

Ficha Prática 3

Programação Funcional CC

LCC 1^o ano (2007/2008)

Resumo

Nesta aula pretende-se por um lado trabalhar sobre padrões de funções recursivas sobre listas (mapeamento, filtragem, e *folding*), e por outro utilizar funções de ordem superior (`map`, `filter` e `foldr`) para definir de forma mais expedita essas mesmas funções. Finalmente, trabalhar-se-á sobre a definição de outras funções de ordem superior.

Conteúdo

1	Funções de Mapeamento, Filtragem, e <i>Folding</i> sobre Listas	2
2	As Funções <code>map</code> e <code>filter</code>	4
3	A Função <code>foldr</code>	6
4	Outras Funções	7

1 Funções de Mapeamento, Filtragem, e *Folding* sobre Listas

Certas funções recursivas sobre listas seguem padrões rígidos, o que permite classificá-las nas seguintes três categorias:

1. *Mapeamento*: aplicam a todos os elementos da lista argumento uma mesma função, obtendo-se como resultado uma lista com a mesma dimensão. Por exemplo:

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

impares :: [Int] -> [Bool]
impares [] = []
impares (x:xs) = (odd x):(impares xs)
```

2. *Filtragem*: calculam como resultado uma sub-sequência da lista argumento, contendo (pela mesma ordem) apenas os elementos que satisfazem um determinado critério. Por exemplo:

```
umaouduasletras :: [String] -> [String]
umaouduasletras [] = []
umaouduasletras (x:xs)
  | (length x)<= 2 = x:(umaouduasletras xs)
  | otherwise      = umaouduasletras xs

filtra_impares :: [Int] -> [Int]
filtra_impares [] = []
filtra_impares (x:xs)
  | odd x      = x:(filtra_impares xs)
  | otherwise  = filtra_impares xs
```

3. *Folding*: Combinam através de uma operação binária todos os elementos da lista. Mais exactamente, *iteram* uma operação binária sobre uma lista, o que corresponde às bem conhecidas operações matemáticas de “somatório” ou “produtório” sobre conjuntos. Para a lista vazia resulta um qualquer valor constante (tipicamente o elemento neutro da operação binária, mas pode ser outro qualquer valor). Exemplos:

```
somaLista :: [Int] -> Int
somaLista [] = 0
somaLista (x:xs) = x+(somaLista xs)
```

```
multLista :: [Int] -> Int
multLista []      = 1
multLista (x:xs) = x*(multLista xs)
```

Tarefa 1

Defina as seguintes três novas funções, e diga se correspondem a algum dos padrões acima referidos.

1. Função **paresord** que recebe uma lista de pares de números e devolve apenas os pares em que a primeira componente é inferior à segunda.
2. Função **myconcat** que recebe uma lista de strings e as junta (concatena) numa única string.
3. Função **maximos** que recebe uma lista de pares de números (de tipo **float**) e calcula uma lista contendo em cada posição o maior elemento de cada par.

2 As Funções `map` e `filter`

Para possibilitar a definição fácil de funções que seguem os padrões anteriormente mencionados, é possível captar-se esses padrões em funções recursivas mais abstractas. Para o caso do mapeamento e da filtragem, temos as duas seguintes funções (pré-definidas):

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

filter :: (a -> Bool) -> [a] -> [a]
filter p []      = []
filter p (x:xs)
  | p x          = x:(filter p xs)
  | otherwise    = filter p xs
```

Observe-se que estas funções, ditas de *ordem superior*, recebem como argumento uma outra função, que especifica, no caso de `map`, qual a operação a aplicar a cada elemento da lista, e no caso de `filter`, qual o critério de filtragem (dado por uma função de teste, ou *predicado*). Assim,

- (`map f l`) aplica a função `f` a todos os elementos da lista `l`. Observe a concordância de tipos entre os elementos da lista `l` e o domínio da função `f`.
- (`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 domínio do predicado `p`.

A utilização destas funções permite a definição de funções *implicitamente recursivas*. Por exemplo, uma nova versão da função `dobros` pode ser definida como:

```
dobros' l = map (2*) l
```

ou de forma ainda mais simples, `dobros' = map (2*)`.

Tarefa 2

Avalie cada uma das seguintes expressões

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])`

Tarefa 3

Defina novas versões de todas as funções de mapeamento e filtragem definidas na secção anterior, utilizando para isso as funções de ordem superior acima referidas.

Tarefa 4

Considere a função seguinte

```
indicativo :: String -> [String] -> [String]
indicativo ind telefns = filter (concorda ind) telefns
  where concorda :: String -> String -> Bool
        concorda [] _ = True
        concorda (x:xs) (y:ys) = (x==y) && (concorda xs ys)
        concorda (x:xs) [] = False
```

que recebe uma lista de Algarismos com um indicativo, uma lista de listas de Algarismos representando números de telefone, e selecciona os números que começam com o indicativo dado. Por exemplo:

```
indicativo "253" ["253116787","213448023","253119905"]
devolve ["253116787","253119905"].
```

Redefina esta função com recursividade explícita, isto é, evitando a utilização de `filter`.

Tarefa 5

Considere a função seguinte

```
abrev :: [String] -> [String]
abrev lnoms = map conv lnoms
  where conv :: String -> String
        conv nom = let ns = (words nom)
                    in if (length ns) > 1
                        then (head (head ns)):(". " ++ (last ns))
                        else nom
```

que converte uma lista de nomes numa lista de abreviaturas desses nomes, da seguinte forma: ["João Carlos Mendes", "Ana Carla Oliveira"] em ["J. Mendes", "A. Oliveira"].

Defina agora esta função com recursividade explícita, isto é, evitando a utilização de `map`.

3 A Função foldr

A função `foldr`, tal como `map` e `filter`, permite escrever de forma expedita, sem recursividade explícita, um grande conjunto de funções (incluindo as próprias funções `map` e `filter`).

O funcionamento desta função pode ser facilmente compreendido se se considerar que os constructores `(:)` e `[]` são simplesmente substituídos pelos dois parâmetros de `foldr`. Por exemplo, recordando que `[1,2,3] == 1:(2:(3:[]))`, tem-se que

```
foldr (+) 0 [1,2,3] => 1+(2+(3+0))
foldr (*) 1 [1,2,3] => 1*(2*(3*1))
```

Isto permite definir:

```
somaLista l = foldr (+) 0 l
multLista l = foldr (*) 1 l
```

Tarefa 6

Investigue o tipo e funcionamento de cada uma das seguintes funções do Haskell:

- *concat*
- *and*
- *or*

Escreva definições destas funções usando `foldr`.

Tarefa 7

Considere a seguinte definição de uma função que separa uma lista em duas partes de comprimento idêntico:

```
separa [] = ([],[])
separa (h:t) = (h:r,l)
  where (l,r) = separa t
```

Escreva uma nova definição desta função recorrendo a `foldr`.

4 Outras Funções

Na resolução das tarefas 8 e 9 utilize, sempre que lhe pareça natural, as funções `map`, `filter`, e `foldr`.

Tarefa 8

Pretende-se guardar a informação sobre os resultados dos jogos de uma jornada de um campeonato de futebol na seguinte estrutura de dados:

```
type Jornada = [Jogo]
type Jogo = ((Equipa,Golos), (Equipa,Golos))
type Equipa = String
type Golos = Int
```

Defina as seguintes funções:

1. `igualj :: Jornada -> Bool`
que verifica se nenhuma equipa joga com ela própria.
2. `semrepet :: Jornada -> Bool`
que verifica se nenhuma equipa joga mais do que um jogo.
3. `equipas :: Jornada -> [Equipa]`
que dá a lista das equipas que participam na jornada.
4. `empates :: Jornada -> [(Equipa,Equipa)]`
que dá a listas dos pares de equipas que empataram na jornada.
5. `pontos :: Jornada -> [(Equipa,Int)]`
que calcula os pontos que cada equipa obteve na jornada (venceu - 3 pontos; perdeu - 0 pontos; empatou - 1 ponto)

Tarefa 9

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 $3.4x^3 + 2x^4 + 1.5x^3 + 7.1x^5$.

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

Tarefa 10

Considere as duas seguintes funções:

```
merge  :: (Ord a) => [a] -> [a] -> [a]
insert :: (Ord a) => a -> [a] -> [a]
```

A primeira efectua a fusão de duas listas ordenadas de forma crescente; a segunda insere um elemento numa lista ordenada de forma crescente:

```
merge [1,4] [2,3] => [1,2,3,4]
insert "bb" ["aa","cc"] => ["aa","bb","cc"]
```

Uma definição possível de *insert* é

```
insert x [] = [x]
insert x (h:t)
  | (x<=h) = x:h:t
  | otherwise = h:(insert x t)
```

1. Escreva a função *merge* utilizando *foldr* e *insert*.
2. Relembre o algoritmo de ordenação insertion sort, implementado em Haskell por uma função:

```
isort :: (Ord a) => [a] -> [a]
```

Reescreva esta função utilizando *foldr*.