

# Listas

André Santos

(a partir de *slides* elaborados por André Santos,  
Fernando Castor e Márcio Cornélio Lopes)

# Listas

- Coleções de objetos de um **mesmo tipo**.

- Exemplos:

```
[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 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
```



# Exercícios

- Defina estas 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]

# Expressão case

- Permite casamento de padrões com **valores arbitrários**
  - Não apenas **argumentos** da 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**
  - 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
```

```
reverse [] = []
```

```
reverse (a:as) = reverse as ++  
  [a]
```

```
id x = x
```

- Funções com várias instâncias de tipo

# Polimorfismo

```
rep 0 ch = []
```

```
rep n ch = ch : rep (n-1) ch
```

- hugs/GHCI: inferência de tipos  
: **type rep**

```
Int -> a -> [a]
```

# Exemplo: Biblioteca

```
type Pessoa = String
```

```
type Livro = String
```

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

# Exemplo de um banco de dados

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



# Funções sobre o banco de dados - consultas

`livros ::`

`BancoDados -> Pessoa -> [Livro]`

`emprestimos ::`

`BancoDados -> Livro -> [Pessoa]`

`emprestado ::`

`BancoDados -> Livro -> Bool`

`qtdEmprestimos ::`

`BancoDados -> Pessoa -> Int`

# Funções sobre o banco de dados - atualizações

**emprestar ::**

**BancoDados -> Pessoa -> Livro  
-> BancoDados**

**devolver ::**

**BancoDados -> Pessoa -> Livro  
-> BancoDados**

# 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 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
```

```
devolver ::
```

```
    BancoDados -> Pessoa -> Livro -> BancoDados
```

# Exercício

- Defina uma função que ordena uma lista de inteiros utilizando o algoritmo **quick sort**

```
qSort :: [Int] -> [Int]
qSort [] = []
qSort (x:xs) =
    qSort [y | y <- xs, y < x] ++
    [x] ++
    qSort [y | y <- xs, y >= x]
```