Definindo funções simples

Programação Funcional

Prof. Rodrigo Ribeiro

Objetivos

- Definir funções em Haskell.
- Definir funções usando casamento de padrão.
- Definir funções sobre listas.

Setup inicial

```haskell
module Aula02 where

main :: IO ()
main = return ()
```

Condicionais

- If's em Haskell devem sempre possuir o else correspondente.

```
absolute :: Int -> Int
absolute n = if n < 0 then n * (-1)
             else n

signal :: Int -> Int
signal n = if n < 0 then -1
           else if n == 0 then 0
                else 1
```

Guardas

- Maneira mais elegante de expressar condicionais.
  - `otherwise` é definido como `True` e pode ser usado como caso padrão em guardas.

```
absolute1 :: Int -> Int
absolute1 n
  | n < 0 = n * (-1)
  | otherwise = n

signal1 :: Int -> Int
signal1 n
  | n < 0 = -1
  | n == 0 = 0
  | otherwise = 1
```

Casamento de padrão

```
not :: Bool -> Bool
not False = True
not True = False

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

Casamento de padrão

- Eliminando padrões desnecessários usando wildcards.
  - Se primeiro parâmetro é falso, o resultado é falso.

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

Casamento de padrão

- Se o primeiro parâmetro é verdadeiro, o resultado é igual ao segundo parâmetro.

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

Exemplo

Desenvolva funções para os operadores lógicos or e xor.

```
(||) :: Bool -> Bool -> Bool -- or
xor  :: Bool -> Bool -> Bool
```

Resposta

```haskell
(||) :: Bool -> Bool -> Bool
True || _ = True
False || b = b

xor :: Bool -> Bool -> Bool
True `xor` True = True
False `xor` False = True
_    `xor` _    = False
```

Padrões sobre tuplas

```
fst :: (a,b) -> a
fst (x, _) = x

snd :: (a,b) -> b
snd (_, y) = y
```

Exemplo

- Definida a função sobre tuplas a seguir:

```
third :: (a,b,c) -> c
```

Resposta

```haskell
third :: (a,b,c) -> c
third (_,_,z) = z
```

Exemplo

- As raízes de uma equação $ax^2 + bx + c$ são dadas pela fórmula abaixo.

$$\Delta = b^2 - 4ac \qquad x = \frac{-b \pm \sqrt{\Delta}}{2a}$$

Elabore uma função Haskell para retornar as raizes de uma equação, caso elas existam.

```
roots :: Float -> Float -> Float -> (Float, Float)
```

Resposta

```haskell
roots :: Float -> Float -> Float -> (Float, Float)
roots a b c
   = (r1, r2)
     where
       delta = b^2 - 4 * a * c
       r1    = ((-b) + sqrt delta) / 2 * a
       r2    = ((-b) - sqrt delta) / 2 * a
```

Listas

- Listas em Haskell possuem dois padrões básicos:
  - [] é a lista vazia
  - (x : xs) é uma lista com cabeça x e cauda xs.
- Tipos dos construtores de listas

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

Listas

- A lista [1,2,3,4] é apenas uma forma simplificada de escrever $1 : (2 : (3 : (4 : [])))$.
- Funções sobre listas são normalmente definidas usando casamento de padrão e recursão.
  - Nesta aula não veremos funções recursivas... :(

Funções sobre listas

- `null` testa se uma lista é vazia.

```
null :: [a] -> Bool
null [] = True
null _  = False
```

Funções sobre listas

- A seguinte função testa se uma lista possui exatamente um elemento

```
unit :: [a] -> Bool
unit [ _ ] = True
unit _ = False
```

List comprehensions

- Notação para construção de listas inspirada em teoria de conjuntos.
- Forma geral: `[expr | x <- list]`
- Exemplo:

```
addOne :: [Int] -> [Int]
addOne xs = [x + 1 | x <- xs]
```

List comprehensions

- Ex.: o conjunto $\{x^2 \mid x \in \{1, 2, 3, 4, 5\}\}$ é representado por

```
ex :: [Int]
ex = [x^2 | x <- [1,2,3,4,5]]
```

List comprehensions

- Calculando o produto cartesiano de duas listas

```
cartProd :: [a] -> [b] -> [(a,b)]
cartProd xs ys = [(x,y) | x <- xs, y <- ys]
```

List comprehensions

- Uso de guardas em list comprehensions.
- Forma geral: `[expr | x <- list, condition]`
- Exemplo:

```
sumEven :: [Int] -> [Int]
sumEven xs = [x | x <- xs, even x]
```

List comprehensions

- Calculando triplas Pitagóricas

```
triples :: Int -> [(Int,Int,Int)]
triples n = [(x,y,z) | x <- [1..n]
                     , y <- [1..n]
                     , z <- [1..n]
                     , x^2 == y^2 + z^2]
```

List comprehensions

- Casamento de padrão em list comprehensions
- Forma geral: `[expr | pattern <- list]`
- Exemplo

```
heads :: [[a]] -> [a]
heads xss = [x | (x : _) <- xss]
```

List comprehensions

- Fatorando um número inteiro.

```
factors :: Int -> [Int]
factors n = [x | x <- [1..n], n `mod` x == 0]
```

Quicksort

- Ordene recursivamente os elementos menores e maiores que o pivot.

```
qsort :: [Int] -> [Int]
qsort [] = []
qsort (x : xs) = smaller ++ [x] ++ greater
   where
     smaller = qsort [y | y <- xs, y <= x]
     greater = qsort [z | z <- xs, z > x]
```

Exercício

- Dizemos que um número n é perfeito se ele é igual a soma de seus fatores menores que n. Crie uma função Haskell que retorna todos os números perfeitos em um dado intervalo.
- Dica: Em sua solução use as funções:

```
sum [1,2,3] = 6
init [1,2,3] = [1,2]
```

Exercício

- Dizemos que um número n é primo se ele é divisível por 1 e por ele mesmo. Crie uma função Haskell que retorne todos os números primos em um dado intervalo. *Dica*: use a função `factors`.

```
primes :: Int -> Bool
```

Exercício

- Defina a função

```
max3 :: Int -> Int -> Int -> Int
```

que a partir de 3 números inteiros fornecidos como entrada, retorna o maior deles.

Exercício

- Dado um vetor $\overrightarrow{v} = (x, y)$, sua versão normalizada é calculada da seguinte forma:

$$\left(\frac{x}{\|v\|}, \frac{y}{\|v\|} \right)$$

em que $\|\overrightarrow{v}\|$ é dado por:

$$\|\overrightarrow{v}\| = \sqrt{x^2 + y^2}$$

Exercício

- Defina uma função para testar se uma lista possui 2 elementos ou menos. Faça isso de duas formas: 1) usando casamento de padrão, com uma equação para cada possibilidade e 2) usando a função `length`, de tipo:

```
length :: [a] -> Int
```