

Listas

Listas

Daniel Ventura
INF/UFG

2024/02

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 colchetes separados por vírgulas.

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

Sequências aritméticas I

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 II

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 II

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

Exemplo:

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

Notação em compreensão I

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

Exemplo:

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

define o conjunto

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

Geradores I

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]]  
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

-- x primeiro

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

-- y primeiro

- As variáveis dos geradores posteriores mudam primeiro
- Analogia: laços de 'for' aninhados

```
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 I

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
```

Dependências entre geradores II

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 defini-la usando uma lista em compreensão:

```
concat :: [[a]] -> [a]  
concat xss = [x | xs<-xss, x<-xs]
```

Guardas

As definições em compreensão podem incluir condições (chamadas *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 I

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]
```

Exemplo: testar primos II

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

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

```
> primo 15
False
```

```
> primo 19
True
```

OBS: esta solução é **ineficiente**...

Exemplo: testar primos III

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* I

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)
```

De fato:

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

Usando a função *zip* II

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 a função *zip* III

Usando as funções

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

onde

$$\text{and } [b_1, b_2, \dots, b_n] = b_1 \&\& b_2 \&\& \dots \&\& b_n,$$

definimos 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]
```

Usando a função *zip* IV

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?

Usando a função *zip* V

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 caracteres I

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

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

Por exemplo:

```
"abc"
```

é equivalente a

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

Usando a função *zip* VI

Sem recorrer à função *length*, definimos a função para lista potencialmente infinitas.

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

Exemplo:

```
> take 10 (indices 'a' ['a','a'..])
[0,1,2,3,4,5,6,7,8,9]
```

Cadeias de caracteres II

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

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 caracteres.

Exemplo: contar letras minúsculas.

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

Processamento de listas e de caracteres I

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

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

Processamento de listas e de caracteres II

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 caracter é uma letra minúscula
```

Processamento de listas e de caracteres III

Um outro exemplo: converter cadeias de caracteres 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
```