Haskell :: Listas

George Darmiton da Cunha Cavalcanti
(gdcc@cin.ufpe.br)

Monitores:
Bruno Barros (blbs @ cin.ufpe.br)
Caio Nascimento (ccno @ cin.ufpe.br)

Universidade Federal
de Pernambuco

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]`
- `[]` é uma lista de qualquer tipo.

Listas versus 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 :: Notação


```
[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
```

Exemplo

- Obter os n primeiros elementos de uma lista

```
primeirosn 0 _          = []
primeirosn _ []         = []
primeirosn n (a:x)      = a : primeirosn (n-1) x
```

Exemplo

Verifica se um objeto ∈ a lista

```
pertence p []                    = False
pertence p (a:x)  | p == a       = True
                  | otherwise    = pertence p x
```

Exemplo

Inserir objeto na lista sem repetição

```
insere c []                    = [c]
insere c (a:x)     | c == a    = a:x
                   | otherwise = a : insere c x
```

Outras 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]
```

Outras funções sobre listas

- insertion sort

```
iSort :: [Int] -> [Int]
ins :: Int -> [Int] -> [Int]
```

- sempre existem várias opções de como definir uma função

Outras funções sobre listas

- comprimento

```
length :: [t] -> Int
length [] = 0
length (a:as) = 1 + length as
```

- Esta função é polimórfica.

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 [] [] = []
```

Centro de Informática
U·F·P·E

EXEMPLO

Um Sistema para Bibliotecas usando Listas

Exemplo: Biblioteca


```
type Person = String
type Book = String
type Database = [(Person, Book)]
```

Exemplo de um banco de dados


```
exampleBase =
    [("Alice","Postman Pat"),
     ("Anna","All Alone"),
     ("Alice","Spot"),
     ("Rory","Postman Pat")]
```

Funções sobre o banco de dados :: consultas


```
books :: Database -> Person -> [Book]

borrowers :: Database -> Book ->[Person]

borrowed :: Database -> Book -> Bool

numBorrowed :: Database -> Person -> Int
```

Funções sobre o banco de dados :: atualizações


```
makeLoan ::
  Database -> Person -> Book -> Database

returnLoan ::
  Database -> Person -> Book -> Database
```

Compreensão de lista
(*List Comprehension*)

- Uma lista pode ser especificada pela definição de seus elementos:

```
par x = mod x 2 == 0

[x*x | x <- [9..39], par x]

[100,144,196,256,324,400,484,576,676,784,900,1024,1
156,1296,1444]
```

Exercício

- Redefina as seguintes funções utilizando compreensão de listas:

```
member :: [Int] -> Int -> Bool
books :: Database -> Person -> [Book]
borrowers :: Database -> Book ->[Person]
borrowed :: Database -> Book -> Bool
numBorrowed :: Database -> Person -> Int
returnLoan :: Database -> Person -> Book -> Database
```

Centro de Informática
U·F·P·E

Generalizações

Funções de alta ordem

- Funções como argumentos ou como resultado de outras funções

- Permite
  - definições polimórficas
    - funções aplicadas sobre uma coleção de tipos
    - padrões de recursão usados por várias funções

Exemplos


```
double :: [Int] -> [Int]
double [] = []
double (a:x) = (2*a) : double x

sqrList :: [Int] -> [Int]
sqrList [] = []
sqrList (a:x) = (a*a) : sqrList x
```

Funções de mapeamento (*mapping*)

Avaliando **double**


```
double [1,2] =
double 1:[2] = (2*1) : double [2] =
(2*1) : ((2*2) : double []) =
(2*1) : ((2*2) : []) =
2:(4:[]) = [2,4]
```

A função de mapeamento

- Recebe como argumentos
  - a transformação a ser aplicada a cada elemento da lista
    - uma função
  - a lista de entrada

mapear


```
mapear :: (t -> u) -> [t] -> [u]
mapear f []     = []
mapear f (a:as) = f a : mapear f as


sqrtList xs = mapear sqrt xs
```

Por que funções de alta ordem

- Facilita entendimento das funções
- Facilita modificações (mudança na função de transformação)
- Aumenta reuso de definições/código

Centro de Informática
U·F·P·E

Tipos Algébricos

Introdução

- Previamente, fora vistas maneira de modelar entidades através de tipos básicos
  - Int, Float, Bool e Char

- E através de:
  - Tuplas $(t_1, t_2, ..., t_n)$
  - Listas $[t_1]$
  - Funções $(t_1 \rightarrow t_2)$

- Sabendo que $t_1, t_2, ..., t_n$ são tipos

Introdução

- Entretanto, existem outros tipos que são difíceis de modelar usando os construtores vistos até então, exemplos:
  - representar meses: Janeiro, ..., Dezembro
  - representar um tipo cujos elementos podem ser um inteiro ou uma string
  - representar o tipo árvore

Tipos algébricos: Enumerados

- Criar novos tipos de dados, e novos construtores de tipos:

```
data Bool = True | False

data Estacao = Inverno | Verao |
               Outono | Primavera
data Temp = Frio | Quente

data Cor = Verde | Azul | Amarelo | Preto |
Branco
```

Tipos algébricos: Enumerados

- Funções podem ser definidas usando casamento de padrões sobre tipos algébricos:

```
clima :: Estacao -> Temp
clima Inverno = Frio
clima _       = Quente
```

Tipos algébricos:: Produto

- Ao invés de usar uma tupla, pode-se definir um tipo com um número de componentes, chamado de **tipo produto**

- Exemplo

```
data People = Person Name Age
```

  – Sabendo que *Nome* é um sinônimo de *String* e *Age* de *Int*

```
type Name = String
type Age = Int

Person "José" 22
Person "Maria" 23
```

Tipos algébricos:: Produto

- Um elemento do tipo *People* tem a forma geral *Person n a*

```
showPerson :: People -> String
showPerson (Person n a) = n ++ " -- " ++ show a

showPerson (Person "Maria"  77)
= "Maria -- 77"
```

Por que não usar tuplas?


```
Person :: Name -> Age -> People

type People = (Name, Age)
```

- Com tipos algébricos
  - cada objeto do tipo tem um rótulo (*label*) explícito
  - não se pode confundir um par arbitrário consistindo de uma string e de um número com um *Person*
  - o tipo aparecerá nas mensagens de erro

ADT versus tipos sinônimos: quando usar?


```
data Age = Years Int
type Age = Int

f1 :: Int -> Int
f2 :: Age -> Int
```

Alternativas

- Construtores com argumentos

```
data Shape = Circle Float
           | Rectangle Float Float

Circle 4.9 :: Shape
Rectangle 4.2 2.0 :: Shape

isRound :: Shape -> Bool
isRound (Circle _)      = True
isRound (Rectangle _ _) = False
```

Alternativas


```
area :: Shape -> Int
area (Circle r) = pi*r*r
area (Rectangle h w) = h * w
```

Declaração de tipos algébricos: Forma geral

- **data** *NomeDoTipo*
  = $Construtor_1\ t_{11}\ \ldots\ t_1k_1$
  | $Construtor_2\ t_{21}\ \ldots\ t_2k_2$
  ...
  | $Construtor_n\ t_{n1}\ \ldots\ t_nk_n$

- O tipo pode ser recursivo

- A definição pode ser polimórfica, adicionando argumentos ao *NomeDoTipo*

Tipos algébricos recursivos

- Tipos são geralmente descritos em termos deles próprios.
- Por exemplo:
  - uma expressão pode ser um literal (como 686) ou uma combinação usando operadores aritméticos

```
data Expr = Lit Int
          | Add Expr Expr
          | Sub Expr Expr
```

Tipos algébricos recursivos


```
2                   Lit 2
2+3                 Add (Lit 2) (Lit 3)
(3-1)+3             Add (Sub (Lit 3) (Lit 1)) (Lit 3)
```

```
eval :: Expr -> Int
eval (Lit n)      = n
eval (Add e1 e2) = (eval e1) + (eval e2)
eval (Sub e1 e2) = (eval e1) - (eval e2)
```

Tipos algébricos polimórficos

- Tipos de dados polimórficos:

```
data Pairs t = Pair t t
Pair 6 8 :: Pairs Int
Pair True True :: Pairs Bool
Pair [] [1,3] :: Pair [Int]
```

- Listas

```
data List t = Nil
            | Cons t (List t)
```

- Árvores binárias

```
data Tree t = NilT
            | Node t (Tree t) (Tree t)
```

Derivando instâncias de classes


```
data List t = Nil
            | Cons t (List t)
            deriving (Eq,Ord,Show)

data Tree t = NilT
            | Node t (Tree t) (Tree t)
            deriving (Eq,Ord,Show)
```

Exercícios

- Defina as seguintes funções:

```
showExpr :: Expr -> String
toList :: List t -> [t]
fromList :: [t] -> List t
depth :: Tree t -> Int
```