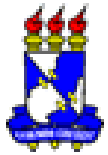
Tuplas e Listas

Prof. Alberto Costa Neto
alberto@ufs.br

Linguagens de Programação



Departamento de Computação
Universidade Federal de Sergipe

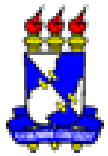
Conteúdo

- Tuplas
- Listas
  - Construção de Listas
  - Operações sobre Listas (Head, Tail, ...)
  - Compreensão de Lista
  - Funções de Alta-Ordem + Filter, Map e Fold

Tuplas

Prof. Alberto Costa Neto
alberto@ufs.br

Linguagens de Programação



Departamento de Computação
Universidade Federal de Sergipe

Tuplas

- Construído a partir de componentes de tipos mais simples:

  (t1,t2,...,tn) consiste em tuplas de valores (v1,v2,...,vn) onde, v1::t1, v2::t2, ..., vn::tn

- Assemelham-se aos registros de Pascal, mas os componentes **não possuem nome**: o que importa é a **posição**

Construindo um tipo Tupla

```
type Endereco = (String, String, Int)
e1 :: Endereco
e1 = (“Rua",“Dom José Tomás", 57)

type Data = (int,int,int)
hoje :: Data
hoje = (03,06,2008)
```

Equivalência Estrutural de Tipos

```
type Endereco = (String, String, Int)
e1 :: Endereco
e1 = (“Rua",“Dom José Tomás", 57)

e2 :: (String, String, Int)
e2 = e1  --  É válido?
```

Funções e Tuplas

```
maxEmin :: Int -> Int -> (Int,Int)
maxEmin x y
   | x >= y = (x,y)
   | otherwise = (y,x)
```

maxEmin (5, 9) => Retorna (9, 5)

maxEmin (4, 3) => Retorna (4, 3)

```
troca :: (Int,Int) -> (Int,Int)
troca (a,b) = (b,a)
```

troca (10, 20) => Retorna (20, 10)

Casamento de Padrões

- Funções sobre tuplas são normalmente definidas por meio de casamento de padrão
- Tipos de padrão:
  - uma **variável**
  - um **wildcard** “_”
  - uma **constante literal**
  - uma **tupla** do tipo (p1, p2, … , pn)

Exemplo de Wildcards e Variáveis

type Aluno = (String, String)

get_matricula (m , _) = m

get_nome (_ , n) = n

set_nome (m , _) nome = (m, nome)

Tuplas e vínculos

mova (5.2,3.1) 10.0 (pi/3)

```
mova :: Pos -> Dist -> Ang -> Pos
mova :: (Float,Float) -> Float -> Float -> (Float,Float)
mova (x,y) d a = (x + d * cos a, y + d * sin a)
```

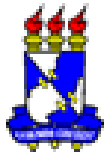
Vínculos produzidos:

x = 5.2, y = 3.1, d = 10.0, a = 1.04719

Listas

Prof. Alberto Costa Neto
alberto@ufs.br

Linguagens de Programação



Departamento de Computação
Universidade Federal de Sergipe

Listas

- Coleção homogênea de itens
- Para qualquer tipo t, existe um tipo em Haskell denotado por [t] e chamado de lista de t
- Notação: [e1,e2,e3,e4]
- Exemplos:

  [ ] é uma lista vazia

  [1,2,3,4,1,4] é uma lista do tipo [Int]

  [True] é uma lista do tipo [Bool]

  ['a','a','b'] = é uma lista de [Char]

  [[12,2],[2,12],[]] = é uma lista de listas do tipo [[Int]]

Listas e Strings

- Uma String em Haskell é na verdade uma lista de Char
- São portanto equivalentes
- Exemplos:

  ['a','a','b'] ≡ "aab" (uma String)

  [Char] ≡ String

  [“abc”, “def”] ≡ [['a','b','c'], ['d','e','f']]

Construção de Listas

- Haskell provê um **operador** (:) para construção de listas (cons)
- Uma expressão x:xs constrói uma lista cuja **cabeça** é o **valor x** e a **cauda**, a **lista xs**
- Exemplos:

```
1 : [2,3] == [1,2,3]
'l' : ['i','s','t','a'] == ['l','i','s','t','a']
'l' : "ista" == "lista"
[1] : [2,3] type error!
"l" : "ist" type error!
```

Geração de Listas

- Haskell oferece um gerador de listas com a sintaxe [n .. m] é a lista [n,n+1,...,m]
  - Se **n > m**, a lista é **vazia**
  - **[n,p .. m]** é a lista cujos dois primeiros elementos são n e p e o último é m, com os elementos em **passos de tamanho p - n**

Gerador de Listas

[2..7] é a lista [2,3,4,5,6,7]

[9,7 .. 0] é a lista [9,7,5,3,1]

[3.1 .. 7.1] é a lista [3.1,4.1,5.1,6.1,7.1]

['a' .. 'm'] é a string "abcdefghijklm"

['a','c' .. 'm'] é a string "acegikm"

Funções head e tail

- **head** obtém a cabeça de uma lista xs

```
head :: [t] -> t
head (x:xs) = x
head [] = error
          “empty list”
```

head [1,2,3] => retorna 1

head "lista" => retorna 'l'

tail obtém a cauda de uma lista xs

```
tail :: [t] -> [t]
tail (x:xs) = xs
tail [] = []
```

tail [1,2,3] => retorna [2,3]

tail "lista" => retorna "ista"

Comprimento e Concatenação

- **length** retorna o comprimento de uma lista xs

```
length :: [t] -> Int
length [ ] = 0
length (x:xs) = 1 +
                length xs
```

[1,2,3] => retorna o valor 3

[“lista”] => retorna o valor 5

- O operador ++ efetua a concatenação de duas listas xs e ys

```
(++) :: [t] -> [t] -> [t]
(++) [] y = y
(++) (x:xs) y =
                x : (++) xs y
```

[1,2] ++ [3] == [1,2,3]

"li" ++ "sta" == "lista"

Compreensão de Listas

- Construção de listas a partir de conceitos e notações da teoria de conjuntos
- Por exemplo, para uma lista de valores que satisfaz a função abaixo:

$A = \{x^2 \mid X \in N \text{ ^ } x \text{ é ímpar ^ } x < 100\}$

- Teríamos em Haskell:

```
[x*x | x <- [1..100], mod x 2 /= 0]
```

Notação: [expressão | v <- lista]

Exemplos de Compreensão de Listas

- Gera uma [Bool] contendo **True** se o valor correspondente na lista é **par** e **False caso contrário**

```
[ mod n 2 == 0 | n <- [ 2,4,4,7,9] ]
```

Retorna => [True,True,True,False,False]

Exemplos de Compreensão de Listas

- Gera uma [Int] com os valores resultantes da soma dos pares de valores das tuplas (Int,Int) contidas na lista [(Int,Int)] passada

```
somaPares :: [(Int,Int)] -> [Int]
somaPares li = [ a+b | (a,b) <- li]

somaPares [(2,3),(4,5)]
```

Retorna => [5,9]

QuickSort c/ Compreensão de Listas

```
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y < x]
               ++ [x]
               ++ qsort [y | y <- xs, y >= x]
```

qsort [10, 5, 1, 2, 4] => Retorna [1,2,4,5,10]

Função de alta-ordem (Filter)

```
filter :: (t -> Bool) -> [t] -> [t]
filter f [] = []
filter f (x : xs) = if f x then x : filter f xs
                           else filter f xs
```

- filter f computa uma função nova que filtra os componentes de uma dada lista, mantendo apenas aqueles componentes x para os quais f x retornou True
- Por exemplo, "filter odd" retorna uma função, do tipo [Int] -> [Int], que filtra a lista [2,3,5,7,11] gerando [3, 5, 7, 11]

Filter - Exemplo

- Outra implementação de filter (usando compreensão de listas)

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [b | b <- xs, p b]
```

filter isLower ['P','l','p'] => Retorna “lp”

filter isLower “Haskell” => Retorna “askell”

Função de alta-ordem (Map)

```
map :: (s -> t) -> [s] -> [t]
map f [] = []
map f (x : xs) = f x : map f xs
```

- map f gera uma nova função que aplica a função f separadamente para cada componente de uma lista dada
- Por exemplo, "map odd" é uma função, do tipo [Int] -> [Bool], que mapeia a lista [2,3,5,7,11] na lista [false, true, true, true, true]

Map - Exemplo

```
map length [“PLP”, “ES”, “RC”, “Algoritmos”]
    => Retorna [3,2,2,10]

map (\x -> x > 0) [-10, -5, 0, 10, 20]
      => Retorna [False,False,False,True,True]
```

Função de alta-ordem (Fold)

```
fold :: (a -> a -> a) -> [a] -> a
fold f [x] = x
fold f (x:xs) = f x (fold f xs)
```

- fold f gera uma nova função que aplica a função f para cada componente de uma lista dada e retorna um único valor
- Por exemplo, "fold +" gera uma função, do tipo [Int] -> Int, que efetua a soma dos valores de uma lista [2,3,5] e gera o valor 10

Fold / Redução - Exemplos

fold (*) [10,20,30] => retorna 6000

fold (++) [“15”, “/”, “04”, “/”, “2011”]
=> retorna “15/04/2011”

```
fatorial :: Int -> Int
fatorial n = fold (*) [1..n]
```

Sugestões de Leitura

- Concepts of Programing Languages (Robert Sebesta)
  - Seção 15.8
- Programming Language Concepts and Paradigms (David Watt)
  - Seções 14.1 a 14.3.4
- Haskell: Uma Abordagem Prática (Cláudio Sá)
  - Capítulos 4, 5, 9 e 13