Paradigmas da Programação I (LESI)
Programação Funcional (LMCC)

Ficha 3

Ano lectivo 2004/05

1 O uso de acumuladores

Relembre a função factorial

```
fact 0 = 1
fact n | n>0 = n * fact (n-1)
```

O cálculo da factorial de um número positivo n é feito multiplicando n pelo factorial de (n-1). A multiplicação fica em suspenso até que o valor de fact (n-1) seja sintetizado.

Uma outra estratégia para resolver o mesmo problema, consiste em definir uma função auxiliar com um parametro extra que serve para ir guardando os resultados parciais – a este parametro extra chama-se *acumulador*.

```
fact n | n >=0 = factAc 1 n
  where factAc ac 0 = ac
        factAc ac n = factAc (ac*n) (n-1)
```

A utilização de uma ou de outra estratégia na definição de funções, para além de ser uma questão de estilo, pode ter implicações na eficiencia das funções que estão a ser definidas.

Tarefa 1 Usando as duas estratégias (com/sem acumuladores) defina duas versões de cada uma das funções que se seguem:

1. Função que calcula o produtório de uma lista de números reais.
2. Função que calcula o valor mínimo de uma lista de inteiros.
3. Função que inverte uma lista.

2 Mais alguns problemas

Manipulação de palavras e textos

Uma palavra é uma sequência de caracteres e, como tal, pode ser implementada directamente pelo tipo `String`. Um texto é aqui visto como uma sequência de palavras. Considere a seguinte definição de tipo:

```
type Texto = [String]
```

Tarefa 2 Algumas funções sobre palavras:

1. Defina uma função que converte todas as letras de uma palavra em letras maiúsculas. (Sugestão: use a função pré-definida `toUpper`.)

2. Defina uma função que, dado um número inteiro não negativo n, gera a palavra de comprimento n só de caracteres '_'.
3. Defina uma função que testa de duas palavras são iguais, sem fazer a distinção entre letras maiúsculas e minúsculas.
4. Defina uma função que coloca todas as palavras começadas por '#' entre chavetas retirando-lhes o caracter '#' inicial.
5. Defina uma função que detecta se uma palavra concorda com uma dada *palavra padrão*. Por padrão, aqui, entende-se uma sequência de caracteres que pode conter o caracter '?' em qualquer parte e '*' só no fim. Estes caracteres têm significados especiais:

   ? pode ser (substituido por) qualquer caracter;

   * pode ser (substituido por) qualquer sequência de caracteres (mas só pode aparecer no fim).

   Por exemplo: as palavras "infeliz", "infelizes" e "infelizmente", concordam com o padrão "??feliz*"; as palavras "prato", "preto" e "prata", concordam com a palavra padrão "pr?t?".

Tarefa 3 Algumas funções sobre textos:

1. Defina uma função que conta o número de caracteres de um texto.
2. Defina uma função que conta o número de ocorrências de uma palavra num texto.
3. Defina uma função que dada uma palavra e um texto devolve a lista com as posições em que essa palavra ocorre no texto.
4. Agora queremos saber o número de ocorrências de *cada* palavra de um texto. Para esse fim:
   (a) Defina a função
   ```
   conta :: String -> [(String,Int)] -> [(String,Int)]
   ```
   que pega numa palavra p e numa lista de pares (palavra, n^o de ocorrências) e acrescenta a contagem de p à lista.
   (b) Utilize a função da alínea anterior para codificar a função que conta o número de ocorrências de cada palavra do texto. (Sugestão: utilize um parametro de acumulação.)
   (c) Utilize a função da alínea anterior para codificar a função que dado um texto indica uma das palavras que mais ocorre nesse texto.
5. Defina uma função para "esconder" texto. Isto é, uma função que recebe um texto e uma lista de inteiros positivos, e substitui as palavras que estão nas posições indicadas na lista por uma string de '_' de tamanho igual à palavra original. (Sugestão: use a função que definiu na tarefa 2)
6. Defina uma função que seleciona de um texto todas as palavras que concordam com uma dada palavra-padrão. (Sugestão: use a função que definiu na tarefa 2)

Polinómios

Uma forma de representar polinómios de uma variável é usar listas de pares (coeficiente, expoente)

```
type Pol = [(Float,Int)]
```

Note que o polinómio pode não estar simplificado. Por exemplo,

```
[(3.4,3), (2.0,4), (1.5,3), (7.1,5)]  :: Pol
```

representa o polinómio $3.4\,x^3 + 2\,x^4 + 1.5\,x^3 + 7.1\,x^5$.

Tarefa 4 1. Sugira representações para os polinómios:

(a) $x^7 + 3x^2$

(b) $5x^3 - 3x^6 + 2$

2. Defina uma função para somar dois polinómios nesta representação.
3. Defina a função de cálculo do valor de um polinómio num ponto.
4. Defina uma função que dado um polinómio, calcule o seu grau.
5. Defina uma função que calcule a derivada de um polinómio.
6. Defina uma função que calcule o produto de dois polinómios.
7. Será que podemos ter nesta representação de polinómios, monómios com expoente negativo ? As funções que definiu contemplam estes casos ?

3 Funções de ordem superior

Funções de *ordem superior* são funções que recebem funções como parametro e/ou retornam funções como resultado. Por exemplo, `map` e `filter` são funções de ordem superior.

- A função `map` pode ser definida do seguinte modo:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x) : (map f xs)
```

  Assim, `(map f l)` aplica a função `f` a todo o elemento da lista `l`. Observe a concordância de tipos entre os elementos da lista `l` e o dominio da função `f`. Repare ainda que o resultado de `(map f l)` é o mesmo de `[ f x | x <- l ]`.

- A função `filter` pode ser definida por:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = if (p x) then x:(filter p xs)
                           else filter p xs
```

  Ou seja, `(filter p l)` seleciona/filtra da lista `l` os elementos que satisfazem o predicado `p`. Observe a concordância de tipos entre os elementos da lista `l` e o dominio do predicado `p`. Repare ainda que o resultado de `(filter p l)` é o mesmo de `[ x | x <- l, p x ]`.

Tarefa 5 Para cada uma das expressões seguintes, determine o seu valor e reescreva-a usando listas por compreensão.

1. `map odd [1,2,3,4,5]`
2. `filter odd [1,2,3,4,5]`
3. `map (\x-> div x 3) [5,6,23,3]`
4. `filter (\y-> (mod y 3 == 0)) [5,6,23,3]`
5. `filter (7<) [1,3..15]`
6. `map (7:) [[2,3],[1,5,3]]`
7. `map (:[]) [1..5]`
8. `map succ (filter odd [1..20])`
9. `filter odd (map succ [1..20])`

Temos, portanto, várias formas de definir funções sobre listas. Considere o exemplo da função `dobra :: [Int] -> [Int]` que recebe uma lista de inteiros e produz a lista dos seus dobros. Podemos definir `dobra` de diferentes modos:

Versão recursiva

```
dobra1 :: [Int] -> [Int]
dobra1 [] = []
dobra1 (x:xs) = (2*x):(dobra1 xs)
```

Usando listas por compreensão

```
dobra2 :: [Int] -> [Int]
dobra2 l = [ 2*x | x <- l ]
```

Usando funções de ordem superior

```
dobra3 :: [Int] -> [Int]
dobra3 l = map (2*) l       -- ou, alternativamente:   dobra3 = map (2*)
```

Tarefa 6 Cada uma das funções que se seguem está definida com base em algum destes três modos. Defina funções equivalentes, usando os outros dois modos.

1.
```
pares :: [Integer] -> [Integer]
pares [] = []
pares (x:xs) | even x    = x:(pares xs)
             | otherwise = pares xs
```

2.
```
maximos :: [(Double,Double)] -> [Double]
maximos l = [ max x y | (x,y) <- l ]
```

3.
```
aprovados :: [(Float,Float)] -> [(Float,Float)]
aprovados notas = filter passa notas
  where passa :: (Float,Float) -> Bool
        passa (p,t) = (p >= 9) && (t >= 9) && (0.4*p)+(0.6*t) > 9.9
```

4.
```
notasAprov :: [(Float,Float)] -> [Integer]
notasAprov [] = []
notasAprov ((np,nt):ns) = let nf = (0.4*np)+(0.6*nt)
                          in if (np >= 9) && (nt >= 9) && nf > 9.9
                                then (round nf):(notasAprov ns)
                                else notasAprov ns
```

5.
```
indicativo  :: [Int] -> [[Int]] -> [[Int]]
indicativo ind telefs = filter (concorda ind) telefs
  where concorda :: [Int] -> [Int] -> Bool
        concorda [] _ = True
        concorda (x:xs) (y:ys) = (x==y) && (concorda xs ys)
        concorda (x:xs) [] = False
```

Outros dois exemplos de funções de ordem superior, são as funções `foldr` e `foldl`, que fazem a extensão de uma operação binária a uma lista de operandos. Estas funções recebem três argumentos: a função binária, o valor inicial e a lista.

- A função `foldr` pode ser definida por:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

  Observe que `(foldr f z [x1,...,xn])` é igual a `(f x1 (...(f xn z)...))`.

  Uma forma simplista de ver a operação do `foldr` é substituir as ocorrências do construtor ':' das listas pela função binária infixa, e fazer o cálculo associando à direita.

- A função `foldl` pode ser definida por:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

  Observe que `(foldl f z [x1,...,xn])` é igual a `(f (...(f z x1)...)  xn)`.

Tarefa 7 Seja `lista = (10:8:3:[])`. Para cada uma das expressões seguintes, determine o seu valor e justifique o resultado.

1. `foldr (+) 0 lista`
2. `foldl (+) 0 lista`
3. `foldr (-) 1 lista`
4. `foldl (-) 1 lista`
5. `foldr (\(x,y) z -> x*y+z) 0 [(3,5), (7,2), (4,1)]`
6. `foldl (\z (x,y) -> x*y+z) 0 [(3,5), (7,2), (4,1)]`

Tarefa 8 Para cada uma das funções a seguir, apresente uma definição recursiva e uma outra (ou várias outras) usando alguma(s) das funções `map`, `filter`, `foldr` ou `foldl`.

1. `quadrado :: Int -> Int`, a função que calcula o quadrado de um número. Note que o quadrado de um número n pode ser calculado pela soma dos n primeiros números impares.
2. `maxPos :: [Int] -> Int`, a função que calcula o maior valor de uma lista de inteiros positivos.
3. `append :: [Int] -> [Int] -> [Int]`, a função que concatena duas listas de inteiros (sem usar a função `++`).
4. `inverte :: [Int] -> [Int]`, a função que inverte lista de inteiros (sem usar a função `reverse`).

Tarefa 9 Reveja as soluções que apresentou para os problemas das tarefas 2, 3 e 4, e apresente soluções alternativas usando funções de ordem superior sempre que tal for possível e conveniente.

Tarefa 10 Leia a seguinte mensagem:

> De aorcdo com uma pqsieusa de uma uinrvesriddae ignlsea, não ipomtra a odrem plea qaul as lrteas de uma plravaa etaso, a úncia csioa iprotmatne é que a piremria e útmlia lrteas etejasm no lgaur crteo.
>
> O rseto pdoe ser uma ttaol csãofnuo que vcoe pdoe anida ler sem gnderas pobrl-maes. Itso é poqrue nós não lmeos cdaa lrtea isladoa, mas a plravaa cmoo um tdoo.
>
> Cosiruo não ?

Com certeza conseguiu entender a mensagem!

Defina então uma função com tipo `String -> String`, que recebe um texto e devolve o texto com as letras interiores das palavras, permutadas de lugar (conforme é sugerido na mensagem).

Observações:

- O tipo `String` é igual a `[Char]`.
- Existem já definidas no Prelude as funções:

  `words :: String -> [String]` que parte uma string na lista de palavras;

  `unwords :: [String] -> String` que é a inversa da função `words`;

  `last :: [a] -> a` que retorna o último elemento de uma lista;

  `init :: [a] -> [a]` que retira o último elemento da lista.
- Para fazer a permutação de letras damos uma sugestão: a função `mix`. Veja se entende bem o funcionamento desta função.

```
mix :: [a] -> [a]
mix [] = []
mix [x] = [x]
mix (x1:x2:x3:xs) = x3:x1:x2:(mix xs)
mix (x1:x2:xs) = x2:x1:(mix xs)
```

Tarefa 11 Uma forma de representar uma linha quebrada no plano cartesiano é através de uma sequência de pontos. Cada ponto é representado por um par de números reais. Assim, por exemplo, `[(2.3,4.0), (1.5,0.5), (7.0,-2.2)]` representa a linha constituida pelos segmentos de recta definidos pelos pontos (2.3,4.0) e (1.5,0.5), e pelos pontos (1.5,0.5) e (7.0,-2.2).

1. Defina uma função que calcule o comprimento de uma linha quebrada.
2. Defina uma função que dado um vector de translação e uma linha quebrada, calcula as coordenadas do resultado da translação.