

Programação Funcional

Ficha 2

Funções recursivas sobre listas

1. Indique como é que o interpretador de haskell avalia as expressões das alíneas que se seguem, apresentando a cadeia de redução de cada uma dessas expressões (i.e., os vários passos intermédios até se chegar ao valor final).

- (a) Considere a seguinte definição:

```
funA :: [Double] -> Double
funA [] = 0
funA (y:ys) = y^2 + (funA ys)
```

Diga, justificando, qual é o valor de `funA [2,3,5,1]`.

- (b) Considere seguinte definição:

```
funB :: [Int] -> [Int]
funB [] = []
funB (h:t) = if (mod h 2) == 0 then h : (funB t)
              else (funB t)
```

Diga, justificando, qual é o valor de `funB [8,5,12]`

- (c) Considere a seguinte definição:

```
funC (x:y:t) = funC t
funC [x] = []
funC [] = []
```

Diga, justificando, qual é o valor de `funC [1,2,3,4,5]`.

- (d) Considere a seguinte definição:

```
funD 1 = g [] 1
g 1 [] = 1
g 1 (h:t) = g (h:1) t
```

Diga, justificando, qual é o valor de `funD "otrec"`.

2. Defina recursivamente as seguintes funções sobre listas:

- (a) `dobros :: [Float] -> [Float]` que recebe uma lista e produz a lista em que cada elemento é o dobro do valor correspondente na lista de entrada.
- (b) `numOcorre :: Char -> String -> Int` que calcula o número de vezes que um carácter ocorre numa string.
- (c) `positivos :: [Int] -> Bool` que testa se uma lista só tem elementos positivos.
- (d) `soPos :: [Int] -> [Int]` que retira todos os elementos não positivos de uma lista de inteiros.
- (e) `somaNeg :: [Int] -> Int` que soma todos os números negativos da lista de entrada.
- (f) `tresUlt :: [a] -> [a]` devolve os últimos três elementos de uma lista. Se a lista de entrada tiver menos de três elementos, devolve a própria lista.

- (g) `segundos :: [(a,b)] -> [b]` que calcula a lista das segundas componentes dos pares.
 - (h) `nosPrimeiros :: (Eq a) => a -> [(a,b)] -> Bool` que testa se um elemento aparece na lista como primeira componente de algum dos pares.
 - (i) `sumTriplos :: (Num a, Num b, Num c) => [(a,b,c)] -> (a,b,c)` soma uma lista de triplos componente a componente.
 Por exemplo, `sumTriplos [(2,4,11), (3,1,-5), (10,-3,6)] = (15,2,12)`
3. Recorrendo a funções do módulo `Data.Char`, defina recursivamente as seguintes funções sobre strings:
- (a) `soDigitos :: [Char] -> [Char]` que recebe uma lista de caracteres, e selecciona dessa lista os caracteres que são algarismos.
 - (b) `minusculas :: [Char] -> Int` que recebe uma lista de caracteres, e conta quantos desses caracteres são letras minúsculas.
 - (c) `nums :: String -> [Int]` que recebe uma string e devolve uma lista com os algarismos que ocorrem nessa string, pela mesma ordem.
4. 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 $2x^3 + 3x^4 + 5x^3 + 4x^5$. Defina as seguintes funções:

- (a) `conta :: Int -> Polinomio -> Int` de forma a que (`conta n p`) indica quantos monómios de grau `n` existem em `p`.
- (b) `grau :: Polinomio -> Int` que indica o grau de um polinómio.
- (c) `selgrau :: Int -> Polinomio -> Polinomio` que selecciona os monómios com um dado grau de um polinómio.
- (d) `deriv :: Polinomio -> Polinomio` que calcula a derivada de um polinómio.
- (e) `calcula :: Float -> Polinomio -> Float` que calcula o valor de um polinómio para um dado valor de x .
- (f) `simp :: Polinomio -> Polinomio` que retira de um polinómio os monómios de coeficiente zero.
- (g) `mult :: Monomio -> Polinomio -> Polinomio` que calcula o resultado da multiplicação de um monómio por um polinómio.
- (h) `normaliza :: Polinomio -> Polinomio` que dado um polinómio constrói um polinómio equivalente em que não podem aparecer varios monómios com o mesmo grau.
- (i) `soma :: Polinomio -> Polinomio -> Polinomio` que faz a soma de dois polinómios de forma que se os polinómios que recebe estiverem normalizados produz também um polinómio normalizado.
- (j) `produto :: Polinomio -> Polinomio -> Polinomio` que calcula o produto de dois polinómios
- (k) `ordena :: Polinomio -> Polinomio` que ordena um polonómio por ordem crescente dos graus dos seus monómios.
- (l) `equiv :: Polinomio -> Polinomio -> Bool` que testa se dois polinómios são equivalentes.