Funções em Haskell

MCTA016-13 - Paradigmas de Programação

Emilio Francesquini
e.francesquini@ufabc.edu.br

2019.Q2

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de **Paradigmas de Programação na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Conteúdo baseado no texto preparado, e gentilmente cedido, pelo Professor Fabrício Olivetti de França da UFABC.

Funções em Haskell

Atividades

UFABC

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

UFABC

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

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

UFABC

```
1 somaUm
```

- Nomes de funções
  - Devem começar com letras minúsculas
  - Por convenção, nomes de funções seguem o estilo *camelCase*

UFABC

```
1 somaUm ::
```

Definição dos tipos de entrada e saída.

```haskell
somaUm :: Integer ...
```

Recebe um valor do tipo inteiro...

UFABC

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

Recebe um valor do tipo inteiro e devolve um valor inteiro.

UFABC

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

A função `somaUm`, dado um valor `x` de tipo inteiro...

UFABC

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

...é definida como...

UFABC

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

...a expressão `x + 1` do tipo inteiro

Mais de uma variável

UFABC

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

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

Na verdade é:

```haskell
-- 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 1

UFABC

Qual a única função possível com a seguinte assinatura?

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

Exemplos de funções

UFABC

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

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

Exemplos de funções

UFABC

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

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

Exemplos de funções

UFABC

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

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

Exercício 2

UFABC

Escreva uma função que retorne as raizes de uma equação do segundo grau:

```haskell
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`

UFABC

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

```
1  f x = y + z
2    where
3      y = e1
4      z = e2
```

- Atenção à indentação!

Exemplo

UFABC

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

Exercício 3

UFABC

Reescreva a função `raiz2Grau` utilizando `where`.

UFABC

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

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

ou

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

UFABC

Também podemos encadear condicionais:

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

Exercício 4

UFABC

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

Note que a assinatura da função agora deve ser:

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

- Por quê?

Expressões *guardadas* (Guard Expressions)

UFABC

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

```haskell
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
```

- Expressões imediatamente à direita de `|` devem ser do tipo `Bool`
- Caso mais de uma seja `True`, **apenas a primeira** delas será executada
- `otherwise` é o caso contrário e é definido como `otherwise = True`

Expressões *guardadas* (Guard Expressions)

UFABC

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

```haskell
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 5

UFABC

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

Para isso utilize a função `error`:

```
1 error "Raízes negativas."
```

Nota sobre erro

UFABC

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

UFABC

Considere a seguinte função:

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

UFABC

Podemos reescreve-la utilizando guardas:

```
1  not :: Bool -> Bool
2  not x | x == TRUE  = FALSE
3        | x == FALSE = TRUE
```

UFABC

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

```
1 not :: Bool -> Bool
2 not TRUE  = FALSE
3 not FALSE = TRUE
```

UFABC

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

```haskell
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 em sequência do primeiro até o último.

Exercício 6 - Pattern Matching

UFABC

Implemente a multiplicação utilizando Pattern Matching:

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

UFABC

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

```haskell
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
```

UFABC

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

```haskell
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
```

UFABC

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:

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

somaMult2 = somaMultX 2
```

Operadores

UFABC

- Operadores nada mais são do que funções cujos nomes são compostos por símbolos e não letras
- Para definir um operador em Haskell, podemos criar na forma infixa ou na forma de função

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

ou

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

UFABC

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

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

UFABC

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

```
1 (#)   = \x -> (\y -> x # y)
2 (x #) = \y -> x # y
3 (# y) = \x -> x # y
```

UFABC

Essas formas são também conhecidas como point-free notation:

```
1  > (/) 4 2
2  2
3  > (/2) 4
4  2
5  > (4/) 2
6  2
```

Exercício 7

UFABC

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

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

Exercício 8

UFABC

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 8

UFABC

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

Exercício 8

UFABC

- 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 8

UFABC

- 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 8

UFABC

- 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 8

UFABC

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 dar erro se `w1 + w2 /= 1.0`.

Exercício 8

UFABC

Acrescente o seguinte código ao final do arquivo:

```
1  turmaA1Pratica = mediaPonderada 0.4 0.6
2  turmaA1Teoria  = mediaPonderada 0.3 0.7
3
4  p1A1P = 3
5  p2A1P = 8
6  p1A1T = 7
7  p2A1T = 10
8
9  mediaP = turmaA1Pratica p1A1P p2A1P
10 mediaT = turmaA1Teoria p1A1T p2A1T
11
12 finalA1 = conceitoFinal
13           (converteNota mediaP) (converteNota mediaT)
```

Exercício 8

UFABC

Acrescente o seguinte código ao final do arquivo:

```
1  turmaA2Pratica = mediaPonderada 0.4 0.6
2  turmaA2Teoria  = mediaPonderada 0.3 0.9
3
4  p1A2P = 3
5  p2A2P = 8
6  p1A2T = 7
7  p2A2T = 10
8
9  mediaA2P = turmaA2Pratica p1A2P p2A2P
10 mediaA2T = turmaA2Teoria p1A2T p2A2T
11
12 finalA2 = conceitoFinal
13           (converteNota mediaA2P) (converteNota
             ↪  mediaA2T)
```

Exercício 8

UFABC

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