

# Funções de ordem superior

Programação Funcional

Prof. Rodrigo Ribeiro

# Setup

```
module Aula07 where

import Data.Char (isLower)

main :: IO ()
main = return ()
```

# Objetivos

- ▶ Definir e usar funções de ordem superior para abstrair padrões de recursividade.
- ▶ Utilizar funções anônimas.
- ▶ Entender composição de funções.

# Função map

# Tarefa

- ▶ Desenvolver uma função que dobra todo número presente em uma lista de inteiros.
- ▶ Como implementar essa função em Haskell?

# Função doubleList

1. Definindo o tipo.

```
doubleList :: [Int] -> [Int]
```

# Função doubleList

## 2. Definindo os casos.

```
doubleList :: [Int] -> [Int]
doubleList [] = _
doubleList (x : xs) = _
```

# Função doubleList

## 3. Definindo o caso base.

```
doubleList :: [Int] -> [Int]
doubleList [] = []
doubleList (x : xs) = _
```



# Função doubleList

4. Definindo o caso recursivo.

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

# Tarefa

- ▶ Definir uma função que realiza a negação de todos os elementos de uma lista de booleanos.
- ▶ Como implementar essa função em Haskell?

# Função notList

## 1. Definindo o tipo

```
notList :: [Bool] -> [Bool]
```

# Função notList

## 2. Definindo os casos

```
notList :: [Bool] -> [Bool]
notList [] = _
notList (x : xs) = _
```

# Função notList

## 3. Definindo o caso base

```
notList :: [Bool] -> [Bool]
notList [] = []
notList (x : xs) = _
```

# Função notList

## 4. Definindo o caso recursivo

```
notList :: [Bool] -> [Bool]
notList [] = []
notList (x : xs) = not x : notList xs
```

## Padrão

- ▶ As definições de `doubleList` e `notList` são muito similares.

```
doubleList :: [Int] -> [Int]
```

```
doubleList [] = []
```

```
doubleList (x : xs) = 2 * x : doubleList xs
```

```
notList :: [Bool] -> [Bool]
```

```
notList [] = []
```

```
notList (x : xs) = not x : notList xs
```

# Refatorar

- ▶ Ao nos depararmos com código similar, devemos utilizar refatoração.
- ▶ Extrair “o que há de comum” e reutilizar.
- ▶ Pergunta: O que há de comum?



## Casos base

- Note que em ambos os casos base, retorna-se uma lista vazia.

```
doubleList :: [Int] -> [Int]
```

```
doubleList [] = []
```

```
doubleList (x : xs) = 2 * x : doubleList xs
```

```
notList :: [Bool] -> [Bool]
```

```
notList [] = []
```

```
notList (x : xs) = not x : notList xs
```

## Caso recursivo

- ▶ No caso recursivo ambas as funções são chamadas recursivamente sobre a cauda.
- ▶ Porém, cada uma aplica uma função diferente sobre a cabeça da lista...

```
doubleList (x : xs) = 2 * x : doubleList xs
```

```
notList (x : xs) = not x : notList xs
```

## Caso recursivo

- ▶ Uma possível generalização:

```
map :: [a] -> [b]
```

```
map [] = []
```

```
map (x : xs) = ? : map xs
```

- ▶ Pergunta: o que deve preencher a ? ?

## Caso recursivo

- ▶ Como as funções `doubleList` e `notList` usam funções diferentes, podemos implementá-las usando a função `map` se ...
  - ▶ Passarmos como parâmetro a função a ser aplicada a cada elemento da lista.

# Função map

- ▶ Definição da função map:
  - ▶ Essa é uma função de ordem superior

```
map :: (a -> b) -> [a] -> [b]
```

```
map _ [] = []
```

```
map f (x : xs) = f x : map f xs
```

## Refatorando

- Definindo doubleList e sumList usando map.

```
doubleList :: [Int] -> [Int]
doubleList xs = map double xs
  where
    double x = 2 * x
```

```
notList :: [Bool] -> [Bool]
notList xs = map not xs
```

# Recapitulando

- ▶ A função `map` expressa o padrão de computação de aplicar uma função sobre cada elemento de uma lista.

## Exercícios

- ▶ Redefina a função `map` usando list comprehensions, ao invés de recursividade.
- ▶ Considera a função `even :: Int -> Bool` que determina se um inteiro é ou não par. Usando-a defina a função `evenList :: [Int] -> [Bool]` de forma que:

```
evenList [1, 2, 3] == [False, True, False]
```



## Função filter

# Tarefa

- ▶ Definir uma função que, a partir de uma string, retorna todos os caracteres minúsculos nela contidos.
- ▶ Como definir essa função em Haskell?

# Função lowers

## 1. Definindo o tipo

```
lowers :: String -> String
```

# Função lowers

## 2. Definindo os casos

- Lembre-se `String = [Char]`

```
lowers :: String -> String
```

```
lowers [] = _
```

```
lowers (x : xs) = _
```

# Função lowers

## 3. Definindo o caso base

```
lowers :: String -> String  
lowers [] = []  
lowers (x : xs) = _
```

# Função lowers

## 4. Definindo o caso recursivo

```
lowers :: String -> String
lowers [] = []
lowers (x : xs)
  | isLower x = x : lowers xs
  | otherwise = lowers xs
```

# Tarefa

- ▶ Implementar uma função que retorne todos os números pares presentes em uma lista de inteiros.

# Função evens

## 1. Definindo o tipo

```
evens :: [Int] -> [Int]
```



# Função evens

## 2. Definindo os casos

```
evens :: [Int] -> [Int]
evens [] = _
evens (x : xs) = _
```

# Função evens

## 3. Definindo o caso base.

```
evens :: [Int] -> [Int]
evens [] = []
evens (x : xs) = _
```

# Função evens

## 4. Definindo o caso recursivo.

```
evens :: [Int] -> [Int]
evens [] = []
evens (x : xs)
  | even x = x : evens xs
  | otherwise = evens xs
```

# Similaridade

- ▶ lowers e evens são bem similares
- ▶ Pergunta: Qual o padrão de computação envolvido?

```
lowers [] = []  
lowers (x : xs)  
  | isLower x = x : lowers xs  
  | otherwise = lowers xs
```

```
evens [] = []  
evens (x : xs)  
  | even x = x : evens xs  
  | otherwise = evens xs
```

## Caso base

- ▶ Ambas as funções retornam a lista vazia, quando sua entrada também é a lista vazia.

```
lowers [] = []  
lowers (x : xs)  
  | isLower x = x : lowers xs  
  | otherwise = lowers xs
```

```
evens [] = []  
evens (x : xs)  
  | even x = x : evens xs  
  | otherwise = evens xs
```

## Caso recursivo

- ▶ Inclui-se a cabeça da lista no resultado caso ela satisfaça a propriedade em questão: ser par ou ser uma letra minúscula.

```
lowers [] = []  
lowers (x : xs)  
  | isLower x = x : lowers xs  
  | otherwise = lowers xs
```

```
evens [] = []  
evens (x : xs)  
  | even x = x : evens xs  
  | otherwise = evens xs
```

## Generalizando

- ▶ Como no caso de `map`, a função que generaliza `lowsers` e `evens` também deve receber uma função (a condição de teste) como parâmetro.

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

# Refatorando

```
lowers :: String -> String  
lowers xs = filter isLower xs
```

```
evens :: [Int] -> [Int]  
evens xs = filter even xs
```



# Definições Locais

- ▶ Blocos where podem ser utilizados para código local.
  - ▶ Problema: muito código para algo tão simples...

```
doubleList :: [Int] -> [Int]
doubleList xs = map double xs
  where
    double x = 2 * x
```

# Funções Anônimas

- ▶ Haskell permite definirmos funções sem atribuir a elas um nome.

```
\ arguments -> code
```

- ▶ Redefinindo doubleList:

```
doubleList :: [Int] -> [Int]  
doubleList xs = map (\ x -> 2 * x) xs
```

# Funções Anônimas

- ▶ A expressão  $\lambda x \rightarrow 2 * x$  é uma função anônima.
  - ▶ O símbolo  $\lambda$  representa a letra grega  $\lambda$ .
- ▶ Nota histórica: A fundamentação teórica das linguagens funcionais é chamada de  $\lambda$ -cálculo e foi criada por volta de 1930 por Alonzo Church.

# Funções Anônimas

- ▶ Expressões  $\lambda$  possuem tipos funcionais.

```
Prelude*> :t \ x -> x ++ "!"  
\ x -> x ++ "!" :: String -> String
```

- ▶ Podem ser utilizadas em qualquer lugar em que espera-se uma função.

```
Prelude*> (\ x -> x ++ "!") "Hello"  
"Hello!"
```

# Funções Anônimas

- ▶ Podem ser utilizadas para definir funções.

```
double = \ x -> 2 * x
```

# Exercícios

- ▶ Implemente a função `filter` utilizando list comprehensions ao invés de recursividade.
- ▶ Implemente uma função para selecionar todos os elementos de uma lista de números inteiros que estão em um certo intervalo numérico fornecido como parâmetro.
  - ▶ Apresente uma implementação usando `filter`.
  - ▶ Apresente uma implementação usando recursão.

## Funções foldr e foldl

# Funções retornam funções

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f = \ y x -> f x y
```

- ▶ flip cria uma função a partir de outra.
- ▶ Exemplo de uso:

```
Prelude*> flip map [1,2,3] (\ x -> 2 * x)
[2,4,6]
```



# Currying

- ▶ Funções recebem um argumento por vez.

```
map :: (a -> b) -> [a] -> [b]
```

```
map :: (a -> b) -> ([a] -> [b])
```

- ▶ Funções de 2 argumentos são, na verdade, funções de um argumento que retornam funções de um argumento.

## Várias maneiras...

```
addThree :: Int -> Int -> Int -> Int
```

```
addThree x y z = x + y + z
```

```
addThree x y = \ z -> x + y + z
```

```
addThree x = \ y z -> x + y + z
```

```
addThree = \ x y z -> x + y + z
```

## Aplicação parcial

- ▶ Se funções recebem um argumento por vez, podemos omitir alguns e retornar outra função.
  - ▶ Chamamos isso de **aplicação parcial**.

```
Prelude*> :t filter isLower :: String -> String
```

# Aplicação parcial

► Exemplo:

```
doubleList :: [Int] -> [Int]  
doubleList = map (\ x -> 2 * x)
```

# Seções

- ▶ Aplicação parcial de operadores.

$(x \#) = \backslash y \rightarrow x \# y$

$(\# y) = \backslash x \rightarrow x \# y$

- ▶ Exemplos

```
doubleList = map (2 *)  
largerThan10 = filter (> 10)
```

# Seções

- ▶ Ordem importa em seções.
- ▶ Exemplos

```
Prelude*> filter (> 10) [1 .. 20]  
[11,12,13,14,15,16,17,18,19,20]
```

```
Prelude*> filter (10 >) [1 .. 20]  
[1,2,3,4,5,6,7,8,9]
```

# Tarefa

- ▶ Implemente a função

`applyAll :: [a -> a] -> a -> a`

- ▶ Que aplica cada função em uma lista a um valor inicial.

# Tarefa

- ▶ Defina os casos:

```
applyAll :: [a -> a] -> a -> a  
applyAll [] x = _  
applyAll (f : fs) = _
```



# Tarefa

- ▶ Defina o caso base:

```
applyAll :: [a -> a] -> a -> a  
applyAll [] x = x  
applyAll (f : fs) = _
```

# Tarefa

- ▶ Defina o caso recursivo:

```
applyAll :: [a -> a] -> a -> a
```

```
applyAll [] x = x
```

```
applyAll (f : fs) x = applyAll fs (f x)
```

# Composição

$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$   
 $g . f = \lambda x \rightarrow g (f x)$

# Exemplos

```
not :: Bool -> Bool
```

```
even :: Int -> Bool
```

```
odd = not . even
```

```
filterNot :: (a -> Bool) -> [a] -> [a]
```

```
filterNot p = filter (not . p)
```

# Exemplos

- Função `maxAverage`: escrita em *point-free* style.

```
maxAverage :: [[Float]] -> Float
maxAverage = maximum . map average . filter (not . null)
  where
    average xs = sum xs / fromIntegral (length xs)
```

# Tarefa

- ▶ Definir `applyAll` usando point-free style.
  - ▶ Dica: `id x = x`

```
applyAll :: [a -> a] -> a -> a
```

```
applyAll [] x = x
```

```
applyAll (f : fs) = applyAll fs (f x)
```

# Tarefa

► Caso base.

```
applyAll :: [a -> a] -> a -> a
```

```
applyAll [] = id
```

```
applyAll (f : fs) = _
```

# Tarefa

► Caso recursivo.

```
applyAll :: [a -> a] -> a -> a
```

```
applyAll [] = id
```

```
applyAll (f : fs) = applyAll fs . f
```



## Exercício

- ▶ Defina uma função que, a partir de uma lista de inteiros, retorne a soma dos quadrados de seus elementos ímpares.
  - ▶ Defina essa função usando point-free style.

# Similaridades

## ► Repetição de código

```
sum [] = 0
```

```
sum (x : xs) = x + sum xs
```

```
concat [] = []
```

```
concat (xs : xss) = xs ++ concat xss
```

```
and [] = True
```

```
and (x : xs) = x && and xs
```

# Padrão

- ▶ Funções anteriores exibem o seguinte padrão.
  - ▶ Caso de lista vazia: retornam um valor padrão.
  - ▶ Lista não vazia: utilizam uma função para combinar a cabeça com o resultado da chamada recursiva sobre a cauda.

# Padrão

- ▶ Vamos abstrair:
  - ▶ Valor retornado para lista vazia
  - ▶ Função para combinar a cabeça e resultado da chamada recursiva.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

```
foldr _ v [] = v
```

```
foldr f v (x : xs) = x `f` foldr f v xs
```

# Refatorando

```
sum = foldr (+) 0
```

```
concat = foldr (++) []
```

```
and = foldr (&&) True
```

foldr is for right

```
foldr (+) 0 (x : y : z : []) =  
x + foldr (+) 0 (y : z : []) =  
x + (y + foldr (+) 0 (z : [])) =  
x + (y + (z + foldr (+) 0 [])) =  
x + (y + (z + 0))
```

## foldr e listas

- ▶ A função `foldr` segue a estrutura de listas.
  - ▶ A lista vazia é substituída por 0
  - ▶ `(:)` é substituído por `(+)`

```
foldr (+) 0 (x : y : z : [])  
           |   |   |  
           (x + (y + (z + 0)))
```

# Funções como foldr

- ▶ Passos para refatorar recursão usando foldr.
  - ▶ Toda definição possui o seguinte template.
  - ▶ função step combina a cabeça com resultado da chamada recursiva.

```
foo = foldr step v
```



## length

- Implementando length usando foldr.

```
length :: [a] -> Int
length [] = 0
length (_ : xs) = 1 + length xs
```

# length

► Versão final

```
length :: [a] -> Int
length = foldr step 0
  where
    step _ ac = 1 + ac
```

## Implementando map

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

## Implementando map

```
map :: (a -> b) -> [a] -> [b]
map f = foldr step []
  where
    step x ac = _
```

## Implementando map

```
map :: (a -> b) -> [a] -> [b]
map f = foldr step []
  where
    step x ac = f x : ac
```

## Função reverse

```
reverse :: [a] -> [a]
reverse = foldr step []
  where
    step x ac = ac ++ [x]
```

# Função reverse

- ▶ Problema da definição anterior: `reverse` usa a concatenação e, por isso, é da ordem de  $O(n^2)$ .
- ▶ Como melhorar isso?

## Função reverse

- ▶ Uso do `foldr` faz com que as chamadas se aninhem à direita.
- ▶ Seria possível fazer um `fold` que aninha chamadas à esquerda?



## Fold left

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

## Exemplo

```
foldl (+) 0 (x : (y : (z : []))) =  
foldl (+) (0 + x) (y : (z : [])) =  
foldl (+) ((0 + x) + y) (z : []) =  
foldl (+) (((0 + x) + y) + z) [] =  
((0 + x) + y) + z
```

## Função reverse

- ▶ Usando `foldl` obtemos complexidade da ordem de  $O(n)$ .

```
reverse = foldl step []  
  where  
    step ac x = x : ac
```

## Exemplo

- Implemente a função `filter` usando `foldr`.

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

## Exemplo

- Implemente a função `filter` usando `foldr`.

```
filter :: (a -> Bool) -> [a] -> [a]
```

```
filter p xs = foldr step []
```

```
  where
```

```
    step x ac = if p x then x : ac else ac
```

## Exercício

- Implemente a concatenação de listas usando foldr.

$(++) :: [a] \rightarrow [a] \rightarrow [a]$

$[] ++ ys = ys$

$(x : xs) ++ ys = x : (xs ++ ys)$

## Exercício

- ▶ A função `takeWhile` é definida como:

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

Implemente `takeWhile` usando `foldr`.

## Exercício

- ▶ A função `all` determina se todos os elementos de uma lista satisfazem um predicado. Seu tipo é:

```
all :: (a -> Bool) -> [a] -> Bool
```

- ▶ Implemente a função `all`: 1) usando recursão e 2) usando `foldr`.



## Exercício

- ▶ Implemente a função `concatMap` de tipo:

`concatMap :: (a -> [b]) -> [a] -> [b]`

- ▶ Usando `concatMap`, reimplente a função `encode`.

## Exercício

- Uma implementação óbvia de `concatMap` é:

```
concatMap :: (a -> [b]) -> [a] -> [b]  
concatMap f = concat . map f
```

## Exercício

Porém, é possível expressar essa função da seguinte forma

```
concatMap f = foldr step []  
    where ...
```

Apresente a implementação de `concatMap` utilizando `foldr`.