

# Ficha 6

## Programação Funcional

2015/16

1. Defina uma função `toDigits :: Int -> [Int]` que, dado um número positivo (na base 10), calcula a lista dos seus dígitos (por ordem inversa). Por exemplo, `toDigits 1234` deve corresponder a `[4,3,2,1]`. Note que

$$1234 = 1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$

```
toDigits :: Int -> [Int]  
toDigits 0 = []  
toDigits x = (mod x 10) : (toDigits (div x 10))
```

2. Pretende-se agora que defina a função inversa da anterior `fromDigits :: [Int] -> Int`. Por exemplo, `fromDigits [4,3,2,1]` deve corresponder a 1234.

- (a) Defina a função com auxílio da função `zipWith`.

```
fromDigits' :: [Int] -> Int  
fromDigits' ints = sum (zipWith (\x y -> x * (10^y)) ints [0,1..])
```

- (b) Defina a função com recursividade explícita. Note que

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

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

- (b) Defina agora a função usando um `foldr`.

```
fromDigits'' :: [Int] -> Int  
fromDigits'' l = foldr (\x y -> x + (y * 10)) 0 l
```

3. Usando as funções anteriores e as funções `intToDigit :: Int -> Char` e `digitToInt :: Char -> Int` do módulo `Data.Char`,

- (a) Defina a função `intStr :: Int -> String` que converte um inteiro numa string. Por exemplo, `intStr 1234` deve corresponder à string `"1234"`.

```
intStr :: Int -> String  
intStr ints = map intToDigit (reverse (toDigits ints))
```

- (c) Defina a função `strInt :: String -> Int` que converte a representação de um inteiro (em base 10) nesse inteiro. Por exemplo, `strInt "12345"` deve corresponder ao número 12345.

```
strInt :: String -> Int  
strInt str = fromDigits (reverse (map digitToInt str))
```

4. Defina a função `agrupa :: String -> [(Char,Int)]` que dada uma string, junta num par  $(x,n)$  as  $n$  ocorrências consecutivas de um carácter  $x$ . Por exemplo, `agrupa 'aaakkkkkwaa'` deve dar como resultado a lista `[('a',3), ('k',5), ('w',1), ('a',2)]`. Sugestão: use a função `span`.

```
agrupa :: String -> [(Char,Int)]
agrupa [] = []
agrupa str = let (a,b) = span ((head str) == ) str in (head a, length a) : agrupa b
```

5. Defina a função `subLists :: [a] -> [[a]]` que calcula todas as sublistas de uma lista; por exemplo,  
`subLists [1,2,3] = [[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]`.

```
subList :: [a] -> [[a]]
subList [] = [[]]
subList (h:ts) = (map (\xs -> h:xs) (subList ts)) ++ (subList ts)
```

6. Considere a seguinte definição para representar matrizes:

```
type Mat a = [[a]]
```

Por exemplo, a matriz (triangular superior)

1	2	3
0	4	5
0	0	6

seria representada por

```
[[1,2,3], [0,4,5], [0,0,6]]
```

Defina as seguintes funções sobre matrizes (use, sempre que achar apropriado, funções de ordem superior).

- (a) `dimOK :: Mat a -> Bool` que testa se uma matriz está bem construída (i.e., se todas as linhas têm a mesma dimensão).

```
dimOk :: Mat a -> Bool
dimOk [] = False
dimOk (h:ts) = dAux (length h) ts where
  dAux x [] = if (x > 0) then True else False
  dAux x (h:ts) = if (x == (length h)) then dAux x ts else False
```

- (b) `dimMat :: Mat a -> (Int,Int)` que calcula a dimensão de uma matriz.

```
dimMat :: Mat a -> (Int,Int)
-- supondo dimOk m == True
dimMat m | dimOk m = (length m, length (head m))
```

- (c) `addMat :: (Num a) => Mat a -> Mat a -> Mat a` que adiciona duas matrizes.

```
addMat :: (Num a) => Mat a -> Mat a -> Mat a
-- ambas matrizes têm as dimensões iguais;
addMat [] [] = []
addMat xs ys = zipWith (zipWith (+)) xs ys
```

- (c) `transpose :: Mat a -> Mat a` que calcula a transposta de uma matriz.

```
transpose :: Mat a -> Mat a
transpose [] = []
transpose mat = if (length (head mat) > 1) then (map head mat) : (transpose (map
tail mat)) else [(map head mat)]
```

(d) `multMat :: Num a => Mat a -> Mat a -> Mat a` que calcula o produto de duas matrizes.

```
multMat :: (Num a) => Mat a -> Mat a -> Mat a  
-- supondo que snd (dimMat m1) == fst (dimMat m2)  
multMat [] _ = []  
multMat m1 m2 = (map (\x -> (sum (zipWith (*) (head m1) x))) (transpose m2)) :  
(multMat (tail m1) m2)
```

(f) `zipWMat :: (a -> b -> c) -> Mat a -> Mat b -> Mat c` que, à semelhança do que acontece com a função `zipWith`, combina duas matrizes. Use essa função para definir uma função que adiciona duas matrizes.

```
zipWMat :: (a -> b -> c) -> Mat a -> Mat b -> Mat c
```

```
zipWMat f mA mB = zipWith (zipWith f) mA mB
```

(g) `triSup :: Num a => Mat a -> Bool` que testa se uma matriz quadrada é triangular superior (i.e., todos os elementos abaixo da diagonal são nulos).

```
triSup :: (Eq a, Num a) => Mat a -> Bool
```

```
triSup [] = False
```

```
triSup mat = tAux 0 mat where
```

```
tAux x [] = True
```

```
tAux x (h:ts) = if ((length (fst (span (0 ==) h))) >= x) then tAux (x + 1) ts else False
```

(h) `rotateLeft :: Mat a -> Mat a` que roda uma matriz 90° para a esquerda. Por exemplo, o resultado de rodar a matriz acima apresentada deve corresponder à

```
matriz  

|   |   |   |
|---|---|---|
| 3 | 5 | 6 |
| 2 | 4 | 0 |
| 1 | 0 | 0 |


```

```
rotateLeft :: Mat a -> Mat a  
rotataLeft [] = []  
rotateLeft m = if (length (head m)) >  
1 then (rotateLeft (map (drop 1) m))  
++ [map head m] else [map head m]
```