Programação Funcional
Listas

Sérgio Soares
scbs@cin.ufpe.br

Listas

- Coleções de objetos de um mesmo tipo.
- Exemplos:

```
[1,2,3,4] :: [Int]
[True] :: [Bool]
[(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 de qualquer tipo.

Listas vs. Conjuntos

- A ordem dos elementos é significante

  `[1,2] /= [2,1]`

  assim como

  `"sergio" /= "oigres"`

- O número 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]`

- `[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]`

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 no corpo de uma 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

```
zip :: [t] -> [u] -> [(t,u)]
zip (a:as) (b:bs) = (a,b):zip as bs
zip [] [] = []
```

Polimorfismo

```
fst :: (t,u) -> t      snd :: (t,u) -> u
fst (x,y) = x          snd (x,y) = y

head :: [t] -> t       tail :: [t] -> [t]
head (a:as) = a        tail (a:as) = as
```

Exemplo: Biblioteca

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

Exemplo de um banco de dados

```
baseExemplo :: BancoDados
baseExemplo =
    [(”Sergio”,”Olga”),
     (”Andre”,”Senna”),
     (”Ricardo”,”O Guarani”),
     (”Andre”,”Olga”)]
```

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 st]
```

Exercícios

- Redefina as seguintes funções utilizando compreensão de listas

```
membro :: [Int] -> Int -> Bool
livros :: BancoDados -> Pessoa -> [Livro]
emprestimos :: BancoDados -> Livro ->[Pessoa]
emprestado :: Database -> 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]
```

Exemplo: Text processing

```
getWord :: String -> String
dropWord :: String -> String
dropSpace :: String -> String
type Word = String
splitWords :: String -> [Word]
```

Exemplo: Text processing

```
type Line = [Word]
getLine :: Int -> [Word] -> Line
dropLine :: Int -> [Word] -> [Word]
splitLines :: [Word] -> [Line]

fill :: String -> [Line]
fill st = splitLines (splitWords st)

joinLines :: [Line] -> String
```

Formas de Definição

- Combinando os itens: folding
  - **+, ++, &&, maxi**
- Aplicando a todos: mapping
  - **double**, todos os segundos elementos de uma lista de pares, transformar código em nome de produto
- Selecionando elementos: filtering
  - **digits**, procurar determinada pessoa em uma lista.