

Programação Funcional

> LISTAS

Hermano Perrelli de Moura
Paradigmas de Linguagens
Computacionais

Listas

- Coleções de objetos de um mesmo tipo.
- Exemplos:
`[1,2,3,4] :: [Int]`
`[True] :: [Bool]`
`[(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 de qualquer tipo.

Listas versus Conjuntos

- A ordem dos elementos é significativa
`[1, 2] /= [2, 1]`
assim como
`"sergio" /= "oigres"`
- O número 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 :: Notação

`[2..7] = [2,3,4,5,6,7]`

`[-1..3] = [-1,0,1,2,3]`

`[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]`

Funções sobre listas

- Problema: somar os elementos de uma lista

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

- Solução: Recursão

- caso base: lista vazia []

```
sumList [] = 0
```

- caso recursivo: lista tem 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`

Outras funções sobre listas

- dobrar os elementos de uma lista
`double :: [Int] -> [Int]`
- Membership: se um elemento está na lista
`member :: [Int] -> Int -> Bool`
- filtragem: apenas os números de uma string
`digits :: String -> String`
- soma de uma lista de pares
`sumPairs :: [(Int, Int)] -> [Int]`

Outras funções sobre listas

- insertion sort

```
iSort :: [Int] -> [Int]
```

```
ins :: Int -> [Int] -> [Int]
```

- sempre existem várias opções de como definir uma função

Expressão case

- permite casamento de padrões no corpo de uma função

```
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 definição usada para vários tipos
- reuso de código
- uso de variáveis de tipos

```
zip :: [t] -> [u] -> [(t,u)]
```

```
zip (a:as) (b:bs) = (a,b):zip as bs
```

```
zip [] [] = []
```

Polimorfismo

`fst :: (t,u) -> t`
`fst (x,y) = x`

`snd :: (t,u) -> u`
`snd (x,y) = y`

`head :: [t] -> t`
`head (a:as) = a`

`tail :: [t] -> [t]`
`tail (a:as) = as`

Exemplo: Biblioteca

```
type Person = String
```

```
type Book = String
```

```
type Database = [ (Person, Book) ]
```

Exemplo de um banco de dados

```
exampleBase =  
    [ ("Alice", "Postman Pat"),  
      ("Anna", "All Alone"),  
      ("Alice", "Spot"),  
      ("Rory", "Postman Pat") ]
```


Funções sobre o banco de dados :: consultas

```
books :: Database -> Person -> [Book]
```

```
borrowers :: Database -> Book -> [Person]
```

```
borrowed :: Database -> Book -> Bool
```

```
numBorrowed :: Database -> Person -> Int
```

Funções sobre o banco de dados :: atualizações

makeLoan ::

Database -> Person -> Book -> Database

returnLoan ::

**Database -> Person -> Book ->
Database**

Compreensões 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 st]
```

Exercício

- Redefina as seguintes funções utilizando compreensão de listas:

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

```
books :: Database -> Person -> [Book]
```

```
borrowers :: Database -> Book -> [Person]
```

```
borrowed :: Database -> Book -> Bool
```

```
numBorrowed :: Database -> Person -> Int
```

```
returnLoan :: Database -> Person -> Book -> Database
```

Exemplo: Text processing (Thompson, pg. 128, Second Edition)

```
getWord :: String -> String
dropWord :: String -> String
dropSpace :: String -> String
type Word = String
splitWords :: String -> [Word]
```

Exemplo: Text processing (Thompson, pg. 128, Second Edition)

```
type Line = [Word]
getLine :: Int -> [Word] -> Line
dropLine :: Int -> [Word] -> [Word]
splitLines :: [Word] -> [Line]

fill :: String -> [Line]
fill st = splitLines (splitWords st)

joinLines :: [Line] -> String
```

Formas de Definição

- Combinando os itens: folding
 - `+`, `++`, `&&`, `maxi`
- Aplicando a todos: mapping
 - `double`, todos os segundos elementos de uma lista de pares, transformar código em nome de produto
- Seleccionando elementos: filtering
 - `digits`, procurar determinada pessoa em uma lista.