Listas

MCTA016-13 - Paradigmas de Programação

Emilio Francesquini
e.francesquini@ufabc.edu.br

2019.Q2

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de **Paradigmas de Programação na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Conteúdo baseado no texto preparado, e gentilmente cedido, pelo Professor Fabrício Olivetti de França da UFABC.

Listas

Listas

UFABC

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

UFABC

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

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

- `(:)` - operador de concatenação de elemento com lista
  - Lê-se: *cons*

UFABC

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

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

UFABC

É uma lista ligada!!

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

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

Criando listas

UFABC

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

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

Criando listas

UFABC

Faixa de valores inclusivos:

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

Criando listas

UFABC

Faixa de valores inclusivos com tamanho do passo:

```
1 [0,2..10] == [0, 2, 4, 6, 8, 10]
```

Criando listas

UFABC

Lista infinita:

```
1 [0,2..]   == [0, 2, 4, 6, 8, 10,..]
```

Ao infinito e além

UFABC

Como o Haskell permite a criação de listas infinitas?

Uma vez que a avaliação é preguiçosa, ao fazer:

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

ele cria apenas uma promessa de lista.

UFABC

Efetivamente ele faz:

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

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

Ao infinito e além

UFABC

- 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 manipulação de listas

Recuperar elementos - `!!`

UFABC

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

```
1 > lista = [0..10]
2 > lista !! 2
3 2
```

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

Recuperar elementos - head

UFABC

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

```
1 > head [0..10]
2 0
```

Recuperar elementos - `tail`

UFABC

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

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

Exercício 1

UFABC

O que a seguinte expressão retornará?

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

Recuperar elementos - take

UFABC

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

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

Recuperar elementos - `drop`

UFABC

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

```
1 > drop 6 [0..10]
2 [7,8,9,10]
```

Exercício 2

UFABC

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

Exercício 2 - Solução

UFABC

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

```haskell
xs !! n = head (drop n xs)
```

Tamanho da lista - `length`

UFABC

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

```
1 > length [1..10]
2 10
```

UFABC

- As funções `sum` e `product` retornam a somatória e produtória de uma lista:

```
1 > sum [1..10]
2 55
3 > product [1..10]
4 3628800
```

Pergunta

Quais tipos de lista são aceitos pelas funções `sum` e `product`?

Concatenando listas

UFABC

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
```

- **Atenção** à complexidade do operador `++`

Exercício 3

UFABC

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

Pattern Matching com Listas

Quais padrões podemos capturar em uma lista?

UFABC

Quais padrões podemos capturar em uma lista?

- Lista vazia:
  - `[]`
- Lista com um elemento:
  - `(x : [])` ou `[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

UFABC

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

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

Implementando a função tamanho

UFABC

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

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

Exercício 4

UFABC

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

UFABC

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

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

Compreensão de Listas

Definindo conjuntos na matemática

UFABC

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*.

UFABC

No Haskell podemos utilizar uma sintaxe parecida:

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

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

UFABC

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)]
```

UFABC

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

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

UFABC

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!

UFABC

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)]
```

UFABC

Equivalente a:

```
1  for (i=1; i<=5; i++) {
2    for (j=i+1; j<=5; j++) {
3       // faça algo
4    }
5  }
```

Exemplo: concat

UFABC

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

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

Exemplo: concat

UFABC

Ela pode ser definida utilizando compreensão de listas:

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

Exercício 5 - `length`

UFABC

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]
```

Exemplo: Divisores

UFABC

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

1. Qual a assinatura?

Exemplo: Divisores

UFABC

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

1. Qual a assinatura?
2. Quais os parâmetros?

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

Exemplo: Divisores

UFABC

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

1. Qual a assinatura?
2. Quais os parâmetros?
3. Qual o gerador?

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

Exemplo: Divisores

UFABC

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

1. Qual a assinatura?
2. Quais os parâmetros?
3. Qual o gerador?
4. Qual o guard?

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

Exemplo: Divisores

UFABC

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

1. Qual a assinatura?
2. Quais os parâmetros?
3. Qual o gerador?
4. Qual o guard?

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

Exemplo: Divisores

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

Exercício 6

UFABC

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

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

Primos

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

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

A função `zip`

UFABC

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`

UFABC

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

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

Função `pairs`

UFABC

A assinatura será:

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

Função pairs

UFABC

E a definição será:

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

Exercício 7

UFABC

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.

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