

## Funções de ordem superior

### BCC22 Programação Funcional — Aula Prática 01

Lucília Camarão de Figueiredo

A prática deve ser realizada antes da aula prevista para a discussão da mesma. Verifique a data prevista para a discussão de cada aula prática no calendário do curso disponível em [www.dcc.ufmg.br/~lucilia/cursos/func](http://www.dcc.ufmg.br/~lucilia/cursos/func). Traga para a aula de discussão o código que você implementou. A discussão não será eficaz se você não tentar fazer os exercícios previamente.

Você pode trabalhar em conjunto com outros colegas, mas deve entender o que está fazendo, pois não irá contar com a ajuda de colegas durante os exames.

A realização das práticas e presença nas aulas são obrigatórias e valem pontos na disciplina.

## 1 Funções de ordem superior

Funções em Haskell são valores que podem ser processados do mesmo modo que qualquer outro dado, tal como números, tuplas ou listas. Nesta prática você vai usar diversas funções de ordem superior, que tomam outras funções como argumento, para escrever definições sucintas de tarefas de processamento de listas. Algumas dessas tarefas você já implementou anteriormente, usando list comprehension ou recursão.

## 2 Map

Uma operação muito comum no processamento de listas é transformar todos os elementos da lista por meio da aplicação de uma função. Por exemplo:

- adicionar um a cada elemento de uma lista de número,
- extrair o primeiro componente de cada par de uma lista,
- converter cada caractere de um string para maiúscula, ou
- adicionar fundo cinza a cada figura de uma lista de figuras.

A função `map` captura esse padrão de computação, possibilitando ao programador evitar repetição de código que resultaria de ter que escrever uma função recursiva em cada caso.

Considere uma função `g` definida em termos de uma função imaginária `f` do seguinte modo:

```
g []      = []
g (x:xs) = f x : g xs
```

A função `g` pode ser escrita usando recursão (como acima), ou list comprehension, ou usando `map`: as três definições são equivalentes.



Figura 1: Função map

```

g xs = [ f x | x <- xs ]
g xs = map f xs

```

A definição de `map` é dada abaixo e ilustrada na figura 1. Note a semelhança com a definição recursiva de `g` (abaixo à esquerda). `map` tem um argumento a mais que `g`: a função `f` a ser aplicada a cada elemento da lista.

<code>g []</code>	<code>= []</code>	<code>map :: (a -&gt; b) -&gt; [a] -&gt; [b]</code>
<code>g (x:xs)</code>	<code>= f x : g xs</code>	<code>map f []</code>
		<code>= []</code>
		<code>map f (x:xs)</code>
		<code>= f x : map f xs</code>

Usando `map` e uma função que opera sobre um elemento da lista podemos facilmente escrever uma função que opera sobre toda a lista. Por exemplo, a função que extrai o primeiro componente de cada par da lista pode ser escrita do seguinte modo (usando `fst :: (a,b) -> a`):

```

fst :: [(a,b)] -> [a]
fst pairs = map fst pairs

```

## Exercícios

- Usando `map` e funções pré-definidas apropriadas, escreva definições para cada uma das seguintes funções:
  - A função `uppers :: String -> String` que converte um string para maiúsculas.
  - A função `doubles :: [Int] -> [Int]` que dobra cada elemento da lista.
  - A função `centsToReais :: [Int] -> [Float]` que converte cada preço em centavos para o preço equivalente em reais.
  - Escreva uma versão de `uppers` usando list comprehension e use-a para verificar a sua resposta para o exercício (a), usando `QuickCheck`.

## 3 Filter

Outra operação comum sobre listas é remover elementos que não satisfazem uma determinada propriedade. Por exemplo, remover caracteres não alfabéticos de um string, ou números negativos de uma lista de inteiros. Esse padrão de computação é capturado pela função `filter`.

Considere uma função `g`, definida em termos de um predicado imaginário `p`, do seguinte modo (um predicado é simplesmente uma função que retorna um valor booleano):

```

g []      = []
g (x:xs) | p x      = x :  g xs
          | otherwise = g xs

```

A função `g` pode ser escrita usando recursão (como acima) ou usando list comprehension, ou com `filter`: as três definições são equivalentes.

```

g xs = [ x | x <- xs, p x ]
g xs = filter p xs

```

Por exemplo, podemos escrever a função `evens :: [Int] -> [Int]`, que remove todos os elementos ímpares de uma lista, usando `filter` e a função pré-definida `even :: Int -> Bool`, que retorna `True` se o argumento é par e `False` caso contrário:

```
evens list = filter even list
```

Isso é equivalente a :

```
evens list = [x | x <- list, even x]
```

A definição de `filter` é dada abaixo. Note a semelhança com a definição recursiva de `g` (abaixo à esquerda). A função `filter` tem um argumento a mais que `g`: o predicado a ser usado para testar cada elemento.

<pre> g []      = [] g (x:xs)   p x      = x :  g xs             otherwise = g xs </pre>	<pre> filter :: (a -&gt; Bool) -&gt; [a] -&gt; [a] filter p []      = [] filter p (x:xs)   p x      = x :  filter p xs                     otherwise = filter p xs </pre>
--	---

## Exercícios

3. Usando `filter` e funções pré-definidas apropriadas, defina cada uma das seguintes funções:

- A função `alphas :: String -> String` que remove todos os caracteres não alfabéticos do string dado como argumento.
- A função `rmChar :: Char -> String -> String` que remove do string todas as ocorrências do caractere dado como primeiro argumento.
- A função `above :: Int -> [Int] -> [Int]` que remove da lista todos os números menores ou iguais ao número dado como primeiro argumento.
- A função `unequals :: [(Int,Int)] -> [(Int,Int)]` que remove da lista todos os pares `(x,y)` tais que `x == y`.
- Escreva uma versão da função `rmChar` usando list comprehension e use-a para testar a sua resposta ao exercício (b), usando QuickCheck.

## 4 List Comprehensions, map e filter

Como vimos, *list comprehensions* processam uma lista usando transformações similares a `map` e `filter`. De modo geral, `[f x | x <- xs, p x]` é equivalente a `map f (filter p xs)`.

### Exercícios

4. Escreva expressões equivalentes às expressões a seguir, usando `map` e `filter`. Use QuickCheck para verificar suas respostas.

- (a) `[toUpper c | c <- s, isAlpha c]`
- (b) `[2 * x | x <- xs, x > 3]`
- (c) `[reverse s | s <- strs, even (length s)]`

## 5 Fold

As funções `map` e `filter` agem sobre elementos da lista individualmente; elas nunca combinam um elemento com outro. Algumas vezes desejamos combinar elementos usando uma mesma operação: por exemplo, a função `sum`, que soma todos os elementos da lista pode ser escrita do seguinte modo:

```
sum []      = 0
sum (x:xs) = x + (sum xs)
```

Essencialmente, nesse código estamos combinando os elementos da lista usando a operação `+`.

Outro exemplo é a função `reverse`, que inverte uma lista:

```
reverse []      = []
reverse (x:xs) = reverse xs ++ [x]
```

Essa função apenas combina os elementos da lista, um a um, colocando-os no final da lista invertida. Nesse caso é um pouco mais difícil ver qual é a *função de combinação*. Isso fica mais fácil se reescrevermos o código do seguinte modo:

```
reverse []      = []
reverse (x:xs) = x 'snoc' (reverse xs)

snoc x xs = xs ++ [x]
```

Agora fica fácil ver que `'snoc'` tem aqui o mesmo papel que `+` no caso de `sum`.

Esses exemplos (e muitos outros) seguem um padrão: quebramos a lista em sua cabeça (`x`) e sua cauda (`xs`), aplicamos a função recursivamente em `xs` e então aplicamos uma dada função sobre `x` e a `xs` modificada. As únicas coisas que precisamos especificar são a função de combinação (tal como `(+)` ou `snoc`) e o valor inicial (tal como `0`, no caso de `sum` e `[]` no caso de `reverse`).

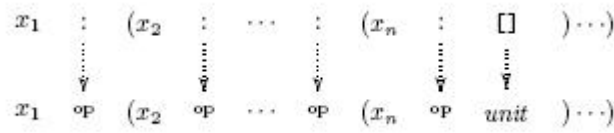


Figura 2: Função foldr

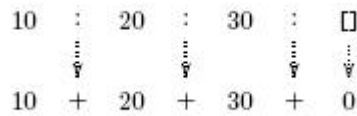


Figura 3: Ilustração de foldr (+) 0 [10,20,30]

Esse padrão é chamado um *fold* e é implementado em Haskell pela função foldr.

```

g []      = u
g (x:xs) = x 'f' g xs

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f u []      = u
foldr f u (x:xs) = x 'f' foldr f u xs

```

A função g pode ser escrita usando recursão (como acima) ou usando fold: as duas definições são equivalentes.

```
g xs = foldr f u xs
```

Uma forma de visualizar a ação de foldr é mostrada na figura 2. Dada uma função  $f :: a \rightarrow b \rightarrow b$ , e um valor inicial  $u :: b$  (algumas vezes chamado *unit*), e uma lista  $list :: [a]$ , a função foldr retorna o valor que resulta de substituir cada  $:$  (cons) na lista por  $f$  e substituir  $[]$  (nil) por  $u$ .

Por exemplo, podemos definir  $sum :: [Int] \rightarrow Int$  do seguinte modo, usando  $(+)$  e 0 como valor inicial:

```

sum :: [Int] -> Int
sum ns = foldr (+) 0 ns

```

(OBS: para tratar como função um operador infixado, tal como  $+$ , ele deve ser envolvido entre parênteses)

## Exercícios

- Você agora vai escrever diversas funções, primeiro usando recursão e em seguida usando foldr. Você pode usar funções pré-definidas nas bibliotecas de Haskell. Para cada função, escreva um teste QuickCheck para testar a equivalência entre as duas implementações.
  - Observe a função recursiva `productRec :: [Int] -> Int`, que computa o produto de todos os números de uma lista. Escreva uma função equivalente `productFold` usando foldr.
  - Escreva a função recursiva `andRec :: [Bool] -> Bool` que verifica se todos os elementos da lista são verdadeiros. Em seguida, escreva a função equivalente `andFold`, usando foldr.
  - Escreva a função recursiva `concatRec :: [String] -> String`, que junta todos os elementos de uma lista de strings em um único string. Em seguida, escreva a função análoga `concatFold`, usando foldr. *Note:* essas funções são similares às funções `product` e `concat` definidas na biblioteca `Prelude.hs`, mas a função `concat` tem o tipo mais geral `[[a]] -> [a]`.

- (d) Escreva a função recursiva `rmCharsRec :: String -> String -> String` que remove do segundo string todos os caracteres que ocorrem no primeiro string, usando a função `rmChar` do exercício (2b). Por exemplo

```
Main> rmCharsRec ['a'..'l'] "football"
"oot"
```

Em seguida, escreva a função equivalente `rmCharsFold`, usando `foldr`. Verifique a equivalência entre as duas implementações usando `QuickCheck`.

## 6 Exercícios adicionais

### Manipulação de Matrizes

Neste exercício, você vai implementar adição e multiplicação de matrizes. Uma matriz será representada como uma lista de listas de inteiros. Por exemplo:

$\begin{pmatrix} 1 & 4 & 9 \\ 2 & 5 & 7 \end{pmatrix}$  é representada como `[ [1,4,9], [2,5,7] ]`

A seguinte declaração de tipo, incluída no arquivo `pratica03.hs`, define `Matrix` como tipo sinônimo de `[[Int]]`.

```
type Matrix = [[Int]]
```

Sua primeira tarefa é escrever uma função para testar se uma lista de lista de inteiros de fato constitui uma matriz. Esse teste deve verificar duas coisas 1) que todas as linhas da matriz têm o mesmo comprimento, e 2) que existe pelo menos uma linha e pelo menos uma coluna na lista de listas.

### Exercícios

6. (a) Escreva a função `uniform :: [Int] -> Bool` que testa se os inteiros em uma lista são todos iguais. Você pode usar a função pré-definida `all`, que testa se todos os elementos de uma lista satisfazem um dado predicado (verifique o tipo da função `all` para saber como usá-la). Se preferir, você pode tentar definir tudo em termos de `foldr` e `map`.
- (b) Usando a função `uniform`, escreva a função `valid :: Matrix -> Bool` que testa se uma lista de listas de inteiros é uma matriz (isto é, se satisfaz as propriedades 1) e 2) especificadas acima).

Duas outras coisas que gostaríamos de determinar são as dimensões da matriz (número de linhas e colunas) e se a matriz é quadrada (número de linhas igual ao número de colunas). Uma função que poderá ser útil na sua implementação é a função `uncurry`, que toma como argumento uma função que recebe dois argumentos e retorna uma função equivalente que recebe esses dois argumentos como um par.

### Exercícios

7. (a) Verifique a definição de `uncurry`. Qual é o valor retornado pela seguinte expressão?

```
Main> uncurry (+) (10,8)
```

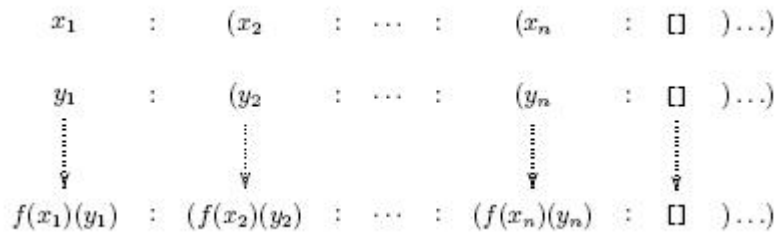


Figura 4: Ilustração de `zipWith` para listas de mesmo comprimento

- (b) Escreva a função `size :: Matrix -> (Int,Int)` que retorna o número de linhas e colunas da matriz, como um par (linhas, colunas).

```
Main> size [[1,2,3],[4,5,6]] (2,3)
```

- (c) Escreva a função `square :: Matrix -> Bool` que testa se a matriz é quadrada.

Outra função muito útil é `zipWith`. Essa função é muito parecida com a função `zip` que você já conhece, a qual combina duas listas em uma lista de pares. A diferença é que ao invés de combinar os elementos das duas listas em um par, `zipWith` recebe como argumento uma função, que é usada para combinar os elementos das duas listas. A definição de `zipWith` é a seguinte (A figura 4 ilustra o comportamento dessa função):

```
zipWith f [] _ = []
zipWith f _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

## Exercícios

8. Defina `zipWith` em termos de `map`, `zip` e `uncurry`.

Adicionar duas matrizes de mesma dimensão consiste em adicionar os elementos que ocorrem na mesma posição (i.e., mesma linha e coluna) nas duas matrizes, para obter o elemento nessa posição na matriz resultante. Por exemplo:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} + \begin{pmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \end{pmatrix} = \begin{pmatrix} 11 & 22 & 33 \\ 44 & 55 & 66 \end{pmatrix}$$

Vamos usar `zipWith` para implementar a adição de matrizes.

## Exercícios

9. Escreva a função `plusM` que soma duas matrizes. Essa função deve retornar um erro caso os argumentos não sejam apropriados. Pode ser útil definir uma função auxiliar `addRow` que adiciona duas linhas da matriz.

Para a multiplicação de matrizes, precisamos do chamado *produto interno*, ou produto de dois vetores:

$$(a_1, a_2, \dots, a_n) \cdot (b_1, b_2, \dots, b_n) = a_1 b_1 + a_2 b_2 + \dots + a_n b_n$$

A multiplicação de matrizes é definida do seguinte modo: duas matrizes de dimensões  $(n, m)$  e  $(m, p)$  são multiplicadas de modo a formar uma matriz de dimensão  $(n, p)$ , na qual o elemento na linha  $i$ , coluna  $j$  é o produto interno da linha  $i$  da primeira matriz pela coluna  $j$  da segunda matriz. Por exemplo:

$$\begin{pmatrix} 1 & 10 \\ 100 & 10 \end{pmatrix} \times \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 31 & 42 \\ 130 & 240 \end{pmatrix}$$

## Exercícios

10. Defina a função `timesM` que efetua a multiplicação de matrizes. Essa função deve retornar erro caso os argumentos não sejam apropriados. Pode ser útil definir uma função auxiliar `innerProduct` para fazer o produto interno de dois vetores (listas).

*Dica:* escreva uma função auxiliar `multRow :: [Int] -> Matrix -> [Int]` que multiplica uma linha de matriz por outra matriz. Essa função deve retornar uma lista, onde cada elemento é o produto interno da linha de matriz por cada uma das colunas da outra matriz. Para obter mais facilmente as colunas de uma matriz você pode usar a função pré-definida `transpose`.

11. Agora você vai tentar um exercício mais difícil: escrever uma função que retorna a inversa de uma matriz dada. A solução envolve os seguintes passos:
- (a) Converter as entradas da matriz para valores de tipo `Double` ou `Rational`, para permitir usar esses valores como argumentos do operador de divisão (`/`).
  - (b) Definir uma função que obtém o determinante de uma matriz. Isso vai indicar se a matriz possui inversa ou não.
  - (c) Finalmente, definir uma função para obter a inversa.

Existem diversos algoritmos para computar o determinante e a inversa de uma matriz. Veja, por exemplo:

<http://mathworld.wolfram.com/MatrixInverse.html> ou  
[http://en.wikipedia.org/wiki/Invertible\\_matrix](http://en.wikipedia.org/wiki/Invertible_matrix)

Tente implementar uma solução para esse problema e teste sua solução de maneira adequada.