

## 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 [1]) ++ [3]) ++ [2]) ++ [1]
                = [] ++ [3] ++ [2] ++ [1]
                = ...
                = [3,2,1]
```

98

## 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
```

99

## 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
```

100

## 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
```

101

## Funções com parâmetro de acumulação

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

102

## Listas por compreensão

Na matemática é costume definir conjuntos por compreensão à custa de outros conjuntos.

$\{2x \mid x \in \{10,3,7,2\}\}$

O conjunto  $\{20,6,14,4\}$ .

$\{n \mid n \in \{4, -5, 8, 20, -7, 1\} \wedge 0 \leq n \leq 10\}$

O conjunto  $\{4, 8, 1\}$ .

Em Haskell podem definir-se **listas por compreensão**, de modo semelhante, construindo novas listas à custa de outras listas.

```
[ 2*x | x <- [10,3,7,2] ]
```

A lista  $[20, 6, 14, 4]$ .

```
[ n | n <- [4,-5,8,20,-7,1], 0<=n, n<=10 ]
```

A lista  $[4, 8, 1]$ .

103

## Listas por compreensão

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

A expressão `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)]
```

104

## 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"
```

105

## 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)]
```

106

## 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 *part*.

107

## 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]
```

108

## 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]
```

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]
```

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

109