Introdução à Linguagem Haskell

Programação Funcional em Haskell

Prof. Rodrigo Ribeiro

Introdução à Linguagem Haskell — (I)

- Haskell é uma linguagem funcional pura.
  - Haskell possui avaliação sobre demanda.
  - Haskell é fortemente tipada.
- Diferenças entre linguagens funcionais e imperativas.
- Linguagens puras e efeitos colaterais.

Introdução à Linguagem Haskell — (II)

- Plataforma Haskell
  - Compilador / intepretador, ferramentas e bibliotecas
- Disponível para diversas plataformas.
- Além disso é útil um bom editor de texto...
  - Minha escolha pessoal: emacs.
  - Existem outros igualmente bons: sublime, kate, gedit...

Introdução à Linguagem Haskell — (III)

- O interpretador GHCi:
  - Ferramenta de linha de comando para "teste" de programas Haskell.
  - Para acessá-lo basta digitar ghci no prompt de comando (terminal).
- Algumas funções iniciais:
  - operadores aritméticos e lógicos

Introdução à Linguagem Haskell — (IV)

- Uma primeira função:

```
doubleMe :: Int -> Int
doubleMe x = x + x
```

- Componentes da função
  - Anotação de tipo (opcional)
  - Definição (corpo) da função

Introdução à Linguagem Haskell — (V)

```
[1 of 1] Compiling Main     ( Teste.hs, interpreted )
Ok, modules loaded: Main.
ghci> doubleMe 9
18
ghci> doubleMe 8
16
```

Introdução à Linguagem Haskell — (VI)

- A função (+) é polimórfica em Haskell...

```
doubleMe' x = x + x

[1 of 1] Compiling Main     ( Teste.hs, interpreted )
Ok, modules loaded: Main.
ghci> doubleMe' 9.2
18.4
ghci> doubleMe' 8.3
16.6
```

Introdução à Linguagem Haskell — (VII)

- Outra função simples em Haskell

```
doubleUs x y = x*2 + y*2
```

- Essa pode ser definida usando doubleMe'

```
doubleUs' x y = doubleMe' x + doubleMe' y
```

Introdução à Linguagem Haskell — (VIII)

- Comando condicional em Haskell
  - Note que o comando if em Haskell sempre possui o else.

```
doubleSmall x = (if x > 100 then x else x * 2) + 1
```

Introdução à Linguagem Haskell — (IX)

- Caracteres e Strings em Haskell
  - Representados pelos tipos String e Char
  - Strings são listas de caracteres
  - Listas serão vistas daqui a pouco...
- Representando constantes
  - String: “abc”
  - Char: ‘a’

Introdução à Linguagem Haskell — (X)

- Listas: Estruturas de dados muito comuns em Haskell.
- Listas são tipos de dados homogêneos.
  - Todos elementos de uma lista são do mesmo tipo.
- Definição recursiva de listas:
  - lista vazia: []
  - lista com pelo menos um elemento: (x:xs)

Introdução à Linguagem Haskell – (XI)

- Listas finitas podem ser representadas por simples enumeração de seus elementos:

```
my_list :: [Int]
my_list = [1,2,3,4]
```

- Note que essa definição é equivalente a:

```
my_list' :: [Int]
my_list' = 1 : 2 : 3 : 4 : []
```

Introdução à Linguagem Haskell — (XII)

- Sobre o ":"
  - Normalmente chamado de "cons"
  - Possui o tipo

```
(:) : a -> [a] -> [a]
```

- Listas são construídas utilizando ":" e "[]".
- O tipo de "[]" é:

```
[] :: [a]
```

Introdução à Linguagem Haskell — (XIII)

- Algumas funções sobre listas

```
-- concatenação de duas listas
(++) :: [a] -> [a] -> [a]
-- obter um elemento em uma posição
(!!) :: [a] -> Int -> a
-- obter o primeiro elemento
head :: [a] -> a
-- obter o último elemento
last :: [a] -> a
-- obtem toda a lista, exceto o último elemento
init :: [a] -> [a]
```

Introdução à Linguagem Haskell — (XIV)

- Algumas funções sobre listas

```
-- calcula o número de elementos da lista
length :: [a] -> Int
-- null testa se a lista é ou não vazia
null :: [a] -> Bool
-- reverse inverte uma lista
reverse :: [a] -> [a]
-- take extrai os n primeiros elementos de uma lista
take :: Int -> [a] -> [a]
-- drop remove os n primeiros elementos de uma lista
drop :: Int -> [a] -> [a]
```

Introdução à Linguagem Haskell — (XV)

- Mais sobre listas
  - Para tipos cujos valores formam um conjunto enumerável, podemos definir listas por intervalos.

```
my_enum_int :: [Int]
my_enum_int = [1..20]

my_enum_char :: [Char]
my_enum_char = ['a'..'z']
```

Introdução à Linguagem Haskell — (XVI)

- Avaliação sobre demanda, um pequeno exemplo

```
naturals :: [Int]
naturals = [0..]

teste_lazy = take 20 (tail naturals)
```

Introdução à Linguagem Haskell — (XVII)

- List Comprehensions
  - Notação similar a conjuntos para representar listas
  - Ex: $S = \{2x \mid x \in \mathbb{N} \wedge x \leq 10\}$.

```
s = [2 * x | x <- [0..], x <= 10]
```

Introdução à Linguagem Haskell — (XVIII)

- Tuplas
  - Uma n-upla é uma sequência contendo n valores de tipos possivelmente distintos.
  - Como listas são polimórficas, podemos ter listas de tuplas...
- Exemplos

```
t :: (String, Int)
t = ("blah",0)

tl :: [(String,Int)]
tl = [t]
```

Introdução à Linguagem Haskell — (XIX)

- Exemplo: Criar uma função que retorne todas as triplas pitagóricas presentes no intervalo $[1, n]$.
  - Dizemos que (a, b, c) é uma tripla pitagórica se $c^2 = a^2 + b^2$.
- Para isso, iremos utilizar list comprehensions...
- Primeiro, gerando todas as triplas, sem restrições:

```
triples' :: Int -> [(Int,Int,Int)]
triples' n = [(x,y,z) | x <- [1..n],
                        y <- [1..n],
                        z <- [1..n]]
```

Introdução à Linguagem Haskell — (XX)

- Como selecionar apenas as triplas pitagóricas?
  - Basta selecionar triplas (x, y, z) tais que $z^2 = x^2 + y^2$.

```
triples :: Int -> [(Int,Int,Int)]
triples n = [(x,y,z) | x <- [1..n],
                       y <- [1..n],
                       z <- [1..n],
                       z*z == x*x + y*y]
```

Introdução à Linguagem Haskell — (XXI)

- Haskell é uma linguagem fortemente tipada.
  - Isso quer dizer que toda expressão em Haskell possui um tipo
  - Tipos são verificados durante a compilação e evitam a execução de código "sem sentido".
- Compreender o sistema de tipos é uma parte importante da linguagem Haskell (e de outras linguagens também).
- Usando o ghci para verificar tipos:

```
ghci> :t 'a'
'a' :: Char
ghci> :t "HELLO!"
"HELLO!" :: [Char]
ghci> :t (True, 'a')
(True, 'a') :: (Bool, Char)
```

Introdução à Linguagem Haskell — (XXII)

- Funções e tipos funcionais.

```
removeUpper :: String -> String
removeUpper s = [x | x <- s, x `elem` ['A'..'Z']]

addTree :: Int -> Int -> Int -> Int
addTree x y z = x + y + z
```

- O construtor de tipos "->" associa à direita.

```
Int -> Int -> Int -> Int = Int -> (Int -> (Int -> Int))
```

Introdução à Linguagem Haskell — (XXIII)

- Alguns tipos básicos em Haskell
  - Int, Integer
  - Double, Float
  - Char, Bool
  - tuplas, listas
- Tipos definidos pelo usuário.
  - Veremos sobre isso em um encontro posterior...

Introdução à Linguagem Haskell — (XXIV)

- Variáveis de tipo
  - Representam um tipo qualquer.
  - Presença de variáveis de tipos $\Rightarrow$ tipo polimórfico.

```
ghci> :t head
head :: [a] -> a
```

Introdução à Linguagem Haskell — (XXV)

- Classes de tipos
  - Significado de 'Eq a'

```
ghci> :t (==)
(==) :: Eq a => a -> a -> Bool
```

- Diversas operações são definidas em classes de tipos em Haskell.
  - Comparações
  - Conversão de / para Strings
  - e muitas outras.

Introdução à Linguagem Haskell — (XXVI)

- Casamento de padrão
  - Mecanismo que permite definir funções "por casos".

```
mySimpleFunction :: Int -> String
mySimpleFunction 42 = "Yes! This is the answer!"
mySimpleFunction x  = "You have passed " ++ (show x)

*Main> mySimpleFunction 42
"Yes! This is the answer!"
*Main> mySimpleFunction 20
"You have passed 20"
```

Introdução à Linguagem Haskell — (XXVII)

- Casamento de padrão pode falhar...

```
dope :: String -> String
dope "homer" = "says dope"

*Main> dope "larry"
"*** Exception: Non-exhaustive patterns in function dope
```

- Situação ideal: casamento de padrão deve cobrir todos os casos!

Introdução à Linguagem Haskell — (XXVIII)

- Casamento de padrão sobre tuplas

```
addVectors :: Num a => (a,a) -> (a,a) -> (a,a)
addVectors a b = (fst a + fst b, snd a + snd b)

addVectors' :: Num a => (a,a) -> (a,a) -> (a,a)
addVectors' (x1,y1) (x2,y2) = (x1 + x2, y1 + y2)
```

Introdução à Linguagem Haskell — (XXIX)

- Casamento de padrão sobre listas

```
head' :: [a] -> a
head' [] = error "Can't call head on an empty list, dummy!"
head' (x:_) = x
```

Introdução à Linguagem Haskell — (XXX)

- Mais exemplos de casamento de padrão sobre listas

```
tell :: (Show a) => [a] -> String
tell [] = "The list is empty"
tell (x:[]) = "The list has one element: " ++ show x
tell (x:y:[]) = "The list has two elements: " ++
                 show x ++ " and " ++ show y
tell (x:y:_) = "This list is long." ++
               "The first two elements are: " ++
                show x ++ " and " ++ show y
```

Introdução à Linguagem Haskell — (XXXI)

- "As" patterns

```
capital :: String -> String
capital "" = "Empty string, whoops!"
capital all@(x:xs) = "The first letter of " ++
                     all ++ " is " ++ [x]
```

Introdução à Linguagem Haskell — (XXXII)

- Guardas
- Expressões utilizadas para teste de uma condição.
- Exemplo: cálculo de índice de massa corporal

```
imc :: RealFloat a => a -> String
imc i
  | i <= 18.5 = "Abaixo do peso ideal"
  | i <= 25.0 = "Normal"
  | i <= 30.0 = "Acima do peso ideal"
  | otherwise = "Sem comentários"
```

Introdução à Linguagem Haskell — (XXXIII)

- Cláusulas where: permitem definições locais a uma função
- Exemplo:
  - Cálculo do imc: $imc = \frac{p}{h^2}$

```
imc' :: RealFloat a => a -> a -> String
imc' p h
  | i <= 18.5 = "Abaixo do peso ideal"
  | i <= 25.0 = "Normal"
  | i <= 30.0 = "Acima do peso ideal"
  | otherwise = "Sem comentários"
  where i = p / h^2
```

Introdução à Linguagem Haskell — (XXXIV)

- Let expressions
  - Permitem definições locais, similares a cláusulas where.
- Exemplo: cálculo da área da superfície de um cilindro.

```
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
  let sideArea = 2 * pi * r * h
      topArea = pi * r ^2
  in  sideArea + 2 * topArea
```

Introdução à Linguagem Haskell — (XXXV)

- Expressões case
  - Permite realizar casamento de padrão dentro de uma expressão
  - Também pode ser simulada por uma definição local + let / where

```
describeList :: [a] -> String
describeList xs = "The list is " ++
                  case xs of
                     [] -> "empty."
                     [x] -> "a singleton list."
                     xs -> "a longer list."
```

Introdução à Linguagem Haskell — (XXXVI)

- Haskell não possui comandos de repetição
  - Devemos utilizar recursão.
- Recursão
  - Para entender recursão, devemos entender recursão. :)

Introdução à Linguagem Haskell — (XXXVII)

- Um primeiro exemplo:

```
fatorial :: Int -> Int
fatorial 0 = 1
fatorial n = n * fatorial (n - 1)
```

- Exatamente igual (a menos da sintaxe) a definição matemática:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \end{aligned}$$

Introdução à Linguagem Haskell — (XXXVIII)

- Recursão sobre listas
  - Calculando o número de elementos:

```
length' :: [a] -> Int
length' [] = 0
length' (x : xs) = 1 + length xs
```

```
 - O elemento pertence a lista?
```

```
elem' :: Eq a => a -> [a] -> Bool
elem' x [] = False
elem' x (y:ys) = x == y || elem' x ys
```

Introdução à Linguagem Haskell — (XXXIX)

- Mais recursão sobre listas
  - Concatenação e invertendo duas listas

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)

reverse :: [a] -> [a]
reverse [] = []
reverse (x : xs) = reverse xs ++ [x]
```

Introdução à Linguagem Haskell — (XL)

- Quicksort!

```
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x : xs) = small ++ [x] ++ big
                 where
                   small = qsort [y | y <- xs, y <= x]
                   big   = qsort [z | z <- xs, z > x]
```