Programação Funcional
Aula 4 — Listas

Pedro Vasconcelos
DCC/FCUP

2021

Listas

Listas são coleções de elementos:

- em que a ordem é significativa
- possivelmente com elementos repetidos

Listas em Haskell

Uma lista em Haskell

ou é vazia `[]`;

ou é `x:xs` (`x` seguido da lista `xs`).

Notação em extensão

Usamos parêntesis rectos e elementos separados por vírgulas.

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

Sequências aritméticas

Expressões da forma $[a..b]$ ou $[a,b..c]$ (a, b e c são números).

```
> [1..10]
[1,2,3,4,5,6,7,8,9,10]

> [1,3..10]
[1,3,5,7,9]

> [10,9..1]
[10,9,8,7,6,5,4,3,2,1]
```

Sequências aritméticas (cont.)

Também podemos construir listas infinitas usando expressões `[a..]` ou `[a,b..]`.

```
> take 10 [1,3..]
[1,3,5,7,9,11,13,15,17,19]
```

A listagem de uma lista infinita no GHCi não termina (interrompemos usando *Ctrl-C*):

```
> [1,3..]
[1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,
39,41,43,45,47,49,51,53,55,57,59,61,63,65,67,69,71,73,
Interrupted
```

Notação em compreensão

Em matemática é usual definir conjunto apartir de outro usando notação em compreensão.

Exemplo:

$$\{x^2 \ : \ x \in \{1,2,3,4,5\}\}$$

define o conjunto

$$\{1,4,9,16,25\}$$

Notação em compreensão (cont.)

Podemos definir uma lista a partir de outra usando uma notação análoga.

Exemplo:

```
> [x^2 | x<-[1,2,3,4,5]]
[1, 4, 9, 16, 25]
```

Geradores

Um termo "*padrão*`<-`*lista*" chama-se um gerador:

- determina quais os valores das variáveis no padrão
- e a ordem pela qual os valores são gerados

Podemos também usar múltiplos geradores.

```
> [(x,y) | x<-[1,2,3], y<-[4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

Gera todos os pares (x, y) tal que x toma valores $[1, 2, 3]$ e y toma valores $[4, 5]$.

Ordem entre geradores

x primeiro, *y* depois

```
> [(x,y) | x<-[1,2,3], y<-[4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

y primeiro, *x* depois

```
> [(x,y) | y<-[4,5], x<-[1,2,3]]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

Analogia: ciclos 'for' imbricados

```
for(x=1; x<=3; x++)
  for(y=4; y<=5; y++)
     print(x,y);
```

```
for(y=4; y<=5; y++)
  for(x=1; x<=3; x++)
     print(x,y);
```

Dependências entre geradores

Os valores usados em geradores podem depender dos valores *anteriores* mas não dos *posteriores*.

```
> [(x,y) | x<-[1..3], y<-[x..3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]

> [(x,y) | y<-[x..3], x<-[1..3]]
error: Variable not in scope: x
```

Dependências entre geradores (cont.)

Um exemplo: a função *concat* (do prelúdio-padrão) concatena uma lista de listas, e.g.:

```
> concat [[1,2,3],[4,5],[6,7]]
[1,2,3,4,5,6,7]
```

Podemos definir usando uma lista em compreensão:

```
concat :: [[a]] -> [a]
concat listas = [valor | lista<-listas, valor<-lista]
```

Guardas

As definições em compreensão podem incluir condições sobre os valores (designadas *guardas*).

Exemplo: os inteiros *x* tal que *x* está entre 1 e 10 e *x* é par.

```
> [x | x<-[1..10], x`mod`2==0]
[2,4,6,8,10]
```

Exemplo maior: testar primos

Vamos começar por definir uma função auxiliar para listar todos os divisores de um inteiro positivo:

```
divisores :: Int -> [Int]
divisores n = [x | x<-[1..n], n`mod`x==0]
```

Exemplo:

```
> divisores 15
[1,3,5,15]
> divisores 19
[1,19]
```

Exemplo maior: testar primos (cont.)

Vamos agora definir uma função para testar primos: *n* é primo sse os seus divisores são *exatamente* 1 e *n*.

```
testarPrimo :: Int -> Bool
testarPrimo n = divisores n == [1,n]
```

```
> testarPrimo 15
False
> testarPrimo 19
True
```

(Um exercício da folha 3 propõe uma alternativa mais eficiente.)

Exemplo maior: testar primos (cont.)

Podemos usar a função `testePrimo` como guarda para listar todos os primos até a um limite dado.

```
primos :: Int -> [Int]
primos n = [x | x<-[2..n], testaPrimo x]
```

Exemplo:

```
> primos 50
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]
```

A função `zip`

A função `zip` do prelúdio-padrão combina duas listas na lista dos pares de elementos correspondentes.

```
zip :: [a] -> [b] -> [(a,b)]
```

Exemplo:

```
> zip ['a','b','c'] [1,2,3,4]
[('a',1), ('b',2), ('c',3)]
```

Se as listas tiverem comprimentos diferentes o resultado tem o comprimento da *menor*.

Usando a função `zip`

Podemos usar `zip` para combinar elementos de uma lista com os seus índices.

Exemplo: procurar índices de ocorrências de um valor numa lista.

```
indices :: Eq a => a -> [a] -> [Int]
indices x ys = [i | (y,i)<-zip ys [0..n], x==y]
     where n = length ys - 1
```

```
> indices 'a' ['b','a','n','a','n','a']
[1,3,5]
```

Usando a função `zip` (cont.)

Também podemos usar `zip` e `tail` para listar pares de elementos consecutivos de uma lista.

```
pares :: [a] -> [(a,a)]
pares xs = zip xs (tail xs)
```

```
xs          =  [x1, x2, ..., x_n-1, x_n]
tail xs     =  [x2, x3, ..., x_n]
zip xs (tail xs) = [(x1,x2), (x2,x3), ..., (x_n-1, x_n)]
```

Usando a função `zip` (cont.)

Exemplos

```
> pares [1,2,3,4]
[(1,2),(2,3),(3,4)]

> pares ['a','b','b','a']
[('a','b'),('b','b'),('b','a')]

> pares [1,2]
[(1,2)]

> pares [1]
[]
```

Usando a função `zip` (cont.)

Contar o número de elementos consecutivos iguais:

```
paresIguais :: Eq a => [a] -> Int
paresIguais xs
   = length [(x,x') | (x,x') <- zip xs (tail xs), x==x']
```

Exemplos

```
> paresIguais [1, 1, 2, 2, 3]
2

> paresIguais ['a','b','b','a']
1
```

Cadeias de carateres

O tipo `String` é pré-definido no prelúdio-padrão como um sinónimo de *lista de carateres*.

```
type String = [Char]          -- definido no prelúdio-padrão
```

Por exemplo:

```
"abba"
```

é equivalente a

```
['a','b','b','a']
```

Cadeias de carateres (cont.)

Como as cadeias são listas de carateres, podemos usar as funções de listas com cadeias de carateres.

Exemplos:

```
> length "abcde"
5

> take 3 "abcde"
"abc"

> zip "abc" [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

Cadeias em compreensão

Como as cadeias são listas, também podemos usar notação em compreensão com cadeias de carateres.

Exemplo: contar carateres entre '`A`' e '`Z`' inclusivé.

```
contarLetras :: String -> Int
contarLetras txt = length [c | c<-txt, c>='A' && c<='Z']
```

Processamento de listas e de carateres

Muitas funções especializadas estão definidas em módulos e não diretamente no prelúdio.

Devemos importar um módulo para puder usar as funções nele definidas.

Processamento de listas e de carateres (cont.)

Exemplo: o módulo `Data.Char` contém várias funções sobre caracteres.

```
isUpper :: Char -> Bool
      -- testar se é letra maiúscula
isLower :: Char -> Bool
      -- testar se é letra minúscula
isLetter :: Char -> Bool
      -- testar se é letra (qualquer)
toUpper :: Char -> Char
      -- converter para maiúscula (ou for letra)
toLower :: Char -> Char
       -- converter para minúscula (se for letra)
```

Processamento de listas e de carateres (cont.)

```
import Data.Char

countLetters :: String -> Int
countLetters xs = length [x | x<-xs, isLetter x]

stringToUpper :: String -> String
stringToUpper xs = [toUpper x | x<-xs]


> countLetters "Abba123"
4

> stringToUpper "Abba123"
"ABBA123"
```

Mais informação

Usamos `:browse` no GHCi para listar os tipos de todas as funções num módulo.

```
Prelude> import Data.Char
Prelude Data.Char> :browse
digitToInt :: Char -> Int
isLetter :: Char -> Bool
isMark :: Char -> Bool
⋮
```