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]
                    = tail [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
```

```
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]
```

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

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

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
```