# Paradigmas de Programação

Fabrício Olivetti de França

21 de Junho de 2018

# Listas

# Listas

- Uma das principais estruturas em linguagens funcionais.
- Representa uma coleção de valores de um determinado tipo.
- Todos os valores do **mesmo** tipo.

# Listas

Definição recursiva: ou é uma lista vazia ou um elemento do tipo genérico $a$ concatenado com uma lista de $a$.

```
data [] a = [] | a : [a]
```

(:) - concatenação de elemento com lista

# Listas

Seguindo a definição anterior, a lista `[1, 2, 3, 4]` é representada por:

```
lista = 1 : 2 : 3 : 4 :[]
```

# Listas

É uma lista ligada!!

```
lista = 1 : 2 : 3 : 4 :[]
```

A complexidade das operações são as mesmas da estrutura de lista ligada!

# Criando listas

Existem diversos *syntax sugar* para criação de listas (ainda bem):

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

# Criando listas

Faixa de valores inclusivos:

```
[1..10]   == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

# Criando listas

Faixa de valores inclusivos com tamanho do passo:

```
[0,2..10] == [0, 2, 4, 6, 8, 10]
```

# Criando listas

Lista infinita:

```
[0,2..]   == [0, 2, 4, 6, 8, 10,..]
```

# Ao infinito e além

Como o Haskell permite a criação de listas infinitas?

Uma vez que a avaliação é preguiçosa, ao fazer:

```
lista = [0,2..]
```

ele cria apenas uma **promessa** de lista.

# Ao infinito e além

Efetivamente ele faz:

```
lista = 0 : 2 : geraProximo
```

sendo `geraProximo` uma função que gera o próximo elemento da lista.

# Ao infinito e além

Conforme for necessário, ele gera e avalia os elementos da lista sequencialmente.

Então a lista infinita não existe em memória, apenas uma função que gera quantos elementos você precisar dela.

# Funções básicas para manipular listas

# Recuperar elementos

O operador `!!` recupera o i-ésimo elemento da lista, com índice começando do 0:

```
> lista = [0..10]
> lista !! 2
2
```

Note que esse operador é custoso para listas ligadas! Não abuse dele!

# Recuperar elementos

A função `head` retorna o primeiro elemento da lista:

```
> head [0..10]
0
```

# Recuperar elementos

A função `tail` retorna a lista sem o primeiro elemento (sua cauda):

```
> tail [0..10]
[1,2,3,4,5,6,7,8,9,10]
```

# Exercício

O que a seguinte expressão retornará?

```
> head (tail [0..10])
```

# Recuperar elementos

A função `take` retorna os `n` primeiros elementos da lista:

```
> take 3 [0..10]
[0,1,2]
```

# Recuperar elementos

E a função `drop` retorna a lista sem os `n` primeiros elementos:

```
> drop 6 [0..10]
[7,8,9,10]
```

# Exercício (0.5 pto)

Implemente o operador `!!` utilizando as funções anteriores.

# Tamanho da lista

O tamanho da lista é dado pela função `length`:

```
> length [1..10]
10
```

# Somatória e Produtória

As funções `sum` e `product` retorna a somatória e produtória da lista:

```
> sum [1..10]
55
> product [1..10]
3628800
```

# Concatenando listas

Para concatenar utilizamos o operador ++ para concatenar duas listas ou o : para adicionar um valor ao começo da lista:

```
> [1..3] ++ [4..10] == [1..10]
True
> 1 : [2..10] == [1..10]
True
```

# Exercício

Implemente a função `fatorial` utilizando o que aprendemos até então.

# Pattern Matching com Listas

# Pattern Matching

Quais padrões podemos capturar em uma lista?

# Pattern Matching

Quais padrões podemos capturar em uma lista?

- Lista vazia: `[]`
- Lista com um elemento: `(x : [])`
- Lista com um elemento seguido de vários outros: `(x : xs)`

E qualquer um deles pode ser substituído pelo *não importa* `_`.

# Implementando a função nulo

Para saber se uma lista está vazia utilizamos a função `null`:

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

# Implementando a função tamanho

A função `length` pode ser implementada recursivamente da seguinte forma:

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

# Exercício (0.5 pto)

Implemente a função `take`. Se `n <= 0` deve retornar uma lista vazia.

# Strings

Assim como em outras linguagens, uma `String` no Haskell é uma lista de `Char`:

```
> "Ola Mundo" == ['O','l','a',' ','M','u','n','d','o']
```

# Compreensão de Listas

# Definindo conjuntos na matemática

Na matemática, quando falamos em conjuntos, definimos da seguinte forma:

$$\{x^2 \mid x \in \{1..5\}\}$$

que é lido como *x ao quadrado para todo x do conjunto de um a cinco*.

# Compreensão de Listas

No Haskell podemos utilizar uma sintaxe parecida:

```
> [x^2 | x <- [1..5]]
[1,4,9,16,25]
```

que é lido como *x ao quadrado tal que x vem da lista de valores de um a cinco*.

# Compreensão de Listas

A expressão `x <- [1..5]` é chamada de **expressão geradora**, pois ela gera valores na sequência conforme eles forem requisitados.

Outros exemplos:

```
> [toLower c | c <- "OLA MUNDO"]
"ola mundo"
> [(x, even x) | x <- [1,2,3]]
[(1, False), (2, True), (3, False)]
```

# Compreensão de Listas

Podemos combinar mais do que um gerador e, nesse caso, geramos uma lista da combinação dos valores deles:

```
>[(x,y) | x <- [1..4], y <- [4..5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5),(4,4),(4,5)]
```

# Compreensão de Listas

Se invertermos a ordem dos geradores, geramos a mesma lista mas em ordem diferente:

```
> [(x,y) | y <- [4..5], x <- [1..4]]
[(1,4),(2,4),(3,4),(4,4),(1,5),(2,5),(3,5),(4,5)]
```

Isso é equivalente a um laço `for` encadeado!

# Compreensão de Listas

Um gerador pode depender do valor gerado pelo gerador anterior:

```
> [(i,j) | i <- [1..5], j <- [i+1..5]]
[(1,2),(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),
     (3,4),(3,5),(4,5)]
```

# Compreensão de Listas

Equivalente a:

```
for (i=1; i<=5; i++) {
  for (j=i+1; j<=5; j++) {
     // faça algo
  }
}
```

# Exemplo: concat

A função `concat` transforma uma lista de listas em uma lista única concatenada (conhecido em outras linguagens como `flatten`):

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

# Exemplo: concat

Ela pode ser definida utilizando compreensão de listas:

```
concat xss = [x | xs <- xss, x <- xs]
```

# Exercício: length

Defina a função `length` utilizando compreensão de listas! Dica, você pode somar uma lista de 1s do mesmo tamanho da sua lista.

# Guards

Nas compreensões de lista podemos utilizar o conceito de **guardas** para filtrar o conteúdo dos geradores condicionalmente:

```
> [x | x <- [1..10], even x]
[2,4,6,8,10]
```

# Divisores

Vamos criar uma função chamada `divisores` que retorna uma lista de todos os divisores de `n`. Qual a assinatura?

# Divisores

Vamos criar uma função chamada `divisores` que retorna uma lista de todos os divisores de `n`. Quais os parâmetros?

```
divisores :: Int -> [Int]
```

# Divisores

Vamos criar uma função chamada `divisores` que retorna uma lista de todos os divisores de `n`. Qual o gerador?

```
divisores :: Int -> [Int]
divisores n = [???]
```

# Divisores

Vamos criar uma função chamada `divisores` que retorna uma lista de todos os divisores de `n`. Qual o guard?

```
divisores :: Int -> [Int]
divisores n = [x | x <- [1..n]]
```

# Divisores

Vamos criar uma função chamada `divisores` que retorna uma lista de todos os divisores de `n`. Qual o guard?

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

# Divisores

```
> divisores 15
[1,3,5,15]
```

# Exercício

Utilizando a função `divisores` defina a função `primo` que retorna `True` se um certo número é primo.

# Primo

Note que para determinar se um número não é primo a função `primo` **não** vai gerar **todos** os divisores de `n`.

Por ser uma avaliação preguiçosa ela irá parar na primeira comparação que resultar em `False`:

```
primo 10 => 1 : _ == 1 : 10 : [] (1 == 1)
         => 1 : 2 : _ == 1 : 10 : [] (2 /= 10)
         False
```

# Primos

Com a função `primo` podemos gerar a lista dos primos dentro de uma faixa de valores:

```
primos :: Int -> [Int]
primos n = [x | x <- [1..n], primo x]
```

# A função `zip`

A função `zip` junta duas listas retornando uma lista de pares:

```
> zip [1,2,3] [4,5,6]
[(1,4),(2,5),(3,6)]

> zip [1,2,3] ['a', 'b', 'c']
[(1,'a'),(2,'b'),(3,'c')]

> zip [1,2,3] ['a', 'b', 'c', 'd']
[(1,'a'),(2,'b'),(3,'c')]
```

# Função `pairs`

Vamos criar uma função que, dada uma lista, retorna os pares dos elementos adjacentes dessa lista, ou seja:

```
> pairs [1,2,3]
[(1,2), (2,3)]
```

# Função `pairs`

A assinatura será:

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

# Função `pairs`

E a definição será:

```
pairs :: [a] -> [(a,a)]
pairs xs = zip xs (tail xs)
```

# Exercício (0.5 pto)

Utilizando a função `pairs` defina a função `sorted` que retorna verdadeiro se uma lista está ordenada. Utilize também a função `and` que retorna verdadeiro se **todos** os elementos da lista forem verdadeiros.

```
sorted :: Ord a => [a] -> Bool
```