

Paradigmas de Programação

Prof. Maicon R. Zatelli

Haskell - Programação Funcional
Listas

Universidade Federal de Santa Catarina
Florianópolis - Brasil
2018/1

Construtores de listas

- `:`
- `[]`
- `..`

Haskell

Exemplos

- Lista vazia
`[]`
- Constrói uma lista de inteiros
`2:4:6:8:[]`
Resulta em `[2, 4, 6, 8]`
- Constrói uma lista de caracteres
`'u': 'f': 's': 'c': []`
Resulta em `['u', 'f', 's', 'c'] = "ufsc"`
- Constrói uma lista de tuplas
`(1, 'a'):(2, 'b'):[]`
Resulta em `[(1, 'a'), (2, 'b')]`

- Constrói uma lista de listas

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

Resulta em `[[1,2,3,4], [6,7,8]]`

`"ufsc":"floripa":"praia":[]`

Resulta em `["ufsc","floripa","praia"]`

Forma explícita

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

Forma implícita

[1..10]

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

Forma implícita

Exemplos:

- PA de 0 a 20, razão 2
[0,2..20]
Resulta em: [0,2,4,6,8,10,12,14,16,18,20]
- PA de 1 a 20, razão 3
[1,4..20]
Resulta em: [1,4,7,10,13,16,19]

As listas implícitas podem ser representadas de forma

- **finita**

`[1..100]`, `[10,20..100]`

- ou **infinita**

`[1..]`, `[10,20..]`

Existe uma forma genérica de representação de listas para argumentos de funções

- $(a:b)$, onde a é o primeiro elemento da lista (ou cabeça da lista), e b é o resto da lista (ou cauda da lista).

Cabeça da Lista

```
cabeca :: [t] -> t  
cabeca (a:_) = a
```

Comprimento de uma lista

```
comprimento :: [Int] -> Int  
comprimento [] = 0  
comprimento (_:b) = 1 + (comprimento b)
```

Igualdade entre duas listas

```
igual :: [Int] -> [Int] -> Bool
igual [] [] = True
igual [] _ = False
igual _ [] = False
igual (a:b) (c:d) | (a == c) = igual b d
                  | otherwise = False
```

Dobra o valor dos elementos de uma lista

```
dobro :: [Int] -> [Int]
dobro [] = []
dobro (a:b) = a * 2 : dobro b
```

Ordena os valores de uma lista

```
ordenacao :: [Int] -> [Int]
ordenacao [] = []
--adiciona cada elemento na lista ordenada
ordenacao (a:b) = add a (ordenacao b)

add :: Int -> [Int] -> [Int]
add a [] = [a]
add a (b:c) | (a <= b) = a : b : c --vide bullets
             | otherwise = b : (add a c) --
```

A função add funciona da seguinte forma:

- se a lista é vazia, então o resultado da função add é uma lista apenas com o elemento a
- se $a \leq b$, então a torna-se o primeiro elemento da lista, depois b e c
- caso contrario, b é o primeiro elemento e depois tenta adicionar a na lista restante, que é c

Outros operadores com listas:

- Concatenação: `++`
`[1,2,3,4]++[5,6]`
Resulta em `[1,2,3,4,5,6]`
- Retorna um elemento na posição do segundo argumento: `!!`
`[1,2,3,4,5,6]!!3`
Resulta em 4, pois retorna o elemento da posição 3
- Subtração de listas: `\\` (Necessita: `import Data.List`)
`[1,2,2,3,4,5] \\ [2,3,4]`
Resulta em `[1,2,5]`

List comprehension

- Definição de listas de maneira matemática:
 $[x \mid x \leftarrow [1, 2, 3]]$
 Resulta em: $[1, 2, 3]$

List comprehension

```
maioresQue :: [Int] -> Int -> Int -> [Int]
maioresQue lista p q = [b | b <- lista, maiorQue b, menorQue b]
  where
    maiorQue x = x > p
    menorQue x = x < q
```

```
print (maioresQue [1,7,9,4,5,6,7,4,12] 4 9)
```

- Retorna uma lista com todos os números entre 4 e 9.

List comprehension

```
gerarPares :: [t] -> [u] -> [(t,u)]  
gerarPares 11 12 = [(a,b) | a <- 11, b <- 12]
```

```
putStrLn (show (gerarPares [1,2,3] [4,5]))
```

- Retorna uma lista de duplas, combinando todos os elementos de [1,2,3] com todos os elementos de [4,5], sendo que os elementos 1,2,3 sempre aparecem no primeiro termo da dupla.

List comprehension

- Definição de listas usando expressões lambda:

`(\x->x) [1,2,3,4]`

Resulta em: `[1,2,3,4]`

* Voltaremos ao assunto de expressões lambda em Haskell mais tarde ...

Funções de funções sobre listas

map: aplica uma função sobre cada elemento de uma lista e retorna a lista resultante.

```
map abs [-1,2,-5,3,-8]
```

Retorna [1,2,5,3,8]

Funções de funções sobre listas

filter: aplica uma função sobre cada elemento de uma lista e retorna a lista na qual o resultado da aplicação da função sobre o elemento é verdadeiro.

```
filter odd [1,2,3,4,5,6]
```

Retorna [1,3,5]

Haskell

```
filtrar :: (t -> Bool) -> [t] -> [t]
filtrar f [] = []
filtrar f (a:b) | f a = a: (filtrar f b)
                | otherwise = filtrar f b
```

- A função filtrar acima aplica a função *f* em *a* e se retornar *True*, adiciona *a* na lista e prossegue fazendo isso recursivamente na cauda da lista, até que não haja mais nenhum elemento.

Haskell - Alguns Links Úteis

- <http://learnyouahaskell.com/chapters>
- <http://haskell.tailorfontela.com.br/chapters>
- https://wiki.haskell.org/Haskell_in_5_steps

Ver atividade no Moodle