Programação Funcional
4ª Aula — Listas

Sandra Alves
DCC/FCUP

2018/19

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

Elementos entre parêntesis rectos separados por vírgulas.

```
[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]
```

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]
```

Se tentarmos mostrar uma lista infinita o processo não termina; temos de interromper o interpretador (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

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

Em Haskell podemos definir uma lista apartir de outra usando uma notação semelhante.

Exemplo:

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

Geradores

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

- determina os valores das variáveis no padrão;
- a ordem dos valores gerados.

Podemos usar múltiplos geradores:

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

Ordem entre geradores

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

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

- As variáveis dos geradores posteriores mudam primeiro
- Analogia: ciclos 'for' embricados

```
for(x=1; x<=3; x++)              for(y=4; y<=5; y++)
  for(y=4; y<=5; y++)      vs.     for(x=1; x<=3; x++)
     ...                              ...
```

Dependências entre geradores

Os 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]]
ERRO: x não está definido
```

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 xss = [x | xs<-xss, x<-xs]
```

Guardas

As definições em compreensão podem incluir condições (designadas *guardas*) para filtrar os resultados.

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: testar primos

Usando guardas, é fácil definir uma função 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]
```

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

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

```
> primo 15
False

> primo 19
True
```

NB: esta solução é ineficiente...

Vamos usar o teste de primalidade como *guarda* para listar todos os primos até a um limite dado.

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

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

A função *zip*

A função *zip* definida no 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*

Combinando *zip* e *tail*, vamos definir uma função para obter *pares consecutivos* de elementos de uma lista.

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

Justificação:

$$\begin{aligned} \text{xs} &= x_1 : x_2 : \ldots : x_n : \ldots \\ \text{tail xs} &= x_2 : x_3 : \ldots : x_{n+1} : \ldots \\ \therefore \quad \text{zip xs (tail xs)} &= (x_1, x_2) : (x_2, x_3) : \ldots : (x_n, x_{n+1}) : \ldots \end{aligned}$$

Exemplos:

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

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

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

> pares [1]
[]
```

Usando as funções

```
pares :: [a] -> [(a,a)]
and :: [Bool] -> Bool -- do prelúdio-padrão
```

onde

$$\texttt{and}\ [b_1, b_2, \ldots, b_n] = b_1 \texttt{\&\&} b_2 \texttt{\&\&} \ldots \texttt{\&\&} b_n\ ,$$

vamos definir uma função que verifica se uma lista está por ordem crescente.

```
crescente :: Ord a => [a] -> Bool
crescente xs = and [x<=x' | (x,x')<-pares xs]
```

Alguns exemplos:

```
> crescente [2,3]
True

> crescente [2,3,4,7,8]
True

> crescente [2,8,3,7,4]
False
```

Qual é o resultado com listas com um só elemento?
E com a lista vazia?
Podemos usar *zip* para combinar *elementos* com *índices* numa lista.

Exemplo: procurar um valor numa lista e obter todos os seus índices.

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

Exemplo:

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

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:

```
"abc"
```

é equivalente a

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

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 letras minúsculas.

```
minusculas :: String -> Int
minusculas txt = length [c | c<-txt, c>='a' && c<='z']
```

Processamento de listas e de carateres

Muitas funções especializadas estão pré-definidas em *módulos*.

Para utilizar um módulo devemos *importar* as suas definições.

Exemplo: o módulo `Data.Char` define operações sobre caracteres.

```
import Data.Char

minusculas :: String -> Int
minusculas cs = length [c | c<-cs, isLower c]

-- isLower :: Char -> Bool
-- testar se um carater é uma letra minúscula
```

Um outro exemplo: converter cadeias de carateres em maiúsculas.

```
import Data.Char

stringUpper :: String -> String
stringUpper cs = [toUpper c | c<-cs]

-- toUpper :: Char -> Char
-- converter letras em maiúsculas
```

Mais informação

Podemos usar o GHCi para listar todos os nomes definidos num módulo:

```
Prelude> import Data.Char
Prelude Data.Char> :browse
```