIMC imc
    | imc <= 18.5 = "abaixo do peso"
    -- não preciso fazer && imc > 18.5
    | imc <= 25.0 = "no peso correto"
    | imc <= 30.0 = "acima do peso"
    | otherwise   = "muito acima do peso"
```

# Exercício 04

Utilizando guards, reescreva a função `raiz2Grau` para retornar um erro com raízes negativas.

Para isso utilize a função `error`:

```
error "Raízes negativas."
```

# Nota sobre erro

O uso de `error` interrompe a execução do programa. Nem sempre é a melhor forma de tratar erro, aprenderemos alternativas ao longo do curso.

# Pattern Matching

Considere a seguinte função:

```
not :: Bool -> Bool
not x = if (x == TRUE) then FALSE else TRUE
```

# Pattern Matching

Podemos reescreve-la utilizando guardas:

```
not :: Bool -> Bool
not x | x == TRUE  = FALSE
      | x == FALSE = TRUE
```

# Pattern Matching

Quando temos comparações de igualdade nos guardas, podemos definir as expressões substituindo diretamente os argumentos:

```
not :: Bool -> Bool
not TRUE  = FALSE
not FALSE = TRUE
```

# Pattern Matching

Não precisamos enumerar todos os casos, podemos definir apenas casos especiais:

```
soma :: (Eq a, Num a) => a -> a -> a
soma x 0 = x
soma 0 y = y
soma x y = x + y
```

Assim como os guards, os padrões são avaliados do primeiro definido até o último.

# Pattern Matching

Implemente a multiplicação utilizando Pattern Matching:

```
mul :: Num a => a-> a -> a
mul x y = x*y
```

# Pattern Matching

Implemente a multiplicação utilizando Pattern Matching:

```
mul :: (Eq a, Num a) => a-> a -> a
mul 0 y = 0
mul x 0 = 0
mul x 1 = x
mul 1 y = y
mul x y = x*y
```

# Pattern Matching

Quando a saída não depende da entrada, podemos substituir a entrada por _ (não importa):

```
mul :: (Eq a, Num a) => a-> a -> a
mul 0 _ = 0
mul _ 0 = 0
mul x 1 = x
mul 1 y = y
mul x y = x*y
```

# Pattern Matching

Como o Haskell é preguiçoso, ao identifica um padrão contendo 0 ele não avaliará o outro argumento.

```
mul :: (Eq a, Num a) => a-> a -> a
mul 0 _ = 0
mul _ 0 = 0
mul x 1 = x
mul 1 y = y
mul x y = x*y
```

# Expressões $\lambda$

As expressões lambdas feitas anteriormente são válidas no Haskell.

Vamos criar uma função que retorna uma função que soma e multiplica `x` a um certo número:

```
somaMultX :: a -> (a -> a)
somaMultX x = \y -> x + x*y

somaMult2 = somaMultX 2
```

# Operadores

Para definir um operador em Haskell, podemos criar na forma infixa ou na forma de função:

```
(:+) :: Num a => a -> a -> a
x :+ y = abs x + y
```

ou

```
(:+) :: Num a => a -> a -> a
(:+) x y = abs x + y
```

# Operadores

Da mesma forma, uma função pode ser utilizada como operador se envolta de crases:

```
> mod 10 3
1
> 10 `mod` 3
1
```

# Operadores

Sendo # um operador, temos que `(#)`, `(x #)`, `(# y)` são chamados de seções, e definem:

```
(#)   = \x -> (\y -> x # y)
(x #) = \y -> x # y
(# y) = \x -> x # y
```

# Operadores

Essas formas são também conhecidas como **point-free notation**:

```
> (/) 4 2
2
> (/2) 4
2
> (4/) 2
2
```

# Exercício

Considere o operador `(&&)`, simplique a definição para apenas dois padrões:

```
(&&) :: Bool -> Bool -> Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

# Exercício 04

Os seus malvados favoritos professores de Processamento da Informação resolveram em uma reunião normatizar a quantidade de avaliações, a conversão nota-conceito e a definição do conceito final dados os conceitos de Teoria e Prática:

- As avaliações de consistirão de duas provas na Teoria e duas na Prática
- A nota final de cada turma será dada pela média ponderada das provas, o peso é determinado por cada professor

# Exercício 04

- A conversão conceito final segue a seguinte tabela:
- nota < 5 = F
- nota < 6 = D
- nota < 7 = C
- nota < 8 = B
- nota >= 8 = A

# Exercício 04

- O conceito final é dado pela seguinte tabela:

| Teoria | Pratica | Final |
|---|---|---|
| A | A | A |
| A | B | A |
| A | C | B |
| A | D | B |
| A | F | F |
| B | A | B |
| B | B | B |
| B | C | B |
| B | D | C |
| B | F | F |

# Exercício 04

- O conceito final é dado pela seguinte tabela:

| Teoria | Pratica | Final |
|---|---|---|
| C | A | B |
| C | B | C |
| C | C | C |
| C | D | C |
| C | F | F |
| D | A | C |
| D | B | C |
| D | C | D |
| D | D | D |
| D | F | F |

# Exercício 04

- O conceito final é dado pela seguinte tabela:

| Teoria | Pratica | Final |
|---|---|---|
| F | A | F |
| F | B | F |
| F | C | F |
| F | D | F |
| F | F | F |

# Exercício 04

Crie as seguintes funções para ajudar os professores a calcularem o conceito final de cada aluno:

```haskell
-- dados dois pesos w1, w2, retorna uma função que
-- calcula a média ponderada de p1, p2
mediaPonderada :: (Eq a, Floating a) =>
                  a -> a -> (a -> a -> a)

-- converte uma nota final em conceito
converteNota :: (Ord a, Floating a) => a -> Char

-- calcula conceito final
conceitoFinal :: Char -> Char -> Char
```

A função `mediaPonderada` deve retornar erro se `w1 + w2 /= 1.0`.

# Exercício 04

Acrescente o seguinte código ao final do arquivo:

```
turmaA1Pratica = mediaPonderada 0.4 0.6
turmaA1Teoria  = mediaPonderada 0.3 0.7

p1A1P = 3
p2A1P = 8
p1A1T = 7
p2A1T = 10

mediaP = turmaA1Pratica p1A1P p2A1P
mediaT = turmaA1Teoria p1A1T p2A1T

finalA1 = conceitoFinal
          (converteNota mediaP) (converteNota mediaT)
```

# Exercício 04

Acrescente o seguinte código ao final do arquivo:

```
turmaA2Pratica = mediaPonderada 0.4 0.6
turmaA2Teoria  = mediaPonderada 0.3 0.9

p1A2P = 3
p2A2P = 8
p1A2T = 7
p2A2T = 10

mediaA2P = turmaA2Pratica p1A2P p2A2P
mediaA2T = turmaA2Teoria p1A2T p2A2T

finalA2 = conceitoFinal
          (converteNota mediaA2P) (converteNota mediaA2T)
```

# Exercício 04

Teste os códigos com `print finalA1` e `print finalA2`.