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 com parâmetro de acumulação

- A ideia que está na base destas funções é que elas vão ter um parâmetro extra (o acumulador) onde a resposta vai sendo construída e gravada à medida que a recursão progride.
- O acumulador vai sendo actualizado e passado como parâmetro nas sucessivas chamadas da função.
- Uma vez que o acumulador vai guardando a resposta da função, o seu tipo deve ser igual ao tipo do resultado da função.

Exemplo: A função que inverte uma lista.

A função `inverte` chama uma função auxiliar `inverteAc` com um parâmetro de acumulação e inicializa o acumulador.

O acumulador é inicialmente a lista vazia.

```
inverte :: [a] -> [a]
inverte l = inverteAc l []
  where inverteAc [] ac = ac
        inverteAc (x:xs) ac = inverteAc xs (x:ac)
```

Quando a lista é vazia o acumulador tem a solução completa.

A chamada recursiva é feita actualizando o acumulador.

Funções com parâmetro de acumulação

```
inverte :: [a] -> [a]
inverte l = inverteAc l []
  where inverteAc [] ac = ac
        inverteAc (x:xs) ac = inverteAc xs (x:ac)
```

```
inverte [1,2,3] = inverteAc [1,2,3] []
                = inverteAc [2,3] [1]
                = inverteAc [3] [2,1]
                = inverteAc [] [3,2,1]
                = [3,2,1]
```

A solução está a ser construída no acumulador.

Esta versão é bastante mais eficiente que a função `reverse` anteriormente definida (porque usa o ++ que tem que atravessar a primeira lista).

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

```
reverse [1,2,3] = (reverse [2,3]) ++ [1]
                = ((reverse [3]) ++ [2]) ++ [1]
                = (((reverse []) ++ [3]) ++ [2]) ++ [1]
                = [] ++ [3] ++ [2] ++ [1]
                = …
                = [3,2,1]
```

Funções com parâmetro de acumulação

Podemos sistematizar as seguintes regras para definir funções usando esta técnica:

1. Colocar o acumulador como um parâmetro extra.
2. O acumulador deve ser do mesmo tipo que o do resultado da função.
3. Devolver o acumulador no acaso de paragem da função.
4. Actualizar o acumulador na chamada recursiva da função.
5. A função principal (sem acumulador) chama a função com parâmetro de acumulação, inicializando o acumulador.

Exemplo: O somatório de uma lista de números.

```
somatorio :: Num a => [a] -> a
somatorio l = sumAc l 0
  where sumAc :: Num a => [a] -> a -> a
        sumAc [] n = n
        sumAc (x:xs) n = sumAc xs (x+n)
```

```
somatorio [1,2,3]
        = sumAc [1,2,3] 0
        = sumAc [2,3] (1+0)
        = sumAc [3] (2+1+0)
        = sumAc [] (3+2+1+0)
        = 6
```

Funções com parâmetro de acumulação

Exemplo: O máximo de uma lista não vazia.

```
maximo :: Ord a => [a] -> a
maximo (x:xs) = maxAc xs x
  where maxAc :: Ord a => [a] -> a -> a
        maxAc [] n = n
        maxAc (x:xs) n = if x > n then maxAc xs x
                                  else maxAc xs n
```

```
maximo [2,7,3,9,4] = maxAc [7,3,9,4] 2
                   = maxAc [3,9,4] 7
                   = maxAc [9,4] 7
                   = maxAc [4] 9
                   = maxAc [] 9
                   = 9
```

Funções com parâmetro de acumulação

Exemplo: A função factorial.

```
factorial :: Integer -> Integer
factorial n = factAc n 1
  where factAc :: Integer -> Integer -> Integer
        factAc 0 x = x
        factAc n x | n>0 = factAc (n-1) (n*x)
```

```
factorial 5 = factAc 5 1
            = factAc 4 (5*1)
            = factAc 3 (4*5*1)
            = factAc 2 (3*4*5*1)
            = factAc 1 (2*3*4*5*1)
            = factAc 0 (1*2*3*4*5*1)
            = 120
```

Funções com parâmetro de acumulação

Exemplo: A função `stringToInt :: String -> Int` que converte uma string (representando um número) num valor inteiro.

```
stringToInt "5247" = 5247
```

```
import Data.Char

stringToInt :: String -> Int
stringToInt (x:xs) = aux xs (digitToInt x)
  where aux :: String -> Int -> Int
        aux (h:t) ac = aux t (ac*10 + (digitToInt h))
        aux [] ac = ac
```

```
stringToInt "5247" = aux "247" 5
                   = aux "47" (50+2)
                   = aux "7" (520+4)
                   = aux "" (5240+7)
                   = 5247
```

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
```

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

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

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

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:

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

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

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.

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

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
```

Funções anónimas

As expressões lambda podem ser usadas na definição de funções. Por exemplo:

```
soma x y = x + y

soma1 = \x y -> x + y

soma2 = \x -> (\y -> x + y)

soma3 x = \y -> x + y
```

`soma`, `soma1`, `soma2` e `soma3` são funções equivalentes

Os operadores infixos aplicados apenas a um argumento (a que se dá o nome de secções), são uma forma abreviada de escrever funções anónimas.

Exemplos:

```
(+y)  ≡  \x -> x+y

(x+)  ≡  \y -> x+y

(*3)  ≡  \x -> x*3
```