

# Introdução a Programação Funcional com Haskell

Aplicações em data science e na web

Dia 01

---

Fabrício Olivetti e Emilio Francesquini  
[folivetti@ufabc.edu.br](mailto:folivetti@ufabc.edu.br) e [e.francesquini@ufabc.edu.br](mailto:e.francesquini@ufabc.edu.br)

28/09/2019

Centro de Matemática, Computação e Cognição  
Universidade Federal do ABC



- Estes slides foram preparados para o curso de **Programação funcional em Haskell no mundo real na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.



# Introdução

---

Muitos cursos de Computação e Engenharia introduzem programação com paradigma imperativo e estruturado.

Exemplo clássico da receita de bolo (que não é a melhor forma de descrever o conceito de algoritmo).

Muitas das linguagens de programação são, na realidade, multi-paradigmas, porém favorecendo um dos paradigmas.

---

```
1 aprovados = {}
2 for (i = 0; i < length(alunos); i++) {
3     a = alunos[i];
4     if (a.nota >= 5) {
5         adiciona(aprovados, toUpper(a.nome));
6     }
7 }
8 return sort(aprovados);
```

---

```
1  class Aprovados {
2      private ArrayList aprovados;
3      public Aprovados () {
4          aprovados = new ArraList();
5      }
6      public addAluno(aluno) {
7          if (aluno.nota >= 5) {
8              aprovados.add(aluno.nome.toUpperCase());
9          }
10     }
11     public getAprovados() {
12         return aprovados.sort();
13     }
14
15 }
```

---

```
1 sort [nome aluno | aluno <- alunos, nota aluno >= 5]
```

---



Muitas linguagens de programação estão incorporando elementos de paradigma funcional.

O Python possui alguns elementos do paradigma funcional:

```
anonima = lambda x: 3*x + 1  
par      = lambda x: x%2 == 0
```

```
map(anonima, lista)  
filter(par, lista)
```

```
def preguiçosa(x):  
    for i in range(x):  
        yield anonima(x)
```

---

```
1 public interface List<E> {  
2     void add(E x);  
3     Iterator<E> iterator();  
4 }  
5  
6 array.stream()  
7     .filter(n -> (n % 2) == 0);
```

---

# Haskell

---

- Surgiu em 1990 com o objetivo de ser a primeira linguagem puramente funcional.
- Por muito tempo considerada uma linguagem acadêmica.
- Atualmente é utilizada em diversas empresas (totalmente ou em parte de projetos).

- **Códigos concisos e declarativos:** o programador *declara* o que ele quer ao invés de escrever um passo-a-passo. Programas em Haskell chegam a ser dezenas de vezes menores que em outras linguagens.

---

```
1 take 100 [x | x <- nat, primo x]
```

---

- **Sistema de tipagem forte:** ao contrário de linguagens como *Java*, *C* e *Python*, as declarações de tipo no Haskell são simplificadas (e muitas vezes podem ser ignoradas), porém, seu sistema rigoroso permite que muitos erros comuns sejam detectados em **tempo de compilação**.

Exemplo em Java:

---

```
1 int x      = 10;  
2 double y = 5.1;  
3 System.out.println("Resultado: " + (x*y));
```

---

OK!

- **Sistema de tipagem forte:** ao contrário de linguagens como *Java*, *C* e *Python*, as declarações de tipo no Haskell são simplificadas (e muitas vezes podem ser ignoradas), porém, seu sistema rigoroso permite que muitos erros comuns sejam detectados em **tempo de compilação**.

Exemplo em Haskell:

---

```
1 x = 10  :: Int
2 y = 5.1 :: Double
3 print ("Resultado: " + (x*y) )
```

---

ERRO!



- **Compreensão de listas:** listas são frequentemente utilizadas para a solução de diversos problemas. O Haskell utiliza listas como um de seus conceitos básicos permitindo uma notação muito parecida com a notação de conjuntos na matemática.

$$xs = \{x \mid x \in \mathbb{N}, x \text{ ímpar}\}$$

---

```
1 xs = [x | x <- nat, impar x]
```

---

- **Imutabilidade:** não existe um conceito de variável, apenas nomes e declarações. Uma vez que um nome é declarado com um valor, ele não pode sofrer alterações.

---

```
1 x = 1.0
2 x = 2.0
```

---

ERRO!

- **Funções Recursivas:** com a imutabilidade, o conceito de laços de repetição também não existe em linguagens funcionais. (**Por quê?**) Eles são implementados através de funções recursivas.

---

```
1 int x = 1;
2 for (int i = 1; i <= 10; i++) {
3     x = x * 2;
4 }
5 printf("%d\n", x);
```

---

- **Funções Recursivas:** com a imutabilidade, o conceito de laços de repetição também não existe em linguagens funcionais. (**Por quê?**) Eles são implementados através de funções recursivas.

---

```
1 f 0 = 1
2 f n = 2 * f (n-1) -- Note que f(x) é o mesmo que f x
3
4 print (f 10)
```

---

- **Funções de alta ordem:** funções podem receber funções como parâmetros. Isso permite definir funções genéricas, compor duas ou mais funções e definir linguagens de domínio específicos (ex.: *parsing*).

---

```
1 print (aplique dobro [1,2,3,4])
2 > [2,4,6,8]
```

---

- **Tipos polimórficos:** permite definir funções genéricas que funcionam para classes de tipos. Por exemplo, o operador de soma `+` pode ser utilizado para qualquer tipo numérico.

---

```
1 1 + 2          -- 3
2 1.0 + 3.0     -- 4.0
3 (2%3) + (3%6) -- (7%6)
```

---

- **Avaliação preguiçosa:** ao aplicar uma função, o resultado será computado apenas quando requisitado. Isso permite evitar computações desnecessárias, estimula uma programação modular e permite estruturas de dados infinitos.

---

```
1 listaInf = [1..] -- 1, 2, 3, ...
2 print (take 10 listaInf)
```

---



- Paradigmas de programação e características básicas de Haskell
- Livros
  - ▶ [GH] 1,2
  - ▶ [SGS] 1
  - ▶ [ML] 2



# Ambiente de Programação

---

- **GHCi Haskell Compiler (GHC)**: compilador de código aberto para a linguagem Haskell.
  - ▶ Padrão de fato
  - ▶ Outros compiladores existem mas são incompletos ou têm uma equipe limitada de manutenção
- Possui um modo interativo **ghci** (similar ao **iPython**).
  - ▶ REPL - Read, Evaluate, Print, Loop

Uso recomendado de:

- **Git** - controle de revisão
- **Stack** - gerenciamento de projeto e dependências
- **Haddock** - documentação

## ■ Haskell Stack

- ▶ **ATENÇÃO!!!!** NÃO UTILIZE O APT-GET PARA INSTALAR O GHC OU O STACK!
- ▶ Para instalar o Stack no Linux :

```
1 curl -sSL https://get.haskellstack.org/ | sh
```

ou

```
1 wget -qO- https://get.haskellstack.org/ | sh
```

- ▶ Para instalar no Windows  (você quer mesmo fazer isso? 😊) faça o download do instalador no site <https://docs.haskellstack.org/>

---

```
1 > stack new primeiroProjeto simple
2 > cd primeiroProjeto
3 > stack setup
4 > stack build
5 > stack exec primeiroProjeto
```

---

- Diversos editores de texto tem suporte à edição, compilação e execução de código Haskell. Entre eles estão Emacs, Vim, Atom, Sublime e Visual Studio Code. Todos baseados no Intero, um backend para IDEs de Haskell.
- Fique a vontade para escolher o editor da sua preferência. Em seguida descrevemos as instruções para utilizar o Intero com o Atom e com o MS Visual Studio Code.

Acesse <https://code.visualstudio.com/> e baixe a versão compatível com o seu SO.

Após o download, nas máquinas com Ubuntu do laboratório:

---

```
1 sudo dpkg -i nome_do_arquivo.deb
```

---

Em seguida, precisamos instalar o Intero, hlint.

---

```
1 stack install intero hlint apply-refact
```

---

Com a instalação concluída, abra o Visual Studio Code, no canto inferior esquerdo clique na engrenagem, Extensions e instale a extensão **Haskero** e **haskell-linter**.



- Instalando o ambiente Haskell
- The Haskell Tool Stack
- Introdução à Haskell - Página 15 em diante



# Primeiro Projeto

---

Para criar projetos, utilizaremos a ferramenta **stack**. Essa ferramenta cria um ambiente isolado

---

```
1 $ stack new primeiro-projeto simple
2 $ cd primeiro-projeto
3 $ stack setup
4 $ stack build
5 $ stack exec primeiro-projeto
```

---

Os dois últimos comandos são referentes a compilação do projeto e execução.

O stack cria a seguinte estrutura de diretório:

- **LICENSE:** informação sobre a licença de uso do software.
- **README.md:** informações sobre o projeto em formato Markdown.
- **Setup.hs:** retrocompatibilidade com o sistema cabal.
- **primeiro-projeto.cabal:** informações das dependências do projeto.

- **stack.yaml**: parâmetros do projeto
- **package.yaml**: configurações de compilação e dependências de bibliotecas externas.
- **src/Main.hs**: arquivo principal do projeto.

---

```
1 module Main where    -- indica que é o módulo principal
2
3 main :: IO ()
4 main = do             -- início da função principal
5   putStrLn "hello world"  -- imprime hello world
```

---

---

```
1 $ stack ghci
2 > 2+3*4
3 14
4
5 > (2+3)*4
6 20
7
8 > sqrt (3^2 + 4^2)
9 5.0
```

---

O operador de exponenciação (^) tem precedência maior do que o de multiplicação e divisão (\*,/) que por sua vez têm maior precedência maior que a soma e subtração (+,-).

---

```
1 $ stack ghci
2 > 2+3*4^5 == 2 + (3 * (4^5))
```

---

Para saber a precedência de um operador basta digitar:

---

```
1 > :i (+)
2 class Num a where
3   (+) :: a -> a -> a
4   ...
5   -- Defined in 'GHC.Num'
6 infixl 6 +
```

---

- Pode ser utilizado para qualquer tipo numérico (`class Num`)
- Tem precedência nível 6 (quanto maior o número maior sua prioridade)
- É associativo a esquerda. Ou seja:  $1 + 2 + 3$  vai ser computado na ordem  $(1 + 2) + 3$ .



No Haskell, a aplicação de função é definida como o nome da função seguido dos parâmetros separados por espaço com a maior prioridade na aplicação da função:

---

```
1 f a b      -- f(a,b)
2 f a b + c*d -- f(a,b) + c*d
```

---

A tabela abaixo contém alguns contrastes entre a notação matemática e o Haskell:

Matemática	Haskell
$f(x)$	<code>f x</code>
$f(x, y)$	<code>f x y</code>
$f(g(x))$	<code>f (g x)</code>
$f(x, g(y))$	<code>f x (g y)</code>
$f(x)g(y)$	<code>f x * g y</code>

Editem o arquivo *Main.hs* acrescentando o seguinte conteúdo entre a declaração de módulo e a função **main**:

---

```
1 dobra x = x + x
2
3 quadruplica x = dobra (dobra x)
```

---

No GHCi:

---

```
1 > :l teste.hs
2 > quadruplica 10
3 40
```

---

O comando `:l` carrega as definições contidas em um arquivo fonte.

Acrescentem a seguinte linha no arquivo fonte:

---

```
1 fatorial n = product [1..n]
```

---

e no GHCi:

---

```
1 > :reload
2 > fatorial 5
3 120
```

---

O comando `:t` mostra o tipo da função enquanto o comando `:q` sai do ghci.

---

```
1 > :t dobra
2 dobra :: Num a => a -> a
3
4 > :q
5 $
```

---

- `:h` - Imprime a ajuda
- `:{` seguido de comandos e finalizado por `}` permite comandos com múltiplas linhas
  - ▶ Também é possível separar as linhas com `;`

---

```
1 > :{
2 | fatorial 0 = 1
3 | fatorial n = n * fatorial (n - 1)
4 | :}
5 > fatorial 5
6 120
7 > fatorial2 0 = 1 ; fatorial2 n = n * fatorial2 (n - 1)
8 > fatorial2 7
9 5040
```

---



- Funções
- [GH] 4
- [SGS] 2
- [ML] 4



# Convenções

---

- Os nomes das funções e seus argumentos devem começar com uma letra minúscula e seguida por 0 ou mais letras, maiúsculas ou minúsculas, dígitos, *underscore*, e aspas simples:

`funcao, ordenaLista, soma1, x'`

- Os únicos nomes que não podem ser utilizados são:

case, class, data, default, deriving do, else,  
foreign, if, import, in, infix, infixl, infixr,  
instance, let module, newtype, of, then, type,  
where

- As listas são nomeadas acrescentando o caractere 's' ao nome do que ela representa.
- Uma lista de números  $n$  é nomeada  $ns$ , uma lista de variáveis  $x$  se torna  $xs$ . Uma lista de listas de caracteres tem o nome  $css$ .

- O layout dos códigos em Haskell é similar ao do Python, em que os blocos lógicos são definidos pela indentação.

---

```
1 f x = a*x + b
2   where
3     a = 1
4     b = 3
5 z = f 2 + 3
```

---

- A palavra-chave **where** faz parte da definição de **f**, da mesma forma, as definições de **a** e **b** fazem parte da cláusula **where**. A definição de **z** não faz parte de **f**.

- A definição de tabulação varia de editor para editor.
- Ainda que seja o mesmo editor, a tabulação varia de usuário para usuário.
- Como o espaço é importante no Haskell, usem espaços em vez de tab.
- Use Emacs. 😊



Comentários em uma linha são demarcados pela sequência `--`, comentários em múltiplas linhas são demarcados por `{-` e `-}`:

---

```
1  -- função que dobra o valor de x
2  dobra x = x + x
3
4  {-
5  dobra recebe uma variável numérica
6  e retorna seu valor em dobro.
7  -}
```

---

## Tipos de datos

---



- Um **tipo** é uma coleção de valores relacionados entre si.

## Exemplos

- **Int** compreende todos os valores de números inteiros.
- **Bool** contém apenas os valores **True** e **False**, representando valores lógicos

- Em Haskell, os tipos são definidos pela notação

---

1 `v :: T`

---

- Significando que `v` define um valor do tipo `T`.

---

```
1 False :: Bool
2 True  :: Bool
3 10    :: Int
```

---

- O compilador GHC já vem com suporte nativo a diversos tipos básicos.
- Durante o curso veremos como definir e criar os nossos próprios tipos.

Os tipos são:

- **Bool**: contém os valores **True** e **False**. Expressões booleanas podem ser executadas com os operadores **&&** (e), **||** (ou) e **not**.
- **Char**: contém todos os caracteres no sistema **Unicode**. Podemos representar a letra 'a', o número '5', a seta tripla '≡' e o *homem de terno levitando*<sup>1</sup> '☹'.
- **String**: sequências de caracteres delimitados por aspas duplas: "Olá Mundo".

---

<sup>1</sup>Este é o nome oficial do caracter na tabela Unicode v.7.0!

- **Int**: inteiros com precisão fixa em 64 bits. Representa os valores numéricos de  $-2^{63}$  até  $2^{63} - 1$ .
- **Integer**: inteiros de precisão arbitrária. Representa valores inteiros de qualquer precisão, a memória é o limite. Mais lento do que operações com **Int**.
- **Float**: valores em ponto-flutuante de precisão simples. Permite representar números com um total de 7 dígitos, em média.
- **Double**: valores em ponto-flutuante de precisão dupla. Permite representar números com quase 16 dígitos, em média.

Note que ao escrever:

```
1 x = 3
```

O tipo de `x` pode ser `Int`, `Integer`, `Float` ou `Double`.

## Pergunta

Qual tipo devemos atribuir a `x`?

**Listas** são sequências de elementos do mesmo tipo agrupados por colchetes e separados por vírgula:

1 [1,2,3,4]



Uma lista de tipo T tem tipo [T]:

---

1	[1,2,3,4]	:: [Int]
2	[False, True, True]	:: [Bool]
3	['o', 'l', 'a']	:: [Char]

---

Também podemos ter listas de listas:

---

```
1 [ [1,2,3], [4,5] ] :: [[Int]]
2 [ [ 'o', 'l', 'a' ], [ 'm', 'u', 'n', 'd', 'o' ] ] :: [[Char]]
```

---

Notem que:

- O tipo da lista não especifica seu tamanho
- Não existem limitações quanto ao tipo da lista
- Não existem limitações quanto ao tamanho da lista

- **Tuplas** são sequências finitas de componentes, contendo zero ou mais tipos diferentes:

---

```
1 (True, False) :: (Bool, Bool)
2 (1.0, "Sim", False) :: (Double, String, Bool)
```

---

- O tipo da tupla é definido como  $(T_1, T_2, \dots, T_n)$ .

Notem que:

- O tipo da tupla especifica seu tamanho
- Não existem limitações dos tipos associados a tupla (podemos ter tuplas de tuplas)
- Tuplas **devem** ter um tamanho finito
- Tuplas de aridade 1 não são permitidas para manter compatibilidade do uso de parênteses como ordem de avaliação

- **Funções** são mapas de argumentos de um tipo para resultados em outro tipo. O tipo de uma função é escrita como  $T1 \rightarrow T2$ , ou seja, o mapa do tipo  $T1$  para o tipo  $T2$ :

---

```
1 not  :: Bool -> Bool
2 even :: Int  -> Bool
```

---

Para escrever uma função com múltiplos argumentos, basta separar os argumentos pela `->`, sendo o último o tipo de retorno:

---

```
1 soma :: Int -> Int -> Int
2 soma x y = x + y
3
4 mult :: Int -> Int -> Int -> Int
5 mult x y z = x*y*z
```

---



- Tipos e classes padrões
- Listas
- Tipos: [GH] 3; [SGS] 2; [ML] 3
- Listas: [GH] 5; [SGS] 2; [ML] 2



# Polimorfismo

---

Considere a função `length` que retorna o tamanho de uma lista. Ela deve funcionar para qualquer uma dessas listas:

- 
- 1 `[1,2,3,4] :: [Int]`
  - 2 `[False, True, True] :: [Bool]`
  - 3 `['o', 'l', 'a'] :: [Char]`
- 

## Pergunta

Qual é então o tipo de `length`?

- Qual o tipo de `length`?

---

1 `length :: [a] -> Int`

---

- Quem é `a`?

- Em Haskell, `a` é conhecida como **variável de tipo** e ela indica que a função deve funcionar para listas de qualquer tipo.
- As variáveis de tipo devem seguir a mesma convenção de nomes do Haskell, iniciar com letra minúscula. Como convenção utilizamos `a`, `b`, `c`, `...`.

- Considere agora a função (+), diferente de `length` ela pode ter um comportamento diferente para tipos diferentes.
- Internamente somar dois `Int` pode ser diferente de somar dois `Integer` (e definitivamente é diferente de somar dois `Float`).
- Ainda assim, essa função **deve** ser aplicada a tipos numéricos.

- A ideia de que uma função pode ser aplicada a apenas uma classe de tipos é explicitada pela **Restrição de classe** (*class constraint*).
- Uma restrição é escrita na forma  $C \ a$ , onde  $C$  é o nome da classe e  $a$  uma variável de tipo.

---

1  $(+) \ :: \ \text{Num} \ a \ \Rightarrow \ a \ \rightarrow \ a \ \rightarrow \ a$

---

- A função  $+$  recebe dois tipos de uma classe numérica e retorna um valor desse mesmo tipo.

- Note que nesse caso, ao especificar a entrada como **Int** para o primeiro argumento, todos os outros **devem** ser **Int** também.

---

1 (+) :: Num a => a -> a -> a

---

- Uma vez que uma função contém uma restrição de classe, pode ser necessário definir **instâncias** dessa função para diferentes tipos pertencentes à classe.
- Os valores também podem ter restrição de classe:

---

```
1 3 :: Num a => a
```

---

O que resolve nosso problema anterior.



Funções devem ser escritas em forma de expressões combinando outras funções, de tal forma a manter simplicidade:

---

```
1 impar :: Integral a => a -> Bool  
2 impar n = n `mod` 2 == 1
```

---

Funções devem ser escritas em forma de expressões combinando outras funções, de tal forma a manter simplicidade:

---

```
1 quadrado :: Num a => a -> a
2 quadrado n = n*n
```

---

Funções devem ser escritas em forma de expressões combinando outras funções, de tal forma a manter simplicidade:

---

```
1 quadradoMais6Mod9 :: Integral a => a -> a
2 quadradoMais6Mod9 n = (n*n + 6) `mod` 9
```

---

Escreva uma função que retorne a raiz de uma equação do segundo grau:

---

```
1 raiz2Grau :: Floating a => a -> a -> a -> (a, a)
2 raiz2Grau a b c = ( ???, ??? )
```

---

Teste com `raiz2Grau 4 3 (-5)` e `raiz2Grau 4 3 5`.

Para organizar nosso código, podemos utilizar a cláusula **where** para definir nomes intermediários:

---

```
1 f x = y + z
2   where
3     y = e1
4     z = e2
```

---

---

```
1 euclidiana :: Floating a => a -> a -> a
2 euclidiana x y = sqrt diffSq
3   where
4     diffSq = (x - y)^2
```

---

Reescreva a função `raiz2Grau` utilizando `where`.

A função `if-then-else` nos permite utilizar desvios condicionais em nossas funções:

---

```
1 abs :: Num a => a -> a
2 abs n = if (n >= 0) then n else (-n)
```

---

OU

---

```
1 abs :: Num a => a -> a
2 abs n = if (n >= 0)
3         then n
4         else (-n)
```

---



Também podemos encadear condicionais:

---

```
1 signum :: (Ord a, Num a) => a -> a
2 signum n = if (n == 0)
3           then 0
4           else if (n > 0)
5               then 1
6               else (-1)
```

---

Utilizando condicionais, reescreva a função `raiz2Grau` para retornar (0,0) no caso de delta negativo.

Note que a assinatura da função agora deve ser:

---

```
1 raiz2Grau :: (Ord a, Floating a) => a -> a -> a -> (a,  
  ↪ a)
```

---

Uma alternativa ao uso de `if-then-else` é o uso de *guards* (`|`) que deve ser lido como *tal que*:

---

```
1 signum :: (Ord a, Num a) => a -> a
2 signum n | n == 0      = 0  -- signum n tal que n==0
3                   -- é definido como 0
4           | n > 0      = 1
5           | otherwise = -1
```

---

`otherwise` é o caso contrário e é definido como `otherwise = True`.

Note que as expressões guardadas são avaliadas de cima para baixo, o primeiro verdadeiro será executado e o restante ignorado.

---

```
1 classificaIMC :: Double -> String
2 classificaIMC imc
3   | imc <= 18.5 = "abaixo do peso"
4   -- não preciso fazer && imc > 18.5
5   | imc <= 25.0 = "no peso correto"
6   | imc <= 30.0 = "acima do peso"
7   | otherwise  = "muito acima do peso"
```

---

Utilizando guards, reescreva a função `raiz2Grau` para retornar um erro com raízes negativas.

Para isso utilize a função `error`:

---

```
1 error "Raízes negativas."
```

---

O uso de **error** interrompe a execução do programa. Nem sempre é a melhor forma de tratar erro, aprenderemos alternativas ao longo do curso.

Considere a seguinte função:

---

```
1 not :: Bool -> Bool
2 not x = if (x == True) then False else True
```

---

Podemos reescreve-la utilizando guardas:

---

```
1 not :: Bool -> Bool
2 not x | x == True  = False
3       | x == False = True
```

---



Quando temos comparações de igualdade nos guardas, podemos definir as expressões substituindo diretamente os argumentos:

---

```
1 not :: Bool -> Bool
2 not True  = False
3 not False = True
```

---

Não precisamos enumerar todos os casos, podemos definir apenas casos especiais:

---

```
1 soma :: (Eq a, Num a) => a -> a -> a
2 soma x 0 = x
3 soma 0 y = y
4 soma x y = x + y
```

---

Assim como os guards, os padrões são avaliados do primeiro definido até o último.

Implemente a multiplicação utilizando Pattern Matching:

---

```
1 mul :: Num a => a -> a -> a
2 mul x y = x*y
```

---

Implemente a multiplicação utilizando Pattern Matching:

---

```
1 mul :: (Eq a, Num a) => a -> a -> a
2 mul 0 y = 0
3 mul x 0 = 0
4 mul x 1 = x
5 mul 1 y = y
6 mul x y = x*y
```

---

Quando a saída não depende da entrada, podemos substituir a entrada por `_` (não importa):

---

```
1 mul :: (Eq a, Num a) => a -> a -> a
2 mul 0 _ = 0
3 mul _ 0 = 0
4 mul x 1 = x
5 mul 1 y = y
6 mul x y = x*y
```

---

Como o Haskell é preguiçoso, ao identificar um padrão contendo 0 ele não avaliará o outro argumento.

---

```
1 mul :: (Eq a, Num a) => a -> a -> a
2 mul 0 _ = 0
3 mul _ 0 = 0
4 mul x 1 = x
5 mul 1 y = y
6 mul x y = x*y
```

---

As expressões lambdas, também chamadas de funções anônimas, definem uma função sem nome para uso geral:

---

```
1 -- Recebe um valor numérico e
2 -- retorna uma função que
3 -- recebe um número e retorna outro número
4 somaMultX :: Num a => a -> (a -> a)
5 somaMultX x = \y -> x + x * y
6
7 -- somaMult2 é uma função que
8 -- retorna um valor multiplicado por 2
9 somaMult2 = somaMultX 2
```

---

Para definir um operador em Haskell, podemos criar na forma infixada ou na forma de função:

---

```
1 (:+) :: Num a => a -> a -> a
2 x :+: y = abs x + y
```

---

ou

---

```
1 (:+) :: Num a => a -> a -> a
2 (:+) x y = abs x + y
```

---



Da mesma forma, uma função pode ser utilizada como operador se envolta de crases:

---

```
1 > mod 10 3
2 1
3 > 10 `mod` 3
4 1
```

---

Seendo # um operador, temos que  $(\#)$ ,  $(x \#)$ ,  $(\# y)$  são chamados de seções, e definem:

- 
- 1  $(\#) = \lambda x \rightarrow (\lambda y \rightarrow x \# y)$
  - 2  $(x \#) = \lambda y \rightarrow x \# y$
  - 3  $(\# y) = \lambda x \rightarrow x \# y$
-

Essas formas são também conhecidas como **point-free notation**:

---

```
1 > (/) 4 2
2 2
3 > (/2) 4
4 2
5 > (4/) 2
6 2
```

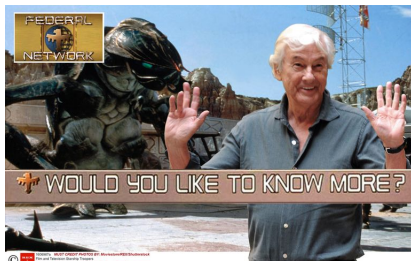
---

Considere o operador ( $\&\&\&$ ), simplique a definição para apenas dois padrões:

---

```
1 ( $\&\&\&$ ) :: Bool -> Bool -> Bool
2 True   $\&\&\&$  True  = True
3 True   $\&\&\&$  False = False
4 False  $\&\&\&$  True  = False
5 False  $\&\&\&$  False = False
```

---



- Tipos polimórficos
  - ▶ [GH] 3; [SGS] 2; [ML] 3
- Funções, casamento de padrões, guardas, lambdas
  - ▶ [GH] 4; [SGS] 2; [ML] 4
- Uma brevíssima introdução à Cálculo  $\lambda$  com Haskell
  - ▶ O combinador Y
- Syntax and Semantics of Programming Languages (Lambda Calculus, Cap. 5)

# Listas

---

- Uma das principais estruturas em linguagens funcionais.
- Representa uma coleção de valores de um determinado tipo.
- Todos os valores devem ser do **mesmo** tipo.

- **Definição recursiva:** ou é uma lista vazia ou um elemento do tipo genérico `a` concatenado com uma lista de `a`.

---

1 `data [] a = [] | a : [a]`

---

- `(:)` - operador de concatenação de elemento com lista
  - ▶ Lê-se: *cons*



Seguindo a definição anterior, a lista [1, 2, 3, 4] é representada por:

---

```
1 lista = 1 : 2 : 3 : 4 : []
```

---

É uma lista ligada!!

---

1 `lista = 1 : 2 : 3 : 4 : []`

---

A complexidade das operações são as mesmas da estrutura de lista ligada!

Existem diversos *syntax sugar* para criação de listas (ainda bem! 😊)

```
1 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Faixa de valores inclusivos:

---

1 `[1..10] == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`

---

Faixa de valores inclusivos com tamanho do passo:

---

1 `[0,2..10] == [0, 2, 4, 6, 8, 10]`

---

Lista infinita:

---

1 `[0,2..]` == `[0, 2, 4, 6, 8, 10,..]`

---

Como o Haskell permite a criação de listas infinitas?

Uma vez que a avaliação é preguiçosa, ao fazer:

---

```
1 lista = [0,2..]
```

---

ele cria apenas uma **promessa** de lista.

Efetivamente ele faz:

---

1 `lista = 0 : 2 : geraProximo`

---

sendo `geraProximo` uma função que gera o próximo elemento da lista.



- Conforme for necessário, ele gera e avalia os elementos da lista sequencialmente.
- Então a lista infinita não existe em memória, apenas uma função que gera quantos elementos você precisar dela.



- Listas
- Livros [GH] 5; [SGS] 2; [ML] 2

COFFEE BREAK (não incluso)

---

## Funções básicas para manipulação de listas

---

- O operador !! recupera o  $i$ -ésimo elemento da lista, com índice começando do 0:

---

```
1 > lista = [0..10]
2 > lista !! 2
3 2
```

---

- Note que esse operador é custoso para listas ligadas! Não abuse dele!

A função **head** retorna o primeiro elemento da lista:

---

```
1 > head [0..10]
2 0
```

---

A função `tail` retorna a lista sem o primeiro elemento (sua cauda):

---

```
1 > tail [0..10]
2 [1,2,3,4,5,6,7,8,9,10]
```

---

O que a seguinte expressão retornará?

```
1 > head (tail [0..10])
```



A função `take` retorna os `n` primeiros elementos da lista:

---

```
1 > take 3 [0..10]
2 [0,1,2]
```

---

E a função **drop** retorna a lista sem os **n** primeiros elementos:

---

```
1 > drop 6 [0..10]
2 [7,8,9,10]
```

---

Implemente o operador `!!` utilizando as funções anteriores.

Implemente o operador !! utilizando as funções anteriores.

---

```
1 xs !! n = head (drop n xs)
```

---

O tamanho da lista é dado pela função `length`:

---

```
1 > length [1..10]
2 10
```

---

- As funções **sum** e **product** retornam a somatória e produtória de uma lista:

---

```
1 > sum [1..10]
2 55
3 > product [1..10]
4 3628800
```

---

## Pergunta

Quais tipos de lista são aceitos pelas funções **sum** e **product**?

Utilizamos o operador ++ para concatenar duas listas ou o : para adicionar um valor ao começo da lista:

---

```
1 > [1..3] ++ [4..10] == [1..10]
2 True
3 > 1 : [2..10] == [1..10]
4 True
```

---

- **Atenção** à complexidade do operador ++

Implemente a função `fatorial` utilizando o que aprendemos até então.



## Pattern Matching com Listas

---

Quais padrões podemos capturar em uma lista?

Quais padrões podemos capturar em uma lista?

- Lista vazia:
  - ▶ []
- Lista com um elemento:
  - ▶ (x : []) ou [x]
- Lista com um elemento seguido de vários outros:
  - ▶ (x : xs)

E qualquer um deles pode ser substituído pelo *não importa* `_`.

Para saber se uma lista está vazia utilizamos a função `null`:

---

```
1 null :: [a] -> Bool
2 null [] = True
3 null _  = False
```

---

A função `length` pode ser implementada recursivamente da seguinte forma:

---

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

---

Implemente a função `take`. Se  $n \leq 0$  deve retornar uma lista vazia.

Assim como em outras linguagens, uma **String** no Haskell é uma lista de **Char**:

```
1 > "Ola Mundo" == ['O', 'l', 'a', ' ', 'M', 'u', 'n', 'd', 'o']
```

# Compreensão de Listas

---



Na matemática, quando falamos em conjuntos, definimos da seguinte forma:

$$\{x^2 \mid x \in \{1..5\}\}$$

que é lido como *x ao quadrado para todo x do conjunto de um a cinco*.

No Haskell podemos utilizar uma sintaxe parecida:

---

```
1 > [x^2 | x <- [1..5]]  
2 [1,4,9,16,25]
```

---

que é lido como *x ao quadrado tal que x vem da lista de valores de um a cinco.*

A expressão `x <- [1..5]` é chamada de **expressão geradora**, pois ela gera valores na sequência conforme eles forem requisitados. Outros exemplos:

---

```
1 > [toLower c | c <- "OLA MUNDO"]
2 "ola mundo"
3 > [(x, even x) | x <- [1,2,3]]
4 [(1, False), (2, True), (3, False)]
```

---

Podemos combinar mais do que um gerador e, nesse caso, geramos uma lista da combinação dos valores deles:

---

```
1 >[(x,y) | x <- [1..4], y <- [4..5]]  
2 [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5),(4,4),(4,5)]
```

---

Se invertermos a ordem dos geradores, geramos a mesma lista mas em ordem diferente:

---

```
1 > [(x,y) | y <- [4..5], x <- [1..4]]  
2 [(1,4),(2,4),(3,4),(4,4),(1,5),(2,5),(3,5),(4,5)]
```

---

Isso é equivalente a um laço **for** encadeado!

Um gerador pode depender do valor gerado pelo gerador anterior:

---

```
1 > [(i,j) | i <- [1..5], j <- [i+1..5]]
2 [(1,2),(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),
3  (3,4),(3,5),(4,5)]
```

---

Equivalente a:

---

```
1 for (i=1; i<=5; i++) {  
2     for (j=i+1; j<=5; j++) {  
3         // faça algo  
4     }  
5 }
```

---

A função `concat` transforma uma lista de listas em uma lista única concatenada (conhecido em outras linguagens como `flatten`):

---

```
1 > concat [[1,2],[3,4]]  
2 [1,2,3,4]
```

---



Ela pode ser definida utilizando compreensão de listas:

---

```
1 concat xss = [x | xs <- xss, x <- xs]
```

---

Defina a função `length` utilizando compreensão de listas!  
Dica, você pode somar uma lista de 1s do mesmo tamanho da sua lista.

Nas compreensões de lista podemos utilizar o conceito de **guardas** para filtrar o conteúdo dos geradores condicionalmente:

---

```
1 > [x | x <- [1..10], even x]
2 [2,4,6,8,10]
```

---

Vamos criar uma função chamada **divisores** que retorna uma lista de todos os divisores de  $n$ .

- 1 Qual a assinatura?

Vamos criar uma função chamada **divisores** que retorna uma lista de todos os divisores de **n**.

- 1 Qual a assinatura?
- 2 Quais os parâmetros?

---

```
1 divisores :: Int -> [Int]
```

---

Vamos criar uma função chamada **divisores** que retorna uma lista de todos os divisores de **n**.

- 1 Qual a assinatura?
- 2 Quais os parâmetros?
- 3 Qual o gerador?

---

```
1 divisores :: Int -> [Int]
2 divisores n = [???
```

---

Vamos criar uma função chamada **divisores** que retorna uma lista de todos os divisores de  $n$ .

- 1 Qual a assinatura?
- 2 Quais os parâmetros?
- 3 Qual o gerador?
- 4 Qual o guard?

---

```
1 divisores :: Int -> [Int]
2 divisores n = [x | x <- [1..n]]
```

---

Vamos criar uma função chamada **divisores** que retorna uma lista de todos os divisores de  $n$ .

- 1 Qual a assinatura?
- 2 Quais os parâmetros?
- 3 Qual o gerador?
- 4 Qual o guard?

---

```
1 divisores :: Int -> [Int]
2 divisores n = [x | x <- [1..n], n `mod` x == 0]
```

---



---

```
1 > divisores 15  
2 [1,3,5,15]
```

---

Utilizando a função `divisores` defina a função `primo` que retorna `True` se um certo número é primo.

Note que para determinar se um número não é primo a função **primo** **não** vai gerar **todos** os divisores de  $n$ .

Por ser uma avaliação preguiçosa ela irá parar na primeira comparação que resultar em **False**:

---

```
1 primo 10 => 1 : _ == 1 : 10 : [] (1 == 1)
2           => 1 : 2 : _ == 1 : 10 : [] (2 /= 10)
3           False
```

---

Com a função `primo` podemos gerar a lista dos primos dentro de uma faixa de valores:

---

```
1 primos :: Int -> [Int]  
2 primos n = [x | x <- [1..n], primo x]
```

---

A função `zip` junta duas listas retornando uma lista de pares:

---

```
1 > zip [1,2,3] [4,5,6]
2 [(1,4),(2,5),(3,6)]
3
4 > zip [1,2,3] ['a', 'b', 'c']
5 [(1,'a'),(2,'b'),(3,'c')]
6
7 > zip [1,2,3] ['a', 'b', 'c', 'd']
8 [(1,'a'),(2,'b'),(3,'c')]
```

---

Vamos criar uma função que, dada uma lista, retorna os pares dos elementos adjacentes dessa lista, ou seja:

---

```
1 > pairs [1,2,3]
2 [(1,2), (2,3)]
```

---

A assinatura será:

---

```
1 pairs :: [a] -> [(a,a)]
```

---

E a definição será:

---

```
1 pairs :: [a] -> [(a,a)]  
2 pairs xs = zip xs (tail xs)
```

---



Utilizando a função `pairs` defina a função `sorted` que retorna verdadeiro se uma lista está ordenada. Utilize também a função `and` que retorna verdadeiro se **todos** os elementos da lista forem verdadeiros.

---

```
1 sorted :: Ord a => [a] -> Bool
```

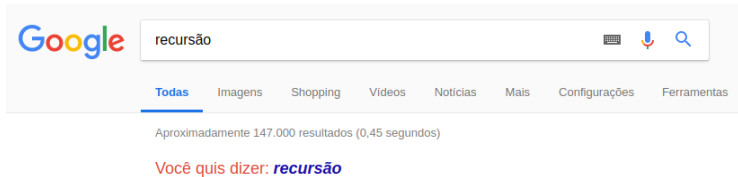
---



- Listas
- Livros [GH] 5; [SGS] 2; [ML] 2

# Recursão

---



A screenshot of a Google search interface. The search bar contains the word "recursão". To the right of the search bar are icons for keyboard input, voice search, and a magnifying glass. Below the search bar, there are navigation tabs: "Todas" (underlined), "Imagens", "Shopping", "Vídeos", "Notícias", "Mais", "Configurações", and "Ferramentas". Below the tabs, it says "Aproximadamente 147.000 resultados (0,45 segundos)". At the bottom, there is a suggestion: "Você quis dizer: **recursão**".



- A recursividade permite expressar ideias declarativas.
- Composta por um ou mais casos bases (para que ela termine) e a chamada recursiva.

$$n! = n.(n - 1)!$$

- Caso base:

$$1! = 0! = 1$$

- Para  $n = 3$ :

$$3! = 3 \cdot 2! = 3 \cdot 2 \cdot 1! = 3 \cdot 2 \cdot 1 = 6$$



---

```
1 fatorial :: Integer -> Integer
2 fatorial 0 = 1
3 fatorial 1 = 1
4 fatorial n = n * fatorial (n-1)
```

---

---

```
1 fatorial :: Integer -> Integer
2 fatorial 0 = 1
3 fatorial 1 = 1
4 fatorial n = n * fatorial (n-1)
```

---

Casos bases primeiro!!

O Haskell avalia as expressões por substituição:

---

```
1 > fatorial 4
2     => 4 * fatorial 3
3     => 4 * (3 * fatorial 2)
4     => 4 * (3 * (2 * fatorial 1))
5     => 4 * (3 * (2 * 1))
6     => 4 * (3 * 2)
7     => 4 * 6
8     => 24
```

---

Ao contrário de outras linguagens, ela não armazena o estado da chamada recursiva em uma pilha, o que evita o estouro da pilha.

---

```
1 > fatorial 4
2     => 4 * fatorial 3
3     => 4 * (3 * fatorial 2)
4     => 4 * (3 * (2 * fatorial 1))
5     => 4 * (3 * (2 * 1))
6     => 4 * (3 * 2)
7     => 4 * 6
8     => 24
```

---

A pilha recursiva do Haskell é a expressão armazenada, ele mantém uma pilha de expressão com a expressão atual. Essa pilha aumenta conforme a expressão expande, e diminui conforme uma operação é avaliada.

---

```
1 > fatorial 4
2     => 4 * fatorial 3
3     => 4 * (3 * fatorial 2)
4     => 4 * (3 * (2 * fatorial 1))
5     => 4 * (3 * (2 * 1))
6     => 4 * (3 * 2)
7     => 4 * 6
8     => 24
```

---

O algoritmo de Euclides para encontrar o Máximo Divisor Comum (*greatest common divisor* - gcd) é definido matematicamente como:

---

```
1 gcd :: Int -> Int -> Int
2 gcd a 0 = a
3 gcd a b = gcd b (a `mod` b)
```

---

---

```
1 > gcd 48 18
2   => gcd 18 12
3   => gcd 12 6
4   => gcd 6 0
5   => 6
```

---

# Recursão em Listas

---



- Podemos também fazer chamadas recursivas em listas, de tal forma a trabalhar com apenas parte dos elementos em cada chamada:

---

```
1 sum :: Num a => [a] -> a
2 sum [] = 0
3 sum ns = ???
```

---

- Podemos também fazer chamadas recursivas em listas, de tal forma a trabalhar com apenas parte dos elementos em cada chamada:

---

```
1 sum :: Num a => [a] -> a
2 sum [] = 0
3 sum ns = (head ns) + sum (tail ns)
```

---

- Por que não usar Pattern Matching?

- Podemos também fazer chamadas recursivas em listas, de tal forma a trabalhar com apenas parte dos elementos em cada chamada:

---

```
1 sum :: Num a => [a] -> a
2 sum []      = 0
3 sum (n:ns) = n + sum ns
```

---

Faça a versão caudal dessa função:

---

```
1 sum :: Num a => [a] -> a
2 sum []      = 0
3 sum (n:ns)  = n + sum ns
```

---

Como ficaria a função `product` baseado na função `sum`:

---

```
1 sum :: Num a => [a] -> a
2 sum []      = 0
3 sum (n:ns) = n + sum ns
```

---

Como ficaria a função `product` baseado na função `sum`:

---

```
1 product :: Num a => [a] -> a
2 product []      = 0
3 product (n:ns) = n + sum ns
```

---

Como ficaria a função `product` baseado na função `sum`:

---

```
1 product :: Num a => [a] -> a
2 product []      = 1
3 product (n:ns) = n * product ns
```

---

E a função `length`?

---

```
1 sum :: Num a => [a] -> a
2 sum []      = 0
3 sum (n:ns) = n + sum ns
```

---



E a função `length`?

---

```
1 length :: [a] -> Int
2 length []      = 0
3 length (n:ns) = 1 + length ns
```

---

- Reparem que muitas soluções recursivas (principalmente com listas) seguem um mesmo esqueleto. Uma vez que vocês dominem esses padrões, fica fácil determinar uma solução.
- Nas próximas aulas vamos criar funções que generalizam tais padrões.

Crie uma função recursiva chamada `insert` que insere um valor `x` em uma lista `ys` ordenada de tal forma a mantê-la ordenada:

---

```
1 insert :: Ord a => a -> [a] -> [a]
```

---

Crie uma função recursiva chamada `isort` que utiliza a função `insert` para implementar o Insertion Sort:

---

```
1 isort :: Ord a => [a] -> [a]
```

---

Em alguns casos o retorno da função recursiva é a chamada dela mesma **múltiplas** vezes:

---

```
1 fib :: Int -> Int
2 fib 0 = 1
3 fib 1 = 1
4 fib n = fib (n-1) + fib (n-2)
```

---

Complete a função `qsort` que implementa o algoritmo Quicksort:

---

```
1 qsort :: Ord a => [a] -> [a]
2 qsort []      = []
3 qsort (x:xs) = qsort menores ++ [x] ++ qsort maiores
4   where
5     menores = [a | ???]
6     maiores = [b | ???]
```

---



- Recursão
  - ▶ Exercícios
- Livros [GH] 6; [SGS] 2; [ML] 5
- Livros [GH] 5; [SGS] 2; [ML] 2

Tarefa para casa

---



- Lista 1
  - ▶ Prazo: 19/10
  - ▶ Link para submissão:  
<https://classroom.github.com/a/1ayC4yUs>
- Não deixe de escolher o seu nome na lista para que possamos relacionar o usuário GitHub à você!

Tentem fazer os exercícios da lista:

[https://wiki.haskell.org/99\\_questions/1\\_to\\_10](https://wiki.haskell.org/99_questions/1_to_10)

E os 10 primeiros exercícios do Project Euler:

<https://projecteuler.net/archives>

No próximo sábado iremos detalhar a solução de alguns deles.

## Referências

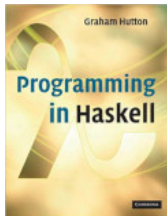
---

Os principal texto utilizado neste curso será o [GH] Segunda Edição.



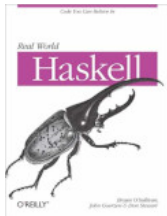
- Programming in Haskell. 2nd Edition.
  - ▶ Por *Graham Hutton*.

A primeira edição (antiga), que tem boa parte do conteúdo da segunda edição, está disponível na biblioteca:



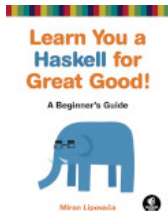
- Link Biblioteca: [http://biblioteca.ufabc.edu.br/index.php?codigo\\_sophia=15287](http://biblioteca.ufabc.edu.br/index.php?codigo_sophia=15287)

- [SGS]



- Real World Haskell.
  - ▶ Por *Bryan O'Sullivan, John Goerzen e Don Stewart*.
  - ▶ Disponível **gratuitamente** em:  
<http://book.realworldhaskell.org/>

- [ML]



- Learn You a Haskell for Great Good!: A Beginner's Guide.
  - ▶ Por *Miran Lipovača*.
  - ▶ Disponível **gratuitamente** em:  
<http://learnyouahaskell.com/>