Programação Funcional

(COMP0393)

Leila M. A. Silva

dcomp
departamento de computação

Tuplas e Listas

(COMP0393)

Aula 4

dcomp
departamento de computação

Tuplas

- Servem para combinar elementos de tipos diferentes em uma mesma estrutura.
- Ex: (“Pedro”, 22, 75.2) é uma tupla (String, Int, Float)
- Notação geral:

  tuplas de tipos $(t_1, t_2, \ldots, t_n)$

  instância $(v_1, v_2, \ldots, v_n)$

  onde $v_1 :: t_1, v_2 :: t_2, \ldots v_n :: t_n$,

Tuplas

- Escreva um função que dados dois números inteiros, retorna o maior e o menor valores
- O retorno será na forma de uma tupla (menor, maior)

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

Tuplas

- Casamento de Padrão em Tuplas
  - Suponha uma função para adicionar os elementos de um par.

```
somaPar :: (Int, Int)-> Int
somaPar (x,y) = x+y
```

  - Usando casamento de padrão poderia escrever

```
somaPar :: (Int, Int)-> Int
somaPar (0,y) = y
somaPar (x,y) = x+y
```

dcomp departamento de computação

Tuplas

- Casamento de Padrão em Tuplas
  - Argumento podem ser casados com padrões

```
nome :: (String, (Int, Float)) -> String
nome (n, p) = n
```

```
> nome ("Maria", (25, 1.75))
"Maria"
```

dcomp departamento de computação

Tuplas

- Casamento de Padrão em Tuplas
  - Aqui também podemos usar _ quando não interessa uma parte do padrão

```
nome :: (String, (Int, Float)) -> String
nome (n, _) = n

idade :: (String, (Int, Float)) -> Int
idade (_, (id, _)) = id

altura :: (String, (Int, Float)) -> Float
altura (_, (_, a)) = a
```

```
> idade ("Maria", (25, 1.75))
25
```

dcomp departamento de computação

Tuplas

- Casamento de Padrão em Tuplas
  - Podemos usar casamento de padrão dentro da cláusula **where**

```
imc :: Float -> Float -> String
imc peso altura
    | razao < magro  = "Abaixo do peso"
    | razao < normal = "Peso normal"
    | razao < gordo  = "Sobrepeso"
    | otherwise      = "Obesidade"
    where razao  = peso / altura ^ 2
          (magro, normal, gordo)  = (18.5, 25.0, 30.0)
```

Tuplas

- Para pares, Haskell provê seletores:
  - fst – devolve o primeiro elemento do par
  - snd – devolve o segundo elemento do par

```
fst (x,y) = x
snd (x,y) = y

somaPar :: (Int, Int) -> Int
somaPar p = fst p + snd p
```

```
somaPar (3,5) ⇝  fst (3,5) + snd (3,5) ⇝
3 + snd (3,5) ⇝ 3 + 5 ⇝ 8
```

Exercício de Fixação

- Dados dois inteiros elabore uma função para retornar o máximo e o número de vezes que ele ocorre

Exercício de Fixação

- Dado dois inteiros elabore uma função para retornar o máximo entre eles e o número de vezes que ele ocorre

```
maxOcorre :: Int -> Int -> (Int,Int)
maxOcorre x y
   | x == y = (x, 2)
   | otherwise = ((max x y), 1)
```

dcomp departamento de computação

Exercícios Recomendados

- Usando `maxOcorre` defina uma função similar só que agora para três números.
- Dada uma tupla com três elementos inteiros, elabore uma função para devolver uma tupla com os elementos em ordem.

Listas

- É uma coleção de itens de **mesmo tipo**.
- Ex:
  - [Int] – listas de inteiros
  - [(Int,Int)] – listas de pares de inteiros
- A lista de elementos do tipo Char é um sinônimo de String
  - [Char] é equivalente a String
    - `['a', 'b', 'c']::[Char]`
    - `['a', 'b', 'c']:: String`
    - `"abc":: String`
- Para definirmos sinônimos de tipos em geral usamos o comando `type`
  - Ex:
```
type Pessoa = String
type Idade = Int
type Peso = Float
```
- Podemos também ter listas de listas, listas de funções, etc..
  - Ex:
```
[[10,14],[1,2,3],[]] :: [[Int]]
[quadX, dobro] :: [Int->Int]
```
- `[1,True]` dá erro! Todos os elementos precisam ser do mesmo tipo

Listas

- Assim como em Matemática, a ordem e a repetição importa nas listas
  - [1,2,2], [2,1,2] e [1,2,2,2] são listas diferentes embora os valores dos elementos sejam sempre 1 e 2.
- Listas podem ser escritas como:
  - [n..m]-lista equivalente à [n, n+1, n+2, ...,m]
  - [3..7] = [3,4,5,6,7]
  - [3.1 ..7.0] =[3.1,4.1,5.1,6.1]
  - [n,p..m] – lista equivalente a [n, p, 2p-n, 3p-2n,...,m]
  - [1,3,9] = [1,3,5,7,9]
  - [‘a’,’c’..’n’] = “acegikm”

Exercícios de Fixação

- Escreva a lista dos pares entre 1 e 11.
- Escreva uma lista de triplas do tipo (Int, Float,Char).

Listas - Compreensões

- Descreve uma lista em termos dos elementos de uma outra lista. Esta facilidade é particular do paradigma funcional.
- Ex: Suponha que dado uma lista `ex`, digamos, `[2,4,7]` você deseje escrever uma lista cujos elementos são o dobro dos elementos da lista `ex`. Então faça:

```
[2*n | n<-ex]
```

  - Leitura: Para todo `n` em `ex` construa a lista contendo `2*n` (respeitando a ordem dos elementos em `ex`)
  - Esta lista é equivalente a `[4,8,14]`.
- O termo `n<-ex` é chamado de **gerador**.

Listas - Compreensões

- Mais exemplos considerando `ex = [2,4,7]`
  - Lista de Booleanos que sinalizam se os valores de ex são ou não pares

```
[ehPar n | n<-ex]
```

onde

```
ehPar :: Int -> Bool
ehPar x = (x `mod` 2 == 0)
```

  - Esta lista é equivalente a `[True, True, False]`.

Listas - Compreensões

- Podemos combinar o gerador com um ou mais *testes* cujos resultados sejam Booleanos.
- Considerando `ex = [2,4,7]` crie a lista contendo o dobro dos elementos pares de `ex`, que sejam maiores que 3.

testes

```
[2*n | n<-ex, ehPar n, n>3]
```

  - Esta lista é equivalente a `[8]`.
  - Avaliação:

```
[2*n | n<-[2,4,7], ehPar n, n>3] ⇝
          n =   2    4    7
   ehPar n = True  True False
     n > 3 = False True
     2*n   =       8
```

Listas - Compreensões

- Não é necessário ser uma variável do lado esquerdo do gerador, pode ser um padrão.

```
somaPares :: [(Int,Int)] -> [Int]
somaPares lista = [x+y| (x,y)<-lista]
```

  - Avaliação:

```
[x+y | (x,y)<-[(2,4),(8,7)]] ~
          x =   2 8
          y =   4 7
        x+y =   6 15
```

Resultado

```
[6, 15]
```

Listas - Compreensões

- Podemos também combinar testes neste caso.

```
somaOrdPares :: [(Int,Int)] -> [Int]
somaOrdPares lista = [x+y| (x,y)<-lista, x<y]
```

  - Avaliação:

```
[x+y | (x,y)<-[(2,4),(8,7)]] ⇝
          x =   2     8
          y =   4     7
        x<y =   True  False
```

Resultado

```
[6]
```

Listas - Compreensões

- Compreensões podem funcionar como filtros.

```
digits :: String-> String
digits st = [ch| ch <-st, isDigit ch]
```

  - Avaliação:

```
[ch | ch <- "N10"] ⇝
          ch =    'N'  '1'   '0'
 isDigit ch =  False True True    7
```

Resultado

"10"

- `isDigit::Char -> Bool` é uma função pré-definida em Haskell que dado uma caracter sinaliza se é ou não um dígito.

Listas - Compreensões

- É possível usar compreensões como parte de funções.
- Ex: Desejo construir uma função para checar se todos os elementos de uma lista são pares

```
todosPares :: [Int]-> Bool
todosPares xs = (xs == [x| x <-xs, ehPar x])
```

Listas - Compreensões

- Podemos usar casamento de padrões dentro de compreensões, bem como definir funções locais dentro do **where**.

```
calcImc :: [(Float, Float)] -> [Float]
calcImc xs = [imc w h | (w, h) <- xs]
    where imc peso altura = peso / altura ^ 2
```

```
calcImc [(70.0, 1.80), (54.0, 1.64)] ↝
   [imc w h | (w,h)<- [(70.0, 1.80), (54.0, 1.64)] ↝
            w =   70.0            54.0
            h =   1.80            1.64
      imc w h =   70.0/1.8^2      54.0/1.64^2

                  [21.60, 20.07]
```

Listas - Compreensões

- Exercício de Fixação
  - Ex: Usando a função `ehPar` e compreensão elabore uma função para checar se todos os elementos são ímpares

Listas - Compreensões

- Exercício de Fixação
  - Ex: Usando a função `ehPar` e compreensão elabore uma função para checar se todos os elementos são ímpares

```
todosImpares :: [Int]-> Bool
todosImpares xs = ([] == [x| x <-xs, ehPar x]
```

Listas - Compreensões

- Exercícios Recomendados
  - Escreva uma função para dada uma lista de inteiros, seleciona os ímpares desta lista e devolve uma lista em que os elementos ímpares aparecem triplicados.
  - Escreva uma função que dada uma String, devolve outra String em que as letras minúsculas foram transformadas em maiúsculas e os demais caracteres permanecem iguais. Você pode usar as função `paraMaiusculo`, já vista, na sua solução.

Show e Read

- O tipo String é um tipo especial de lista, sinônimo de [Char]
- É possível escrever caracteres especiais como parte da string. Ex: “gato\ne\nrato\n”
- Outros exemplos: “\99d\101\n”
- Para visualizar strings escritas desta forma use

```
>putStr “\99d\101\n”
>”cde”
```

- Cuidado!
  - a é uma variável
  - ‘a’ é um caracter
  - “a” é uma string

Show e Read

- Existem duas funções pré-definidas em Haskell que convertem strings para valores e vice-e-versa:
  - show: converte um valor em uma string correspondente.
    - Ex: `show (2+3)↝"5"; show (True||False) ↝"True"`
  - read: converte uma string em um valor
    - Ex: `read "8.2" + 3.8 ↝ 12.0;`

      `read "3" :: Int ↝ 3;`

    Obs: (usando no modo iterativo do GHCi precisa fazer uma anotação de tipo; dentro de expressões e funções não)

Listas – Operadores e Funções Básicas

- Haskell possui vários operadores e funções pré-definidas sobre listas.
- Operador ++ - junta duas listas

  `"Boa"++" tarde!" ↝ "Boa tarde!"`
- Construtor : - adiciona um elemento no início de uma lista

  `2:[4,5] ↝ [2,4,5]`
- Operador `!!` – `xs!!n` retorna o `n`-ésimo elemento da lista `xs`, iniciando do primeiro elemento da lista e contando desde zero.Ex:

  `[2, 4, 5] !! 2 ↝ 5`

Listas – Operadores e Funções Básicas

- concat – uma lista de listas numa única lista

```
concat [[1,2],[],[3,4,5]]↝ [1,2,3,4,5]
```

- length – retorna o tamanho da lista

```
length [2,4,5]↝ 3; length "Maria" ↝ 5
```

- head, last: retornam o primeiro/último elemento de uma lista não vazia

```
head [2, 4, 5] ↝ 2; last [2, 4, 5] ↝ 5;
head [] ↝ exception
```

- tail, init: retornam a lista sem o primeiro/último elemento

```
tail [2, 4, 5] ↝ [4,5]; tail [2] ↝ [];
tail [] ↝ exception; init [2,4,5] ↝ [2,4];
init [2] ↝ []; init [] ↝ exception
```

dcomp departamento de computação

Listas – Operadores e Funções Básicas

- replicate – constrói uma lista replicando um item dado

```
replicate 3 'c' ↝ "ccc"
```

- take - constrói uma lista com os *n* primeiros elementos da lista dada

```
take 2 [3,9,4,20] ↝ [3,9]; take 3 "Maria" ↝ "Mar"
```

- drop – retorna uma lista pulando os *n* primeiros elementos

```
drop 2 [2, 4, 5] ↝ [5]; drop 2 "Maria" ↝ "ria"
```

- splitAt – quebra uma lista numa dada posição

```
splitAt 2 [2, 4, 5] ↝ ([2,4],[5]);
splitAt 3 "Maria" ↝ ("Mar,"ia")
```

Listas – Funções Básicas

- reverse – inverte os elementos de uma lista

```
reverse "amor" ↝ "roma"
```

- zip – transforma um par de listas em uma lista de pares

```
Zip [1,2] [3,4,5] ↝ [(1,3),(2,4)]; zip [] [] ↝ []
```

- unzip – transforma uma lista de pares em um par de listas

```
unzip [(1,3),(2,4)]↝([1,2],[3,4]);unzip []↝([],[])
```

- and/or- conjunção/disjunção de uma lista de Booleanos

```
and [True, False, True]↝False; and [] ↝ True
or [True, False, True]↝True; or [] ↝ False
```

- sum – soma os números de uma lista numérica

```
sum [1, 4, 10]↝15; sum []↝0
```

- product – calcula o produto de uma lista numérica

```
product [12, 4]↝48; product []↝1
```

Listas - Funções

- Exercício de Fixação
  - Ex: Dada uma lista de inteiros, elabore uma função que calcula o produto de todos os elementos pares da lista

Listas - Funções

- Exercício de Fixação
  - Ex: Dada uma lista de inteiros, elabore uma função que calcula o produto de todos os elementos pares da lista

```
produtoPares :: [Int]-> Int
produtoPares xs = product [x| x <-xs, ehPar x]
```

Listas - Compreensões

- Exercícios Recomendados
  - Dada uma lista de reais, elabore uma função para calcular a soma dos valores da lista maiores ou iguais a 5.0
  - Dada uma lista de reais, elabore uma função para calcular a média dos valores da lista
  - Dada uma lista de notas de alunos, elabore uma função para determinar a lista das notas acima da média das notas da lista.
  - Elabore uma função que dadas três palavras, devolve uma só string que quando impressa mostra as três palavras em linhas separadas.
  - Dadas duas listas de palavras, sendo uma de adjetivos e outra de nomes, construa uma lista que explora todas as combinações de adjetivos e nomes das duas listas. Ex: adjetivos = [“bonito”, “alegre”], nomes=[“menino”, “garoto”]. A resposta deve ser [“menino bonito”, “menino alegre”, “garoto bonito”, “garoto alegre”]

Listas - Compreensões

- Exercícios Recomendados
  - Considere um banco de dados de biblioteca com os seguintes tipos

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

    Escreva as seguintes funções:
    - Dada uma pessoa, encontre os livros que ela emprestou;
    - Dado um livro, encontre quem emprestou este livro, assumindo que o livro pode ter mais de um exemplar;
    - Dado um livro, desejamos saber se o mesmo se encontra emprestado ou não;
    - Dada uma pessoa, desejamos saber a quantidade de livros que ela tomou emprestado;
    - Dado um par (Pessoa, Livro), queremos adicioná-lo à lista de emprestados, sinalizando o empréstimo realizado;
    - Dado um par (Pessoa, Livro), queremos removê-lo da lista de emprestados, sinalizando a sua devolução.