

# Programação Funcional

Funções de alta ordem

**Prof. Edson Alves**

Faculdade UnB Gama

2020

# Sumário

1. Funções de alta ordem
2. Mapas, filtros e reduções

## Exemplos de funções das bibliotecas do Haskell

1. A função `lines` recebe uma string e retorna um vetor de strings, o qual corresponde às linhas contidas na string original

```
ghci> :type lines
lines :: String -> [String]

ghci> lines "Hello\nWorld"
["Hello", "World"]
```

2. A função `unlines` é sua inversa: ela recebe um vetor de strings, e une todas elas em uma única string, adicionando o terminador de linha (`'\n '`) entre elas

```
ghci> :type unlines
unlines :: [String] -> String

ghci> unlines ["a", "b", "c"]
"a\nb\nc\n"
```

3. A função `last` retorna o último elemento da lista

```
ghci> last "ABC"
'C'
```

## Exemplos de funções das bibliotecas do Haskell

4. A função complementar de `last` é a função `init`, que retorna todos, menos o último, elementos da lista

```
ghci> init "ABCDE"  
"ABCD"
```

5. A função `(++)` une duas listas em uma única lista

```
ghci> [1..5] ++ [2..4]  
[1, 2, 3, 4, 5, 2, 3, 4]
```

6. A função `concat` generaliza este comportamento, recebendo uma lista de listas e as concatenando em uma única lista

```
ghci> :type concat  
concat :: [[a]] -> [a]
```

```
ghci> concat ["um", "dois", "tres"]  
"umdoistres"
```

## Exemplos de funções das bibliotecas do Haskell

7. A função `reverse` recebe uma lista `xs` e retorna uma nova lista, com todos os elementos de `xs` em ordem inversa

```
ghci> reverse [1..5]
[5, 4, 3, 2, 1]
```

8. As funções `and` e `or` aplicam as operações lógicas binárias (`&&`) e (`||`) em todos os elementos da lista, até que reste apenas um elemento

```
ghci> and [True, False, True]
False
```

```
ghci> or [True, False, True]
True
```

9. A função `splitAt` recebe um inteiro `i` e uma lista `xs`, e retorna um par de listas (`xs[1..i]`, `xs[(i+1)..i]`)

## Exemplos de funções das bibliotecas do Haskell

10. A função `zip` recebe duas listas `xs` e `ys` e gera uma lista de pares `zs`, cujo tamanho é mesmo da menor dentre as duas, cujos elementos  $(x_i, y_i)$  são oriundos destas listas, nesta ordem

```
ghci> :type zip
zip :: [a] -> [b] -> [(a, b)]
```

```
ghci> zip [1..] "Teste"
[(1, 'T'), (2, 'e'), (3, 's'), (4, 't'), (5, 'e')]
```

11. As funções `zip3`, `zip4`, ..., `zip7` são as equivalentes para três, quatro, etc, até sete listas
12. A função `words` quebra uma string em uma lista de palavras, delimitadas por qualquer caractere que corresponda a espaços em branco:

```
ghci> :type words
words :: String -> [String]
```

```
ghci> words "A B\tC\nD\rE"
["A", "B", "C", "D", "E"]
```

## Funções infixadas

- ▶ Haskell utiliza, por padrão, a notação prefixada, de modo que, na aplicação da função `f` aos argumentos `x` e `y`, o nome da função precede os argumentos, que são separados por espaços em branco  
$$z = f\ x\ y$$
- ▶ Se a função recebe dois ou mais argumentos, é possível que a notação infixada traga uma melhor compreensão e leitura
- ▶ Para utilizar a notação infixada, basta colocar o nome da função entre crases (```), tanto em uma definição quanto em uma chamada
- ▶ Ambas formas são intercambiáveis

```
import Data.Bits
```

```
bitwise_or :: Int -> Int -> Int
```

```
bitwise_or a b = a .|. b
```

```
main = print (x, y) where           -- saída: (3, 7)
```

```
    x = bitwise_or 1 2
```

```
    y = 3 `bitwise_or` 5
```

## Exemplos de funções infixadas

1. A função `elem` recebe um elemento `x` e uma lista de elementos `xs` e retorna verdadeiro se `x` pertence a `xs`

```
ghci> :type elem  
elem :: a -> [a] -> Bool
```

```
ghci> 'x' `elem` "Teste"  
False
```

2. A negação de `elem` é a função `notElem`

```
ghci> 'x' `notElem` "Teste"  
True
```

3. A função `isPrefixOf` do módulo `Data.List` recebe os mesmos parâmetros, e retorna verdadeiro se `x` é prefixo de `xs`
4. As funções `isInfixOf` e `isSuffixOf` do mesmo módulo tem comportamento semelhante, retornando verdadeiro se `x` é uma sublista de `xs` ou se `x` é sufixo de `xs`, respectivamente



## Funções de alta ordem

- ▶ Uma função é dita de **alta ordem** se ela recebe uma ou mais funções como parâmetro ou retorna uma função
- ▶ Por exemplo, a função **break** recebe um predicado **P** e uma lista **xs**, e retorna uma par de listas (**ys**, **zs**), onde **xs = ys ++ zs** e **zs** tem início no primeiro elemento **x** de **xs** tal que a expressão '**P x**' é verdadeira

```
ghci> :type break
break :: (a -> Bool) -> [a] -> ([a], [a])
```

```
ghci> break even [1, 1, 2, 3, 5, 8]
([1, 1], [2, 3, 5, 8])
```

- ▶ A função **all** recebe um predicado **P** e uma lista **xs** e retorna verdadeiro se '**P x**' é verdadeira para todos **x** em **xs**
- ▶ A função **any** recebe os mesmos parâmetros, e retorna verdadeiro se '**P x**' é verdadeira para ao menos um elemento de **xs**

## Exemplos de funções de alta ordem

1. A função `takeWhile` recebe um predicado `P` e uma lista `xs` e retorna uma lista `ys` cujos elementos são todos dentre os primeiros elementos `x` de `xs` tais que '`P x`' é verdadeira
2. Sua complementar é a função `dropWhile`, que recebe os mesmo parâmetros e retorna uma lista `ys` cujo primeiro elemento é o primeiro elemento `x` de `xs` para o qual a expressão '`P x`' é falsa

```
Prelude Data.Char> takeWhile isUpper "FGAmaDF"  
"FGA"
```

```
Prelude Data.Char> dropWhile isUpper "FGAmaDF"  
"maDF"
```

3. A função `span` retornam um par de listas com as duas partes resultantes da chamada de `takeWhile`

```
Prelude Data.Char> span isUpper "FGAmaDF"  
("FGA", "maDF")
```

# Laços em Haskell

- ▶ Diferentemente das linguagens imperativas, Haskell não oferece construtos equivalentes aos laços **for** e **while** das linguagens imperativas
- ▶ Para contornar este fato pode-se valer de algumas técnicas distintas
- ▶ Uma maneira é utilizar recursão
- ▶ Outra forma é utilizar funções de alta ordem e abstrações
- ▶ Esta diferença de abordagem tende a ser um fator que dificulta a aprendizagem de Haskell, e linguagens funcionais em geral, para programadores acostumados com linguagens imperativas

## Exemplo de uso de recursão: Capitalização

- ▶ O primeiro exemplo de uso de recursão para substituir laços é o problema de capitalizar as palavras de uma string dada
- ▶ Por capitalizar uma palavra entende-se:
  1. tornar a primeira letra maiúscula; e
  2. transformar todas as demais em minúsculas
- ▶ Para ilustrar as diferenças entre as abordagens imperativa e funcional, será apresentado um código C++ que capitaliza strings
- ▶ Em seguida, será apresentado um código equivalente em Haskell, utilizando recursão em substituição ao laço
- ▶ Para focar apenas no processo de capitalização, o parâmetro das funções será uma lista de palavras, de modo a abstrair o processo de tokenização

# Implementação da capitalização em C++

```
5 vector<string> capitalize(const vector<string>& xs)
6 {
7     vector<string> ys;
8
9     for (auto x : xs)
10    {
11        auto y = x;
12
13        if (not y.empty())
14        {
15            y[0] = toupper(x[0]);
16
17            for (size_t i = 1; i <= x.size(); ++i)
18                y[i] = tolower(x[i]);
19        }
20
21        ys.push_back(y);
22    }
23
24    return ys;
25 }
```

# Implementação da capitalização em Haskell

```
1 import Data.Char
2
3 capitalize [] = []
4 capitalize (x:xs) = cap x : capitalize xs where
5     cap [] = []
6     cap (y:ys) = toUpper y : lower ys
7     lower [] = []
8     lower (z:zs) = toLower z : lower zs
9
10 main = putStr $ unlines $ capitalize xs where
11     xs = ["abc", "XYZ", "Teste", "iPod"]
```

## Exemplo de uso de recursão: Lista de Aprovados

- ▶ Um segundo exemplo de uso de recursão em substituição a laços é o de gerar uma lista de aprovados, a partir de uma lista de alunos e suas menções
- ▶ O critério de aprovação é ter nota final igual ou superior a 5 pontos
- ▶ Os alunos serão representados por uma estrutura que contém o nome do aluno e sua nota final
- ▶ Novamente será apresentado um código em C++, que utiliza laços
- ▶ O código em Haskell novamente substituirá o laço por recursão
- ▶ Atente que neste exemplo, e no anterior, a recursão é composta por um (ou mais) caso(s) base(s), e uma chamada recursiva

# Implementação da lista de aprovados em C++

```
5 struct Student
6 {
7     string name;
8     int score;
9 };
10
11 vector<string> aprovados(const vector<Student>& xs)
12 {
13     vector<string> ys;
14
15     for (auto [name, score] : xs)
16         if (score >= 5)
17             ys.push_back(name);
18
19     return ys;
20 }
```



# Implementação da lista de aprovados em Haskell

```
1 data Student = Student String Int
2
3 aprovados [] = []
4 aprovados (x:xs) | score >= 5 = name : aprovados xs
5                     | otherwise = aprovados xs
6                     where (Student name score) = x
7
8 main = putStr $ unlines $ aprovados xs where
9     xs = [ Student "Ana" 8, Student "Beto" 3, Student "Carlos" 5,
10           Student "Daniel" 4, Student "Edgar" 7 ]
```

## Exemplo de uso de recursão: Produto Escalar

- ▶ O terceiro exemplo de uso de recursão em substituição aos laços é o cálculo do produto escalar entre dois vetores
- ▶ Segundo a Álgebra Linear, dados dois vetores  $\vec{u}, \vec{v} \in \mathbb{R}^n$ , o produto escalar entre  $\vec{u}$  e  $\vec{v}$  é dado por

$$\vec{u} \cdot \vec{v} = \sum_{i=1}^n u_i v_i$$

- ▶ Observe que a própria definição sugere o uso de um laço, representado pelo somatório
- ▶ Para utilizar a recursão, é preciso reinterpretar esta solução
- ▶ No caso base, se  $\vec{u}, \vec{v} \in \mathbb{R}^0$ , então  $\vec{u} \cdot \vec{v} = 0$
- ▶ Se  $\vec{u}, \vec{v} \in \mathbb{R}^n$ , então

$$\vec{u} \cdot \vec{v} = u_1 v_1 + \vec{r} \cdot \vec{s},$$

onde  $\vec{r} = (u_2, u_3, \dots, u_n)$  e  $\vec{s} = (v_2, v_3, \dots, v_n)$

# Implementação do produto interno em C++

```
5 double dot_product(const vector<double>& xs, const vector<double>& ys)
6 {
7     auto res = 0.0;
8
9     for (size_t i = 0; i < xs.size(); ++i)
10         res += xs[i] * ys[i];
11
12     return res;
13 }
```

# Implementação do produto interno em Haskell

```
1 dot_product [] [] = 0.0
2 dot_product (x:xs) (y:ys) = x*y + dot_product xs ys
3
4 main = print $ dot_product xs ys where
5     xs = [ 1.2, -0.8, 5.5, 3.7 ]
6     ys = [ 2.8, 1.3, -4.9, 5.0 ]
```

## Exemplo de uso de recursão: Verificação de primalidade

- ▶ O último exemplo de uso de recursão para substituir laços é o teste de primalidade
- ▶ Dado um inteiro positivo  $n$ , a função `is_prime(n)` deve retornar verdadeiro se  $n$  é primo, e falso, caso contrário
- ▶ A complexidade da função que será apresentada é  $O(\sqrt{n})$ , pois se vale do fato de que, se  $n$  é composto, ele tem ao menos um divisor próprio  $d$  tal que  $d \leq \sqrt{n}$
- ▶ Contudo, para evitar erros de precisão, a função `sqrt` não é utilizada explicitamente

# Implementação da verificação de primalidade em C++

```
5 bool is_prime(int n)
6 {
7     if (n < 2)
8         return false;
9
10    if (n == 2)
11        return true;
12
13    if (n % 2 == 0)
14        return false;
15
16    for (int d = 3; d * d <= n; d += 2)
17        if (n % d == 0)
18            return false;
19
20    return true;
21 }
```

# Implementação da verificação de primalidade em Haskell

```
1 is_prime n | n < 2          = False
2              | n == 2        = True
3              | mod n 2 == 0   = False
4              | otherwise      = test n 3
5      where test n d | d * d > n      = True
6                      | mod n d == 0   = False
7                      | otherwise      = test n (d + 2)
8
9 main = do
10     n <- readLn :: IO Int
11     putStrLn $ show n ++ " eh primo? " ++ show (is_prime n)
```

# Mapas, filtros e reduções

- ▶ Os **mapas**, os **filtros** e as **reduções** são funções de alta ordem fundamentais em programação funcional
- ▶ Elas abstraem três conceitos fundamentais:
  1. A partir de uma lista **xs**, criar uma nova lista **ys** tal que  $y_i = f(x_i)$  para uma função  $f$  dada (mapa)
  2. A partir de uma lista **xs**, criar uma nova lista **ys** formada pelos elementos **x** de **xs** que atendem a um predicado **P** (filtro)
  3. Gerar um elemento **y** a partir de uma lista **xs** através da aplicação sucessiva de uma operação binária **op** e um valor inicial **x0** (redução)
- ▶ Todas as três técnicas recebem uma função como parâmetro
- ▶ A aplicação destas técnicas substituem, em vários casos, a necessidade dos laços das linguagens imperativas



# Mapas

- ▶ Em Haskell, os mapas são implementados por meio da função `map`

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

- ▶ Um mapa recebe uma função `f` que transforma um elemento do tipo `a` em um elemento do tipo `b` e uma lista de elementos do tipo `a`
- ▶ O retorno é uma lista de elementos do tipo `b`, onde  $b_i = f(a_i)$
- ▶ O uso de mapas simplifica o código e o torna mais legível

```
-- Capitalização utilizando mapas
```

```
import Data.Char
```

```
capitalize xs = map cap xs where
  cap [] = []
  cap (y:ys) = toUpper y : map toLower ys
```

# Filtros

- ▶ Em Haskell, os filtros são implementados por meio da função `filter`:

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

- ▶ Um filtro recebe um predicato **P** e uma lista de elementos do tipo `[a]` e retorna uma nova lista do tipo `[a]`
- ▶ Um elemento `a` da lista de entrada estará na lista de saída se, e somente se, a expressão '**P** a' for verdadeira
- ▶ A ordem relativa dos elementos é preservada

```
import Data.Char
```

```
main = print (filter isHexDigit s) where
  s = "Coordenadas (20A, 38F, 40X)"
```

```
-- saída: "Cdeada20A38F40"
```

# Lista de aprovados usando filtros e mapas

```
1 -- Lista de aprovados usando filtros
2 data Student = Student {
3     studentName    :: String,
4     studentScore   :: Int
5 }
6
7 aprovados xs = map studentName (filter f xs) where
8     f x = studentScore x >= 5
9
10 main = putStr $ unlines $ aprovados xs where
11     xs = [ Student "Ana" 8, Student "Beto" 3, Student "Carlos" 5,
12           Student "Daniel" 4, Student "Edgar" 7 ]
```

## Recursão de cauda

- ▶ Uma função é dita **recursiva de cauda** (*tail recursive*) se ela ou retorna valores (casos-base) ou retorna chamadas de si mesma, com diferentes parâmetros
- ▶ Nem toda função recursiva é recursiva de cauda
- ▶ Por exemplo, a definição da função fatorial abaixo é recursiva, mas não recursiva de cauda:

$$n! = \begin{cases} 1, & \text{se } n = 0 \text{ ou } n = 1 \\ n \cdot (n-1)!, & \text{caso contrário} \end{cases}$$

- ▶ Isto porque, na chamada recursiva, o retorno consiste no produto da chamada de  $(n-1)!$  pelo parâmetro  $n$

# Recursão de cauda

- ▶ Contudo, esta definição pode ser modificada para que se torne recursiva de cauda:

$$f(n, m) = \begin{cases} m, & \text{se } n = 0 \text{ ou } n = 1 \\ f(n - 1, n \cdot m), & \text{caso contrário} \end{cases}$$

- ▶ Desde modo,  $n! = f(n, 1)$
- ▶ Observe que a chamada recursiva agora consiste apenas em uma invocação da função  $f$
- ▶ O parâmetro  $m$  é denominado **acumulador**
- ▶ A recursão de cauda permite a **otimização de chamada de cauda** (*tail call optimization* – TCO)
- ▶ Isto porque, neste caso, é possível evitar o uso da pilha de execução, reaproveitando um único registro de ativação a cada chamada

# Referências

1. **SHALOM**, Elad. *A Review of Programming Paradigms Throughout the History – With a Suggestion Toward a Future Approach*, Amazon, 2019.
2. **SULLIVAN**, Bryan O.; **GOERZEN**, John; **STEWART**, Don. *Real World Haskell*, O'Reilly.