Haskell: Listas

UFRN, 2018

Listas

- Listas em Haskell representam coleções de objetos de um determinado tipo.

- Para um dado tipo t, existe o tipo "lista de elementos do tipo t", designado por [t].

- String é um sinônimo para [Char].

Listas

- Por exemplo:

  ```
  [1,2,3,3,0,1] :: [Int]
  [False,False,True] :: [Bool]
  ```

  são uma lista de Int e uma lista de Bool.

- As duas listas abaixo representam o mesmo valor:

  ```
  ['a','b','c'] :: [Char]
  "abc" :: [Char]
  ```

Listas

- Pode-se construir listas de quaisquer tipos:

  ```
  [sales,totalSales] :: [Int->Int]
  [[1,2],[3,4,4],[]] :: [[Int]]
  ```

- A lista vazia ([]) é uma lista de elementos de qualquer tipo.

Listas

- A ordem e número de ocorrências de elementos são importantes:

  $[1,2] \neq [2,1]$

  $[1,1,2] \neq [1,2]$

- Alternativas para listas de números:

```
[3 .. 6]            ==  [3,4,5,6]
[3.1 .. 6.0]        ==  [3.1,4.1,5.1,6.1]
[10,8 .. 3]         ==  [10,8,6,4]
[0.1,0.3 .. 1.0]    ==  [0.1,0.3,0.5,0.7,0.9]
```

Funções sobre Listas

- Definir uma função

  sumList :: [Int] -> Int

  que calcula a soma dos elementos de uma lista.
- A definição deve cobrir listas de tamanho 0, 1, 2...
- Como construir uma definição que cobre todos os casos???

Funções sobre Listas

- Uma lista qualquer é:
  - a lista vazia [], ou
  - uma lista não vazia, que tem um primeiro elemento (cabeça) e o resto (cauda).
- Uma lista com cabeça a e cauda x é escrita no formato (a:x).
- Atenção: em (a:x)
  - a é um elemento da lista
  - x é uma outra lista

Funções sobre Listas

- Voltando à definição de sumList:
  - a soma dos elementos de uma lista vazia é 0.
  - a soma de uma lista não vazia (a:x) é dada pela soma do elemento a com a soma dos elementos da lista x.
- Em Haskell:

```
1 sumList :: [Int] -> Int
2 sumList []     = 0
3 sumList (a:x)  = a + sumList x
```

Funções sobre Listas

```
1 sumList :: [Int] -> Int
2 sumList []     = 0
3 sumList (a:x)  = a + sumList x
```

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

= 1 + 3 + sumList [3,2]

= 1 + 3 + 3 + sumList [2]

= 1 + 3 + 3 + 2 + sumList []

= 1 + 3 + 3 + 2 + 0

= 9

Construtor de Listas

- O operador : é geralmente chamado *cons*.
- Esse operador é um construtor de listas, dados um elemento a e uma lista x. O tipo de a deve ser o mesmo dos elementos da lista x.

```
[1] = 1:[]
[1,2,3] = 1:2:3:[] = 1:[2,3]
[1,2]:[] = [[1,2]]
[]:[1,2] = ???
```

ERRO DE TIPO!!!

Construtor de Listas

- Listas podem ser de qualquer tipo, assim:

  (:) :: Int -> [Int] -> [Int]

  (:) :: Bool -> [Bool] -> [Bool]

  ...
- Restrição: todos os elementos de uma lista devem ser do mesmo tipo!
- O tipo de : pode ser expresso como:

  (:) :: t -> [t] -> [t]

  onde t é uma variável de tipo.

Funções sobre Listas

- Definir uma função

  double :: [Int] -> [Int]

  que, dada uma lista de comprimento c, retorna uma nova lista de mesmo comprimento, onde cada elemento é o dobro do seu correspondente.

```
Main> double [2,3,5,3]
[4,6,10,6]
```

Funções sobre Listas

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

double [5,3]

= (5*2) : double [3]

= 10 : ((2*3) : double [])

= 10 : (6 : [])

= [10,6]

Funções/Operadores Padrões

- A função

  length :: [t] -> Int

  retorna o comprimento de uma lista.
- Pode ser definida como:

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

Funções/Operadores Padrões

- O operador ++ foi usado na concatenação de Strings, mas é definido para qualquer tipo de listas.
- Exemplo:

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

- Pode ser definido como:

```
1 [] ++ y = y
2 (a:x) ++ y = a : (x ++ y)
```

Funções/Operadores Padrões

```
1 [] ++ y = y
2 (a:x) ++ y = a : (x ++ y)
```

[1,2] ++ [3,4]

= 1 : ([2] ++ [3,4])

= 1 : (2 : ([] ++ [3,4]))

= 1 : (2 : [3,4])

= [1,2,3,4]

Funções/Operadores Padrões

- Os operadores ++ e : são diferentes!
  - O operador : recebe um elemento a e uma lista x, retornando uma nova lista onde a é o primeiro elemento.
  - O operador ++ recebe duas listas e retorna sua concatenação!
- Para memorizar:

  item : lista = lista

  lista ++ lista = lista

Exemplo - Ordenação

- Definir uma função

  iSort :: [Int] -> [Int]

  que retorna uma versão ordenada da lista fornecida como entrada.

```
Main> iSort [8,5,9,1]
[1,5,8,9]
```

Esboço da solução

- Uma solução para

  iSort [8,5,9,1]

  consiste em extrair a cauda [5,9,1] e ordená-la, obtendo [1,5,9]
- Então inserir a cabeça 8 na posição correta:

  [1,5,8,9]

  Observe que se trata de uma inserção em uma lista já ordenada!

Especificação top-down

- Uma definição para iSort :

```
1 iSort :: [Int] -> [Int]
2 iSort [] = []
3 iSort (a:x) = ins a (iSort x)
```

- Esse é um exemplo típico de definição *top-down*. A função `iSort` foi definida supondo é possível definir corretamente `ins`.
- Resolver os subproblemas separadamente torna mais fácil a construção de uma solução.

Continuação da solução

- Falta definir

  ins :: Int -> [Int] -> [Int]

- Observe que

  ins 1 [5,8,9] = 1 : [5,8,9]= [1,5,8,9], pois 1 < 5.

- Mas

  ins 8 [3,6,9] = 3 : (ins 8 [6,9])

  = ... = 3 : [6,8,9] = [3,6,8,9], pois 8 > 3

Especificação final

- Isso sugere seguinte definição para ins:

```
1 ins :: Int -> [Int] -> [Int]
2 ins a [] = [a]
3 ins a (b:y)
4   | a <= b      = a : (b:y)
5   | otherwise   = b : ins a y
```

Exemplo de execução

iSort [4,8,1]
= ins 4 (iSort [8,1])
= ins 4 (ins 8 (iSort [1]))
= ins 4 (ins 8 (ins 1 (iSort [])))
= ins 4 (ins 8 (ins 1 []))
= ins 4 (ins 8 [1])
= ins 4 (1 : (ins 8 []))
= ins 4 [1,8]
= 1 : (ins 4 [8])

```
= 1 : (4 : [8])
= 1 : [4,8]
= [1,4,8]
```

Exemplo - função membro

- Definir uma função

  membro :: Int -> [Int] -> Bool

  que verifica se um inteiro está presente em uma lista (não necessariamente ordenada).

```
Main> membro 3 [8,5,9,1]
False
Main> membro 5 [8,5,9,1]
True
```

Esboço da solução

- Se a lista for vazia, o elemento não está presente:

  membro 5 [] = False
- Se for uma lista não vazia e o elemento for igual ao primeiro da lista, a resposta é afirmativa:

  membro 8 [8,5,9,1] = True
- Se o elemento não for igual ao primeiro da lista, deve-se verificar no resto da lista:

  membro 8 [5,9,1,...] = membro 8 [9,1,...]

Especificação

- Uma definição para membro :

```
1 membro b [] = False
2 membro b (a:x)
3    | b == a          = True
4    | otherwise       = membro b x
```

- Definição simplificada:

```
1 membro b [] = False
2 membro b (a:x) = (b==a) || membro b x
```

Exemplo - filtro

- Definir uma função

  digits :: String -> String

  que seleciona ("filtra") os dígitos de uma String.

```
Main> digits "10 and 20 are numbers"
"1020"
```

Esboço da solução

- Se a lista (String) for vazia, a resposta é também uma lista vazia:

  digits [] = []

- Se for uma lista não vazia e o primeiro elemento for um dígito, este será o primeiro elemento da resposta:

  digits ['1','a',...]= '1' : digits ['a',...]

- Se o primeiro elemento não for um dígito, não fará parte da resposta:

  digits ['a','1',...] = digits ['1',...]

Especificação

- Uma definição para digits :

```
1 digits [] = []
2 digits (a:x)
3   | isDigit a = a:digits x
4   | otherwise = digits x
```