Programação Funcional

Ficha 4

Listas por compreensão e optimização com *tupling* e acumuladores

1. Para cada uma das expressões seguintes, exprima por enumeração a lista correspondente. Tente ainda, para cada caso, descobrir uma outra forma de obter o mesmo resultado.

   (a) `[x | x <- [1..20], mod x 2 == 0, mod x 3 == 0]`

   (b) `[x | x <- [y | y <- [1..20], mod y 2 == 0], mod x 3 == 0]`

   (c) `[(x,y) | x <- [0..20], y <- [0..20], x+y == 30]`

   (d) `[sum [y | y <- [1..x], odd y] | x <- [1..10]]`

2. Defina cada uma das listas seguintes por compreensão.

   (a) `[1,2,4,8,16,32,64,128,256,512,1024]`

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

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

   (d) `[[1],[1,1],[1,1,1],[1,1,1,1],[1,1,1,1,1]]`

   (e) `[1,2,6,24,120,720]`

3. Defina a função `digitAlpha :: String -> (String,String)`, que dada uma string, devolve um par de strings: uma apenas com as letras presentes nessa string, e a outra apenas com os números presentes na string. Implemente a função de modo a fazer uma única travessia da string. Relembre que as funções `isDigit,isAlpha :: Char -> Bool` estão já definidas no módulo `Data.Char`.

4. Defina a função `nzp :: [Int] -> (Int,Int,Int)` que, dada uma lista de inteiros, conta o número de valores nagativos, o número de zeros e o número de valores positivos, devolvendo um triplo com essa informação. Certifique-se que a função que definiu percorre a lista apenas uma vez.

5. Defina a função `divMod :: Integral a => a -> a -> (a, a)` que calcula simultaneamente a divisão e o resto da divisão inteira por subtracções sucessivas.

6. Utilizando uma função auxiliar com um acumulador, optimize seguinte definição recursiva que determina qual o número que corresponde a uma lista de digitos.

   ```
   fromDigits :: [Int] -> Int
   fromDigits []    = 0
   fromDigits (h:t) = h*10^(length t) + fromDigits t
   ```

   Note que

   $$\begin{aligned} \texttt{fromDigits [1,2,3,4]} &= 1\times10^3+2\times10^2+3\times10^1+4\times10^0 \\ &= 4+10\times(3+10\times(2+10\times(1+10\times0))) \end{aligned}$$

7. Utilizando uma função auxiliar com acumuladores, optimize seguinte definição que determina a soma do segmento inicial de uma lista com soma máxima.

```
maxSumInit :: (Num a, Ord a) => [a] -> a
maxSumInit l = maximum [sum m | m <- inits l]
```

8. Optimize a seguinte definição recursiva da função que calcula o `n`-ésimo número da sequência de Fibonacci, usando uma função auxiliar com 2 acumuladores que representam, respectivamente, o `n`-ésimo e o `n+1`-ésimo números dessa sequência.

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```