Listas por compreensão

A expessão `x <- [1,2,3,4,5]` é chamada de gerador da lista.

A expessão `10 <= x^2` é uma guarda que restringe os valores produzidos pelo gerador que a precede.

```
> [ x^2 | x <- [1,2,3,4,5], 10 <= x^2 ]
[16,25]
```

As listas por compreensão podem ter vários geradores e várias guardas.

```
> [ (x,y) | x <- [1,2,3], y <- [4,6] ]
[(1,4),(1,6),(2,4),(2,6),(3,4),(3,6)]
```

Mudar a ordem dos geradores muda a ordem dos elementos na lista final.

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

Um gerador pode depender de variáveis introduzidas por geradores anteriores.

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


Listas por compreensão

Pode-se usar a notação `..` para representar uma enumeração com o passo indicado pelos dois primeiros elementos. Caso não se indique o segundo elemento, o passo é um.

```
> [1..5]
[1,2,3,4,5]
```

```
> [1,10..100]
[1,10,19,28,37,46,55,64,73,82,91,100]
```

```
> [20,15..(-7)]
[20,15,10,5,0,-5]
```

```
> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
```


Listas infinitas

É possível também definir listas infinitas.

```
> [1..]
[1,2,3,4,5,6,7,8,9,10,11,…
```

```
> [0,10..]
[0,10,20,30,40,50,60,70,80,90,100,110,120,130,…
```

```
> [ x^3 | x <- [0..], even x ]
[0,8,64,216,512,1000,…
```

```
> take 10 [3,3..]
[3,3,3,3,3,3,3,3,3,3]
```

```
> zip "Haskell" [0..]
[('H',0),('a',1),('s',2),('k',3),('e',4),('l',5),('l',6)]
```


Funções e listas por compreensão

Podem-se definir funções usando listas por compreensão.

Exemplo: A função de ordenação de listas *quick sort.*

```
qsort :: (Ord a) => [a] -> [a]
qsort [] = []
qsort (x:xs) = (qsort [y | y<-xs, y<x]) ++[x]++ (qsort [y | y<-xs, y>=x])
```

Esta versão do *quick sort* faz duas travessias da lista para fazer a sua partição e, por isso, é pior do que a versão anterior com a função auxiliar `parte`.

Funções e listas por compreensão

Exemplo: Usando a função `zip` e listas por compreensão, podemos definir a função que calcula a lista de posições de um dado valor numa lista.

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

O lado esquerdo do gerador da lista é um padrão.

A *lazy evaluation* do Haskell faz com que não seja problemático usar uma lista infinita como argumento da função `zip`.

```
> posicoes 3 [4,5,3,4,5,6,3,5,3,1]
[2,6,8]
```


Funções e listas por compreensão

Exemplo: Calcular os divisores de um número positivo.

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

Testar se um número é primo.

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

```
> primo 5
True
> primo 1
False
```

Lista de números primos até um dado n.

```
primosAte :: Integer -> [Integer]
primosAte n = [ x | x <- [1..n], primo x]
```

Lista infinita de números primos.

```
primos :: [Integer]
primos = [ x | x <- [2..], primo x]
```


Crivo de Eratóstenes

Um algoritmo mais eficiente para encontrar números primos, é o Crivo de Eratóstenes (assim chamado em honra ao matemático grego que o inventou), que permite obter todos os números primos até um determinados valor n. A ideia é a seguinte:

- Começa-se com a lista [2..n].
- Guarda-se o primeiro elemento da lista (pois é um número primo) e removem-se da cauda da lista todos os múltiplos desse primeiro elemento.
- Continua-se a aplicar o passo anterior à restante lista, até que a lista de esgote.

```
crivo []     = []
crivo (x:xs) = x : crivo [ n | n <- xs , n `mod` x /= 0 ]
```

```
primosAte n = crivo [2..n]
```

Lista infinita de números primos.

```
primos = crivo [2..]
```


Factorização em primos

O Teorema Fundamental da Aritmética (enunciado pela primeira vez por Euclides) diz que qualquer número inteiro (maior do que 1) pode ser decomposto num produto de números primos. Esta decomposição é única a menos de uma permutação.

Exemplo: Com o auxílio da lista de números primos, podemos definir uma função que dado um número (maior do que 1), calcula a lista dos seus factores primos.

```
factoriza :: Integer -> [Integer]
factoriza n = aux n primos
  where aux 1 _ = []
        aux n (x:xs)
              | n `mod` x /= 0 = aux n xs
              | otherwise      = x : aux (n `div` x) (x:xs)
```

```
> factoriza 94753
[19,4987]
> factoriza 9475312
[2,2,2,2,7,11,7691]
```

Funções de ordem superior

Em Haskell, as funções são entidades de primeira ordem. Ou seja,

- As funções podem receber outras funções como argumento.

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

Exemplos:

```
dobro :: Int -> Int
dobro x = x + x
```

```
quadruplo :: Int -> Int
quadruplo x = twice dobro x
```

```
quadruplo 5 = twice dobro 5
            = dobro (dobro 5)
            = (dobro 5) + (dobro 5)
            = (5+5) + (5+5)
            = 10 + 10
            = 20
```

```
retira2 :: [a] -> [a]
retira2 l = twice tail l
```

```
retira2 [4,5,7,0,9] = twice tail [4,5,7,0,9]
                    = tail (tail [4,5,7,0,9])
                    = tail [5,7,0,9]
                    = [7,0,9]
```


Funções de ordem superior

- As funções podem devolver outras funções como resultado.

```
mult :: Int -> Int -> Int
mult x y = x * y
```

O tipo é igual a `Int -> (Int -> Int)`, porque `->` é associativo à direita

Exemplos:

```
triplo :: Int -> Int
triplo = mult 3
```

`triplo` tem o mesmo tipo que `mult 3`

```
triplo 5 = mult 3 5
         = 3 * 5
         = 15
```

`mult 3 5 = (mult 3) 5`, porque a aplicação é associativa à esquerda

```
twice (mult 2) 5 = (mult 2) ((mult 2) 5) = mult 2 (mult 2 5)
                 = 2 * (mult 2 5)
                 = 2 * (2 * 5)
                 = 20
```


`map`

Consideremos as seguintes funções:

```
triplos :: [Int] -> [Int]
triplos [] = []
triplos (x:xs) = 3*x : triplos xs
```

Estas funções fazem coisas distintas entre si, mas **a forma como operam é semelhante**: aplicam uma transformação a cada elemento da lista de entrada.

```
maiusculas :: String -> String
maiusculas [] = []
maiusculas (x:xs) = toUpper x : maiusculas xs
```

Dizemos que estas funções têm um **padrão de computação** comum, e apenas diferem na função que é aplicada a cada elemento da lista.

```
somapares :: [(Float,Float)] -> [Float]
somapares [] = []
somapares ((a,b):xs) = a+b : somapares xs
```

A função **`map`** do `Prelude` sintetiza este padrão de computação, abstraindo em relação à função que é aplicada aos elementos da lista.

`map`

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x) : (map f xs)
```

`map` é uma função de ordem superior que recebe a função `f` que é aplicada ao longo da lista.

Exemplos:

```
triplos :: [Int] -> [Int]
triplos l = map (3*) l
```

```
maiusculas :: String -> String
maiusculas xs = map toUpper xs
```

```
somapares :: [(Float,Float)] -> [Float]
somapares l = map aux l
    where aux (a,b) = a+b
```

```
triplos [1,2] = map (3*) [1,2]
              = 3*1 : map (3*) [2]
              = 3*1 : 3*2 : map (3*) []
              = 3*1 : 3*2 : []
              = 3:6:[] = [3,6]
```

Usando listas por compreensão, poderíamos definir a função `map` assim:

```
map f l = [ f x | x <- l ]
```

filter

Consideremos as seguintes funções:

Estas funções fazem coisas distintas entre si, mas **a forma como operam é semelhante**: selecionam da lista de entrada os elementos que verificam uma dada condição.

```
pares :: [Int] -> [Int]
pares [] = []
pares (x:xs) = if even x
               then x : pares xs
               else pares xs
```

Estas funções têm um **padrão de computação** comum, e apenas diferem na condição com que cada elemento da lista é testado.

```
positivos :: [Double] -> [Double]
positivos [] = []
positivos (x:xs)
          | x > 0     = x : positivos xs
          | otherwise = positivos xs
```

A função **`filter`** do `Prelude` sintetiza este padrão de computação, abstraindo em relação à condição com que os elementos da lista são testados.

filter

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
     | p x       = x : filter p xs
     | otherwise = filter p xs
```

`filter` é uma função de ordem superior que recebe a condição `p` (um predicado) com que cada elemento da lista é testado.

Exemplos:

```
pares :: [Int] -> [Int]
pares l = filter even l
```

```
pares [1,2,3,4] = filter even [1,2,3,4]
                = filter even [2,3,4]
                = 2 : filter even [3,4]
                = 2 : filter even [4]
                = 2 : 4 : filter even []
                = 2:4:[] = [2,4]
```

```
positivos :: [Double] -> [Double]
positivos xs = filter (>0) xs
```

Usando listas por compreensão, poderíamos definir a função `filter` assim:

```
filter p l = [  x | x <- l, p x ]
```


Funções anónimas

Em Haskell é possível definir funções sem lhes dar nome, através expressões lambda.

Por exemplo, `\x -> x+x`

É uma **função anónima** que recebe um número `x` e devolve como resultado `x+x`.

```
> (\x -> x+x) 5
10
```

Uma expressão lambda tem a seguinte forma (a notação é inspirada no *λ-calculus*):

\padrão ... padrão -> expressão

Exemplos:

```
> (\x y -> x+y) 3 8
11
```

```
> (\(x1,y1) (x2,y2) -> (x1+x2,y1+y2)) (3,2) (7,9)
(10,11)
```

```
> (\(x:xs) -> xs) [1,2,3]
[2,3]
```

```
> (\(x:xs) y -> y:xs) [1,2,3] 9
[9,2,3]
```


Funções anónimas

As expressões lambda são úteis para evitar declarações de pequenas funções auxiliares.

Exemplo: Em vez de

```
trocapares :: [(a,b)] -> [(b,a)]
trocapares l = map troca l
   where troca (x,y) = (y,x)
```

pode-se escrever

```
trocapares l = map (\(x,y)->(y,x)) l
```

Exemplo:

```
multiplosDe :: Int -> [Int] -> [Int]
multiplosDe n xs = filter (\x -> mod x n == 0) xs
```