Listas em Haskell

Lucas Albertins

Departamento de Computação - UFRPE

Listas

Coleções de elementos de um mesmo tipo

Example

```
[ 1 , 2 , 3 , 4 ] :: [Int]
[(5,True),(7,True)] :: [(Int,Bool)]
[[4,2],[3,7,7,1],[],[9]] :: [[Int]]
[‘b‘,‘o‘,‘m‘] :: [Char]
"bom" :: [Char]
```

Sinônimos de tipos: `type String = [Char]`

`[]` é uma lista vazia de qualquer tipo

Estruturas de dados recursivas

Listas vs Conjuntos

A ordem dos elementos é relevante

`[1,2] /= [2,1]` , assim como `"ola" /= "alo"`

Duplicação de elementos também importa

`[True,True] /= [True]` |

O construtor de listas (:)

Outra forma de escrever listas

[5]	é o mesmo que	5:[]
[4,5]	é o mesmo que	4:(5:[])
[2,3,4,5]	é o mesmo que	2:3:4:5:[]

(:) é um construtor polimórfico

```
(:) :: Int -> [Int] -> [Int]
(:) :: Bool -> [Bool] -> [Bool]
(:) :: t -> [t] -> [t]
```

Listas

```
[2..7] = [2,3,4,5,6,7]
[-1..3] = [-1,0,1,2,3]
['a'..'d'] = ['a','b','c','d']
[2.8..5.0] = [2.8,3.8,4.8]
[7,5..0] = [7,5,3,1]
[2.8,3.3..5.0] = [2.8,3.3,3.8,4.3,4.8]
```

Exercícios

- Quantos itens existem nas seguintes listas?
  - [2,3]
  - [[2,3]]
- Qual o tipo de [[2,3]] ?
- Qual o resultado da avaliação de
  - [2,4..9]
  - [2..2]
  - [2,7..4]
  - [10,9..1]
  - [10..1]
  - [2,9,8..1]

Funções sobre listas

Problema

Somar os elementos de uma lista

```
sumList :: [Int] -> Int
```

Solução: recursão

Caso base: lista vazia

```
sumList [] = 0
```

Caso recursivo: lista possui cabeça e cauda

```
sumList (a:as) = a + sumList as
```

Avaliando

```
sumList [2,3,4,5]
 = 2 + sumList [3,4,5]
 = 2 + (3 + sumList [4,5])
 = 2 + (3 + (4 + sumList [5]))
 = 2 + (3 + (4 + (5 + sumList [])))
 = 2 + (3 + (4 + (5 + 0)))
 = 14
```

Exercícios

Defina funções sobre listas para

- dobrar os elementos de uma lista: `double :: [Int] -> [Int]`
- determinar se um valor faz parte de uma lista:
  `member :: [Int] -> Int -> Bool`
- filtrar apenas os números de uma lista:
  `digits :: String -> String`
- somar uma lista de pares: `sumPairs :: [(Int,Int)]->[Int]`

Casamento de padrões: lista

Obtendo o maior valor de uma lista de inteiros

```
maiorLista :: [Int] -> Int
maiorLista [] = minBound :: Int
maiorLista [x] = x
maiorLista (x:xs) =
 | x > maiorLista xs = x
 | otherwise = maiorLista xs
```

Apenas construtores podem ser usados para casar padrões

Expressão case

Permite casamento de padrão com valores arbitrários, ou seja, não apenas argumentos de funções

```
firstDigit :: String -> Char
firstDigit st = case (digits st) of
  [] -> '\0'
  (a:as) -> a
```

Outras funções sobre listas

Comprimento

```
length :: [t] -> Int
length [] = 0
length (a:as) = 1 + length as
```

Concatenação

```
(++) :: [t] -> [t] -> [t]
[] ++ y = y
(x:xs) ++ y = x : (xs ++ y)
```

Estas funções são polimórficas

Polimorfismo

Função possui um tipo genérico

Mesma função usada para vários tipos

Reuso de código

Uso de variáveis de tipo

Quando os tipos dos elementos não importam

```
zip :: [t] -> [u] -> [(t,u)]
zip (a:as) (b:bs) = (a,b):zip as bs
zip  _ _ = []
```

Polimorfismo

```
length [] = 0
length (a:as) = 1 + length as

rev [] = []
rev (a:as) = rev as ++ [a]

id x = x
```

Funções com variáveis instância de tipos

Polimorfismo

```
replicate 0 ch = []
replicate n ch = ch : rep (n-1) ch
```

Inferência de tipos

No GHCi

```
> :t replicate
replicate :: Int -> a -> [a]
```

Exercícios

Defina as seguintes funções

- take: devolve uma lista como os *n* primeiros elementos da lista de entrada
- drop: devolve uma lista contendo os elementos da lista de entrada, exceto pelos *n* primeiros
- takeWhile recebe uma função predicado e devolve uma lista contendo todos os elementos da lista de entrada que antecedem o primeiro para o qual a função predicado produz valor False.
  - Ex: takeWhile (>10) [14, 13, 11, 9, 23] = [14, 13, 11]
- dropWhile: ver definições de drop e takeWhile

Exemplo: Biblioteca

```
type Pessoa = String
type Livro = String
type BancoDados = [(Pessoa,Livro)]
```

Exemplo de uma base de dados

```
baseExemplo :: BancoDados
baseExemplo = [("Sergio","O␣Senhor␣dos␣Aneis"),
("Andre", "Duna"),
("Fernando", "Jonathan␣Strange␣&␣Mr.␣Norrell"),
("Fernando","Duna")]
```

Funções sobre a base de dados - consultas

```
livros :: BancoDados -> Pessoa -> [Livro]

emprestimos ::  BancoDados -> Livro ->[Pessoa]

emprestado :: BancoDados -> Livro -> Bool

qtdEmprestimos :: BancoDados -> Pessoa -> Int
```

Funções sobre a base de dados - atualizações

```
emprestar :: BancoDados -> Pessoa -> Livro -> BancoDados

devolver :: BancoDados -> Pessoa -> Livro -> BancoDados
```

Compreensão de listas

Usadas para definir listas em função de outras listas

```
doubleList xs = [2*a|a <- xs]
doubleIfEven xs = [2*a|a <- xs, isEven a]

sumPairs :: [(Int,Int)] -> [Int]
sumPairs lp = [a+b|(a,b) <- lp]

digits :: String -> String
digits st = [ch | ch <- st, isDigit ch]
```

Exercícios

Redefina as seguintes funções utilizando compreensão de listas

```
membro :: [Int] -> Int -> Bool

livros :: BancoDados -> Pessoa -> [Livro]

emprestimos ::  BancoDados -> Livro -> [Pessoa]

emprestado :: BancoDados -> Livro -> Bool

qtdEmprestimos :: BancoDados -> Pessoa -> Int

emprestar :: BancoDados -> Pessoa -> Livro ->
  BancoDados

devolver :: BancoDados -> Pessoa -> Livro ->
  BancoDados
```

Bibliografia

[1] Simon Thompson.
Haskell: the craft of functional programming.
Addison-Wesley, terceira edition, Julho 2011.

Slides elaborados a partir de originais por André Santos e Fernando Castor