

# 1ª Aula Prática de PLC: Tipos Básicos e Estruturados em Haskell

# Roteiro

- Tipos Básicos
- Tuplas
- Introdução às Listas
- Funções sobre Listas
- Compreensão de Listas

# Tipos Básicos

Haskell provê uma coleção de tipos primitivos e também permite tipos estruturados, definidos pelo programador, provendo grande flexibilidade na modelagem de programas.

Int / Integer:	-1, 0, 14, 500
Char:	'a', 'z', 'K', '0', '@'
Bool:	True, False
Strings:	"azul", "coelho", "IF686ec"
Float / Double:	18.021, -400.09999, 0.23333

# Exercícios Resolvidos

1. Crie uma função: `primo :: Int -> Bool`, que dado um inteiro positivo, informe se ele é, ou não, um número primo.

Exemplos:

```
Main> primo 1
```

```
False
```

```
Main> primo 251
```

```
True
```

```
Main> primo 621
```

```
False
```

# Tipos Estruturados

Haskell admite a possibilidade de que o usuário construa seus próprios tipos de dados, de acordo com as necessidades que ele tenha de simular problemas do mundo real. Os tipos estruturados são construídos a partir de outros tipos, primitivos ou estruturados. Listas e n-uplas (chamadas de tuplas na documentação em inglês da linguagem) são os tipos estruturados mais comuns.

# Tuplas

Em Haskell, Tuplas são coleções possivelmente heterogêneas (Mas também podem ser homogêneas) de dados, com tamanho fixo (Podem ser duplas, triplas, quádruplas, n-uplas).

Tupla de (Int, Int):	(1, 0)
Tupla de (Int, Bool):	(45, True)
Tupla de (Int, Int, Char):	(97, 80, 'a')
Tupla de ((Int, Bool), Int):	((79, True), 400)

# Tuplas

Funções que manipulam tuplas são normalmente definidas utilizando casamento de padrões.

Exemplos:

```
soma :: (Int, Int) -> (Int, Int) -> (Int, Int)
```

```
soma (a1, b1) (a2, b2) = (a1+a2, b1+b2)
```

```
produtoEscalar :: (Int, Int) -> (Int, Int) -> Int
```

```
produtoEscalar (a1, b1) (a2, b2) = a1*a2 + b1*b2
```

# Exercícios Resolvidos

2. Implemente a função: `divisaoEuclidiana :: Int -> Int -> (Int, Int)`, que calcula o quociente e o resto da divisão de dois números inteiros utilizando subtrações sucessivas. Na entrada, o primeiro número passado será o dividendo e o segundo, o divisor. Na saída, o primeiro número da tupla será o quociente e o segundo, o resto.

Exemplos:

```
Main> divisaoEuclidiana 24 2
```

```
(12, 0)
```

```
Main> divisaoEuclidiana 47 6
```

```
(7, 5)
```



# Introdução às Listas

Em Haskell, listas são estruturas de dados homogêneas, ou seja, armazenam vários elementos do mesmo tipo.

Lista de Int: [1, 2, 3, 4, 5]

Lista de Bool: [True, True, False]

Lista de (Int, Char): [(97, 'a'), (104, 'h'), (116, 't'), (119, 'w')]

Lista de Char: ['h', 'e', 'l', 'l', 'o']

String: "hello"

Podemos ter uma lista com inteiros ou uma lista de caracteres, porém não podemos ter uma lista com inteiros e caracteres.

Lista de Int e Char: [4, '7', 0, 'P', 'C'] --ERRO

# Introdução às Listas

Pode-se definir uma lista indicando os limites superior e inferior de um conjunto conhecido, onde existe uma relação de ordem crescente entre os elementos, no seguinte formado:

`[limiteInferior .. limiteSuperior]`

Exemplos:

`[1 .. 5] == [1, 2, 3, 4, 5]`

`[10 .. 17] == [10, 11, 12, 13, 14, 15, 16, 17]`

`[3.1 .. 7.5] == [3.1, 4.1, 5.1, 6.1, 7.1]`

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

`[9 .. 0] == []`

`['a' .. 'z'] == "abcdefghijklmnopqrstuvwxyz"`

# Introdução às Listas

Pode-se definir uma lista (estilo progressão aritmética) no seguinte formado:

`[termo1, termo2 .. limite]`

Exemplos:

`[1, 3 .. 15] == [1, 3, 5, 7, 9, 11, 13, 15]`

`['a', 'e' .. 'z'] == "aeimquy"`

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

`[16, 12 .. 3] == [16, 12, 8, 4]`

`[-5.0, 0.0 .. 15.0] == [-5.0, 0.0, 5.0, 10.0, 15.0]`

`[0.5, 0.7 .. 1] == [0.5, 0.7, 0.8999999999999999, 1.0999999999999999]`

# Funções sobre Listas

Existe um grande número de funções pré-definidas para a manipulação de listas em Haskell. Estas funções fazem parte do arquivo Prelude.hs, carregado no momento em que o sistema é chamado e permanece ativo até o final da execução.

Função	Tipo
(:)	$a \rightarrow [a] \rightarrow [a]$
(++)	$[a] \rightarrow [a] \rightarrow [a]$
(!!)	$[a] \rightarrow \text{Int} \rightarrow a$
length	$[a] \rightarrow \text{Int}$
...	...

# Funções sobre Listas:

## Operador (:)

O símbolo (:) representa o operador de construção de listas. Toda lista é construída através deste operador polimórfico.

```
Prelude> :t (:)
```

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

O operador deve ser aplicado à argumentos de um mesmo tipo.

```
Prelude> 'a':['b', 'c', 'd', 'e']
```

```
"abcde"
```

```
Prelude> 1:[10, 100, 1000, 10000]
```

```
[1, 10, 100, 1000, 10000]
```

```
Prelude> (2, True):[(1, False), (4, True), (3, False)]
```

```
[(2, True), (1, False), (4, True), (3, False)]
```

# Funções sobre Listas:

## Operador (++)

Outro operador para listas é o de concatenação (++).

```
Prelude> :t (++)
```

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

É também um operador polimórfico, entretanto só concatena listas do mesmo tipo.

```
Prelude> [1, 2, 3] ++ [4, 5, 6, 7, 8, 9]
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

```
Prelude> 'c' ++ "asa" --ERRO
```

```
Prelude> "c" ++ "asa"
```

```
"casa"
```

# Funções sobre Listas:

## Operador (!!)

O operador (!!) é utilizado para obter um elemento de uma lista pelo seu índice. Também é polimórfico.

```
Prelude> :t (!!)  
(!!) :: [a] -> Int -> a
```

O índice inicia a partir de 0.

```
Prelude> [71, 89, 127, 163, 167] !! 2  
127
```

```
Prelude> "hashtag" !! 4  
't'
```

# Funções sobre Listas:

## length

A função polimórfica `length` retorna o tamanho de uma lista, obviamente.

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

A lista vazia tem tamanho 0.

```
Prelude> length [0 .. 8]
```

9

```
Prelude> length [[2, 4, 6, 8], [3, 5, 7, 9]]
```

2

```
Prelude> length [ ]
```

0



# Exercícios Resolvidos

3. Um palíndromo é uma sequência de unidades que pode ser lida tanto da direita para a esquerda como da esquerda para a direita. Crie uma função: `palindromo :: String -> Bool`, que dada uma cadeia de caracteres de entrada, retorne se ela é um palíndromo ou não.

Exemplos:

```
Main> palindromo "12321"
```

```
True
```

```
Main> palindromo "subinoonibus"
```

```
True
```

# Funções sobre Listas:

## head e tail

Em uma lista não vazia existe sempre a cabeça (head) e a cauda (tail). Logo, as funções polimórficas head e tail retornam a cabeça e a cauda da lista, respectivamente.

```
Prelude> head "mama"
```

```
'm'
```

```
Prelude> tail "mama"
```

```
"ama"
```

```
Prelude> (head [10, 20, 30, 40, 50]):(tail [10, 20, 30, 40, 50])  
[10, 20, 30, 40, 50]
```

# Funções sobre Listas:

## init e last

Semelhantes às funções head e tail, init recebe uma lista de um tipo qualquer e a retorna toda com exceção do seu último elemento e last retorna o último elemento de uma lista.

```
Prelude> last "mama"
```

```
'a'
```

```
Prelude> init "mama"
```

```
"mam"
```

```
Prelude> init [10, 20, 30, 40, 50] ++ [last [10, 20, 30, 40, 50]]  
[10, 20, 30, 40, 50]
```

# Funções sobre Listas:

## take e drop

A função `take` recebe um número e uma lista e extrai a quantidade de elementos desde o início dessa lista. A função `drop` funciona de forma similar, só que esta descarta um número de elementos a partir do início da lista.

```
Prelude> take 3 ['b', 'c', 'd', 'e']
```

```
"bcd"
```

```
Prelude> drop 3 ['b', 'c', 'd', 'e']
```

```
"e"
```

```
Prelude> take 7 [67, 73, 79, 83, 89, 97]
```

```
[67, 73, 79, 83, 89, 97]
```

```
Prelude> take 3 [1, 2, 3, 4] ++ drop 3 [1, 2, 3, 4]
```

```
[1, 2, 3, 4]
```

# Funções sobre Listas:

## splitAt

A função `splitAt` recebe um inteiro `n` e uma lista e retorna uma dupla, onde o primeiro membro da dupla é uma nova lista que contém os `n` primeiros elementos da lista passada e o segundo membro da dupla é outra lista que contém os elementos restantes.

```
Prelude> splitAt 3 "alohomora"  
("alo", "homora")
```

```
Prelude> splitAt 1 [77, 231, 385, 539]  
([77], [231, 385, 539])
```

```
Prelude> splitAt 0 [77, 231, 385, 539]  
([], [77, 231, 385, 539])
```

# Funções sobre Listas:

## elem e notElem

As funções `elem` e `notElem` recebem um elemento e uma lista do mesmo tipo e retornam um valor booleano indicando se o elemento está presente ou não naquela lista.

```
Prelude> elem 'a' "luta"
```

```
True
```

```
Prelude> notElem 'a' "luta"
```

```
False
```

```
Prelude> elem 27 [7, 13, 11, 17, 29, 23]
```

```
False
```

```
Prelude> notElem 27 [7, 13, 11, 17, 29, 23]
```

```
True
```

# Compreensão de Listas

A definição de listas por compreensão é feita por um construtor de listas que utiliza conceitos e notações da teoria dos conjuntos. Assim, para um conjunto  $a$  temos:

$$a = [ e(x) \mid x \leftarrow \text{lista}, p_1(x), \dots, p_n(x) ]$$

Onde  $e(x)$  é uma expressão em  $x$  e os vários  $p_i(x)$  são proposições em  $x$ .

```
Prelude> [x | x <- [1..20], (x `mod` 4)==0]
```

```
[4, 8, 12, 16, 20]
```

```
Prelude> [x^2 | x <- [0..100], even x, x>=20, x<=30]
```

```
[400, 484, 576, 676, 784, 900]
```

# Compreensão de Listas

Na estrutura de compreensão de listas, as proposições  $pi(x)$  não são obrigatórias:

```
Prelude> [x*10 | x <- [5..10]]  
[50, 60, 70, 80, 90, 100]
```

Também é possível utilizar estruturas condicionais na expressão  $e(x)$ :

```
Prelude> [if (even x) then (show(x) ++ " e par!")  
else (show(x) ++ " e impar!") | x <- [0..5]]  
["0 e par!", "1 e impar!", "2 e par!", "3 e impar!", "4 e par!", "5 e  
impar!"]
```



## Exercício 4.

Implemente a função `quickSort :: [Int] → [Int]`

## Exercício 5.

Implemente a função `doisAdois :: Int → [(Int, Int)]` que receba um número inteiro  $n$  e retorne uma lista de tuplas com todas as combinações dois a dois (Considere que  $(x,y) = (y,x)$ ) do conjunto dos inteiros de 0 a  $n$ .

Exemplos:

```
Main> doisAdois 3
```

```
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
```

```
Main> doisAdois 0
```

```
[(0,0)]
```