Ficha 5

Programaçao Funcional

2015/16

1. Uma forma de representar polinómios de uma variável é usar listas de monómios representados por pares (coeficiente, expoente)

```
type Polinomio = [Monomio]
type Monomio = (Float,Int)
```

Por exemplo, `[(2,3), (3,4), (5,3), (4,5)]` representa o polinómio $2x^3 + 3x^4 + 5x^3 + 4x^5$.

(a) Defina uma funcao `conta :: Int -> Polinomio -> Int` de forma a que `(conta n p)` indica quantos monomios de grau `n` existem em `p`.

conta :: Int -> Polinomio -> Int
conta _ [] = 0
conta n (h:ts) = if (n == (snd(h))) then 1 + (conta n ts) else 0 + (conta n ts)

(b) Defina a funçao `grau :: Polinomio -> Int` que indica o grau de um polinómio.

grau :: Polinomio -> Int
grau [(c,e)] = e
grau ((c,e):(cs,es):ts) = if (es > e) then grau ((cs,es):ts) else grau ((c,e):ts)

(c) Defina `selgrau :: Int -> Polinomio -> Polinomio` que selecciona os monómios com um dado grau de um polinómio. Use uma funcão de ordem superior.

selgrau :: Int -> Polinomio -> Polinomio
selgrau g poli = filter (\(c,e) -> e == g) poli

(d) Complete a definiçao da função `deriv` de forma a que esta calcule a derivada de um polinomio.

```
deriv :: Polinomio -> Polinomio
deriv p = map ........ p
```

deriv :: Polinomio -> Polinomio
deriv pol = map (\(c,e) -> (c * (fromIntegral e),e - 1)) pol

(e) Defina a funcão `calcula :: Float -> Polinomio -> Float` que calcula o valor de um polinomio para uma dado valor de x.

calcula :: Float -> Polinomio -> Float
calcula _ [] = 0
calcula x ((c,e):ts) = ((x ^ e) * c) + (calcula x ts)

(f) Defina a funçao `simp :: Polinomio -> Polinomio` que retira de um polinómio os monomios de coeficiente zero. De preferência, use funções de ordem superior.

simp :: Polinomio -> Polinomio
simp pol = filter (\(c,e) -> c /= 0) pol

(g) Complete a definiçao da função `mult` de forma a que esta calcule o resultado da multiplicaçao de um monomio por um polinómio.

```
mult :: Monomio -> Polinomio -> Polinomio
mult (c,e) p = map ........ p
```

```
mult :: Monomio -> Polinomio -> Polinomio
mult _ [] = []
mult (c,e) p = map (\(x,y) -> (x * c,e + y)) p
```

(h) Defina `normaliza :: Polinomio -> Polinomio` que que dado um polinómio constroi um polinomio equivalente em que nao podem aparecer varios monómios com o mesmo grau.

```
normaliza :: Polinomio -> Polinomio
normaliza [] = []
normaliza (h:ts) = acresAux h (normaliza ts) where
    acresAux :: Monomio -> Polinomio -> Polinomio
    acresAux (c,e) [] = [(c,e)]
    acresAux (c,e) ((x,y):ts) = if (e == y) then (c + x, e):ts else (x,y) : (acresAux (c,e) ts)
```

(i) Defina a funçao `soma :: Polinomio -> Polinomio -> Polinomio` que faz a soma de dois polinomios de forma que se os polinómios que recebe estiverem normalizados produz também um polinómio normalizado.

```
soma :: Polinomio -> Polinomio -> Polinomio
soma [] poli = poli
soma (h:ts) poli = soma ts (acresAux h poli) where
    acresAux :: Monomio -> Polinomio -> Polinomio
    acresAux (c,e) [] = [(c,e)]
    acresAux (c,e) ((x,y):ts) = if (e == y) then (c + x, e):ts else (x,y) : (acresAux
(c,e) ts)
```

(j) Defina a funcao `produto :: Polinomio -> Polinomio -> Polinomio` que calcula o produto de dois polinómios

```
produto :: Polinomio -> Polinomio -> Polinomio
produto [] poli = poli
produto poli [] = poli
produto [x] poli = mult x poli
produto (x:xs) poli = (mult x poli) ++ (produto xs poli)
```

(k) Defina a funçao `ordena :: Polinomio -> Polinomio` que ordena um polonomio por ordem crescente dos graus dos seus monomios.

```
ordena :: Polinomio -> Polinomio
ordena [] = []
ordena (h:ts) = ordAux h (ordena ts) where
    ordAux m [] = [m]
    ordAux (c,e) ((x,y):ts) = if (e < y) then (c,e):(x,y):ts else (x,y) : (ordAux (c,e) ts)
```

(l) Defina a funcão `equiv :: Polinomio -> Polinomio -> Bool` que testa se dois polinomios sao equivalentes.

```
equiv :: Polinomio -> Polinomio -> Bool
equiv poli1 poli2 = (ordena (normaliza poli1)) == (ordena (normaliza poli2))
```

2. Defina uma funçao `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 funcão que definiu percorre a lista apenas uma vez.

```
nzp :: [Int] -> (Int,Int,Int)
nzp [] = (0,0,0)
nzp (h:ts) | h < 0 = (1 + a,0 + b,0 + c)
           | h == 0 = (0 + a,1 + b,0 + c)
           | h > 0 = (0 + a,0 + b,1 + c)
  where (a,b,c) = nzp ts
```

3. Defina a funcao `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 funcões `isDigit, isAlpha::Char->Bool` estao j definidas no modulo `Data.Char`).

```
digitAlpha :: String -> (String,String)
digitAlpha [] = ([],[])
digitAlpha (h:ts) | isAlpha h = (h:a,b)
                  | isDigit h = (a,h:b)
                  | otherwise = (a,b)
      where (a,b) = digitAlpha ts
```

4. Para cada uma das expressoes seguintes, exprima por enumeracão 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]`

```
l4a' = [x | x <- [2,4..20], mod x 3 == 0]
```

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

```
l4b' = [x | x <- [1..20], mod x 6 == 0]
```

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

```
l4c' = [(x,y) | x <- [10,20], y <- [20..10], x+y == 30]
```

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

```
l4d' = [concat [[x**2] ++ [x**2] | x <- [1..5]]]
```

5. Defina cada uma das listas seguintes por compreensao.

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

```
l5a = [2^x | x <- [0..10]]
```

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

```
l5b' = [(x,6-x) | x <- [1..5]]
```

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

```
l5c = [[1..x] | x <- [1..5]]
```

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

```
l5d = [[1 | x <- [1..y]] | y <- [1..5]]
```

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

```
l5e = [product [1..x] | x <- [1..6]]
```

6. Apresente definicoes das seguintes funcões de ordem superior, já pré-definidas no Prelude:

   (a) `zipWith :: (a->b->c) -> [a] -> [b] -> [c]` que combina os elementos de duas listas usando uma função específica; por exemplo:
   `zipWith (+) [1,2,3,4,5] [10,20,30,40] = [11,22,33,44].`
   zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
   zipwith' _ [] _ = []
   zipWith' _ _ [] = []
   zipWith' f (x:xs) (y:ys) = (f x y) : (zipWith' f xs ys)

   (b) `takeWhile :: (a->Bool) -> [a] -> [a]` que determina os primeiros elementos da lista que satisfazem um dado predicado; por exemplo:
   `takeWhile odd [1,3,4,5,6,6] = [1,3].`
   takeWhile' :: (a -> Bool) -> [a] -> [a]
   takeWhile' _ [] = []
   takeWhile' f (h:ts) = if (f h) then h : (takeWhile' f ts) else []

   (c) `dropWhile :: (a->Bool) -> [a] -> [a]` que elimina os primeiros elementos da lista que satisfazem um dado predicado; por exemplo:
   `dropWhile odd [1,3,4,5,6,6] = [4,5,6,6].`
   dropWhile' :: (a -> Bool) -> [a] -> [a]
   dropWhile' _ [] = []
   dropWhile' f (h:ts) = if (f h) then dropWhile' f ts else (h:ts)

   (d) `span :: (a-> Bool) -> [a] -> ([a],[a])`, que calcula simultaneamente os dois resultados anteriores. Note que apesar de poder ser definida à custa das outras duas, usando a definição

   ```
   span p l = (takeWhile p l, dropWhile p l)
   ```

   nessa definicão há trabalho redundante que pode ser evitado. Apresente uma definiçao alternativa onde nao haja duplicaçao de trabalho.
   span' :: (a -> Bool) -> [a] -> ([a],[a])
   span' _ [] = ([],[])
   span' f (h:ts) = if (f h) then (h:a,b) else ([],(h:ts)) where
   (a,b) = span' f ts

   span'' :: (a -> Bool) -> [a] -> ([a],[a])
   span'' f list = sAux f list [] where
   sAux _ [] new = (new,[])
   sAux f (h:ts) new = if (f h) then sAux f ts (new ++ [h]) else (new,(h:ts))