

# **Introdução a Programação Funcional com Haskell**

---

Fabrcio Olivetti de Franca, Emlio Francesquini

22 de Setembro de 2018

# 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.

## Paradigma Estruturado

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

## Orientação a Objetos

```
class Aprovados {  
    private ArrayList aprovados;  
    public Aprovados () {  
        aprovados = new ArrraList();  
    }  
    public addAluno(aluno) {  
        if (aluno.nota >= 5) {  
            aprovados.add(aluno.nome.toUpperCase());  
        }  
    }  
    public getAprovados() {  
        return aprovados.sort();  
    }  
}
```

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

Muitas linguagens de programação estão incorporando elementos de paradigma funcional.



## Paradigma Funcional no Python

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)
```

```
public interface List<E> {  
    void add(E x);  
    Iterator<E> iterator();  
}
```

```
array.stream()  
    .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.

```
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:

```
int x    = 10;  
double y = 5.1;  
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:

```
x = 10  :: Int
y = 5.1 :: Double
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.

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

```
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.

```
x = 1.0
```

```
x = 2.0
```

ERRO!

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

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

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

```
f 0 = 1
```

```
f n = 2 * f (n-1)
```

```
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*).

```
print (aplique dobro [1,2,3,4])  
> [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 + 2           -- 3
1.0 + 3.0       -- 4.0
(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.

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

# **Ambiente de Programação**

**Glasgow Haskell Compiler:** compilador de código aberto para a linguagem Haskell.

Possui um modo interativo **ghci** (similar ao **iPython**).



No terminal:

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

ou

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

Visual Studio Code

com os pacotes:

- Haskell Syntax Highlighting
- Haskero
- hoople-vscode

Atom

com os pacotes:

- haskell-grammar
- language-haskell

# **Primeiro Projeto**

## Primeiro projeto compilável

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

```
$ stack new primeiro-projeto simple
```

```
$ cd primeiro-projeto
```

```
$ stack setup
```

```
$ stack build
```

```
$ 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.

```
module Main where    -- indica que é o módulo principal

main :: IO ()
main = do            -- início da função principal
    putStrLn "hello world"    -- imprime hello world
```



```
$ stack ghci
```

```
> 2+3*4
```

```
14
```

```
> (2+3)*4
```

```
20
```

```
> sqrt (3^2 + 4^2)
```

```
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 (+,-).

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

## Informação de operadores

Para saber a precedência de um operador basta digitar:

```
> :i (+)
class Num a where
  (+) :: a -> a -> a
  ...
  -- Defined in 'GHC.Num'
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:

```
f a b      -- f(a,b)
```

```
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`:

```
dobra x = x + x
```

```
quadruplica x = dobra (dobra x)
```

No GHCi:

```
> :l teste.hs  
> quadruplica 10  
40
```

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

Acrescentem a seguinte linha no arquivo fonte:

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

e no GHCi:

```
> :reload
```

```
> fatorial 5
```

```
120
```



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

```
> :t dobra
```

```
dobra :: Num a => a -> a
```

```
> :q
```

```
$
```

# 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  $css$ .

## Regra de layout

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

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

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

A definição de tabulação varia de editor para editor. Como o espaço é importante no Haskell, usem espaço ao invés de tab.

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

```
-- função que dobra o valor de x  
dobra x = x + x
```

```
{-  
dobra recebe uma variável numérica  
e retorna seu valor em dobro.  
-}
```



# **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

$v :: T$

significando que  $v$  define um valor do tipo  $T$ .

False :: Bool

True :: Bool

10 :: Int

O compilador GHC já vem com suporte nativo a diversos tipos básicos para uso.

Durante o curso veremos como definir alguns deles.

- **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' e a seta tripla '≡'.
- **String:** sequências de caracteres delimitados por aspas duplas: "Olá Mundo".

- **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:

```
x = 3
```

O tipo de *x* pode ser *Int*, *Integer*, *Float* e *Double*. Qual tipo devemos atribuir a *x*?



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

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

Uma lista de tipo T tem tipo [T]:

```
[1,2,3,4]           :: [Int]
```

```
[False, True, True] :: [Bool]
```

```
['o', 'l', 'a']     :: [Char]
```

Também podemos ter listas de listas:

```
[ [1,2,3], [4,5] ] :: [[Int]]
```

```
[ [ '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:

```
(True, False)      :: (Bool, Bool)
```

```
(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  $T2$ :

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

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

## Funções de múltiplos argumentos

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

```
soma :: Int -> Int -> Int
```

```
soma x y = x + y
```

```
mult :: Int -> Int -> Int -> Int
```

```
mult x y z = x*y*z
```



# Polimorfismo

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

```
[1,2,3,4]           :: [Int]
```

```
[False, True, True] :: [Bool]
```

```
['o', 'l', 'a']     :: [Char]
```

Qual o tipo de length?

```
[1,2,3,4]           :: [Int]
```

```
[False, True, True] :: [Bool]
```

```
['o', 'l', 'a']     :: [Char]
```

Qual o tipo de `length`?

```
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. De todo modo, essa função **deve** ser aplicada a tipos numéricos.

## Overloaded types

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

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

O operador `+` recebe dois tipos de uma classe numérica e retorna um valor desse tipo.

## Overloaded types

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

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



## Exemplos de funções

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

```
impar :: Integral a => a -> Bool
```

```
impar n = n `mod` 2 == 1
```

## Exemplos de funções

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

```
quadrado :: Num a => a -> a
```

```
quadrado n = n*n
```

## Exemplos de funções

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

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

## Exercício 01

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

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

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

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

```
raiz2Grau :: Floating a => a -> a -> a -> (a, a)
raiz2Grau a b c = ( ((-b) + sqrt (b^2 - 4*a*c)) / (2*a),
                   ((-b) - sqrt (b^2 - 4*a*c)) / (2*a) )
```

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

```
f x = y + z
```

```
  where
```

```
    y = e1
```

```
    z = e2
```

## Exemplo

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

## Exercício 02

Reescreva a função `raiz2Grau` utilizando `where`.



## Resposta 02

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

```
raiz2Grau :: Floating a => a -> a -> a -> (a, a)
```

```
raiz2Grau a b c = (x1, x2)
```

**where**

```
x1      = ((-b) + sqDelta) / (2*a)
```

```
x2      = ((-b) - sqDelta) / (2*a)
```

```
sqDelta = sqrt delta
```

```
delta   = b^2 - 4*a*c
```

## Condicionais

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

```
abs :: Num a => a -> a
```

```
abs n = if (n >= 0) then n else (-n)
```

ou

```
abs :: Num a => a -> a
```

```
abs n = if (n >= 0)
      then n
      else (-n)
```

Também podemos encadear condicionais:

```
signum :: (Ord a, Num a) => a -> a
signum n = if (n == 0)
            then 0
            else if (n > 0)
                   then 1
                   else (-1)
```

## Exercício 03

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

Note que a assinatura da função agora deve ser:

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

## Resposta 03

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

```
raiz2Grau :: (Ord a, Floating a) => a -> a -> a -> (a, a)
raiz2Grau a b c = (x1, x2)
  where
    x1 = if delta >= 0
          then ((-b) + sqDelta) / (2*a)
          else 0
    x2 = if delta >= 0
          then ((-b) - sqDelta) / (2*a)
          else 0
    sqDelta = sqrt delta
    delta = b^2 - 4*a*c
```

## Expressões *guardadas* (Guard Expressions)

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

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

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

## Expressões *guardadas* (Guard Expressions)

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

```
classificaIMC :: Double -> String
classificaIMC imc
  | imc <= 18.5 = "abaixo do peso"
  -- não preciso fazer && imc > 18.5
  | imc <= 25.0 = "no peso correto"
  | imc <= 30.0 = "acima do peso"
  | otherwise   = "muito acima do peso"
```

## Exercício 04

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

Para isso utilize a função `error`:

```
error "Raízes negativas."
```



## Resposta 04

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

```
raiz2Grau :: (Ord a, Floating a) => a -> a -> a -> (a, a)
raiz2Grau a b c
  | delta >= 0    = (x1, x2)
  | otherwise     = error "Raízes negativas."
where
  x1      = ((-b) + sqDelta) / (2*a)
  x2      = ((-b) - sqDelta) / (2*a)
  sqDelta = sqrt delta
  delta   = b^2 - 4*a*c
```

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:

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

```
not x = if (x == True) then False else True
```

Podemos reescreve-la utilizando guardas:

```
not :: Bool -> Bool
not x | x == True  = False
      | x == False = True
```

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

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

```
not True = False
```

```
not False = True
```

## Pattern Matching

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

```
soma :: (Eq a, Num a) => a -> a -> a
```

```
soma x 0 = x
```

```
soma 0 y = y
```

```
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:

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

## Pattern Matching

Implemente a multiplicação utilizando Pattern Matching:

```
mul :: (Eq a, Num a) => a -> a -> a
```

```
mul 0 y = 0
```

```
mul x 0 = 0
```

```
mul x 1 = x
```

```
mul 1 y = y
```

```
mul x y = x*y
```



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

```
mul :: (Eq a, Num a) => a -> a -> a
```

```
mul 0 _ = 0
```

```
mul _ 0 = 0
```

```
mul x 1 = x
```

```
mul 1 y = y
```

```
mul x y = x*y
```

## Pattern Matching

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

```
mul :: (Eq a, Num a) => a -> a -> a
```

```
mul 0 _ = 0
```

```
mul _ 0 = 0
```

```
mul x 1 = x
```

```
mul 1 y = y
```

```
mul x y = x*y
```

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

```
-- Recebe um valor numérico e
-- retorna uma função que
-- recebe um número e retorna outro número
somaMultX :: Num a => a -> (a -> a)
somaMultX x = \y -> x + x * y

-- somaMult2 é uma função que
-- retorna um valor multiplicado por 2
somaMult2 = somaMultX 2
```

# Operadores

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

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

```
x :+ y = abs x + y
```

ou

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

```
(:+) x y = abs x + y
```

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

```
> mod 10 3
```

```
1
```

```
> 10 `mod` 3
```

```
1
```

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

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

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

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

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

> (/) 4 2

2

> (/2) 4

2

> (4/) 2

2

## Exercício 05

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

```
(&&) :: Bool -> Bool -> Bool
```

```
True  && True   = True
```

```
True  && False  = False
```

```
False && True    = False
```

```
False && False  = False
```



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

```
(&&) :: Bool -> Bool -> Bool
```

```
True && True = True
```

```
_     && _     = False
```

# Listas

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

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

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

(:) - concatenação de elemento com lista

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

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

É uma lista ligada!!

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

Faixa de valores inclusivos:

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



Faixa de valores inclusivos com tamanho do passo:

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

Lista infinita:

```
[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:

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

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

Efetivamente ele faz:

```
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.

**COFFEE BREAK (não incluso)**

## **Funções básicas para manipular listas**

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

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

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



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

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

```
0
```

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

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

## Exercício 06

O que a seguinte expressão retornará?

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

## Exercício 06

O que a seguinte expressão retornará?

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

```
1
```

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

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

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

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

## Exercício 07

Implemente o operador !! utilizando as funções anteriores.

Implemente o operador !! utilizando as funções anteriores.

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



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

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

```
10
```

As funções `sum` e `product` retornam a somatória e produtória da lista:

```
> sum [1..10]
```

```
55
```

```
> product [1..10]
```

```
3628800
```

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

```
> [1..3] ++ [4..10] == [1..10]
```

```
True
```

```
> 1 : [2..10] == [1..10]
```

```
True
```

## Exercício 08

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

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

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

## **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 : [])`
- Lista com um elemento seguido de vários outros: `(x : xs)`

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



## Implementando a função nulo

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

```
null :: [a] -> Bool
```

```
null [] = True
```

```
null _ = False
```

## Implementando a função tamanho

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

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

## Exercício 09

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

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

```
take :: Int -> [a] -> [a]
```

```
take n _ | n <= 0 = []
```

```
take _ []         = []
```

```
take n (x:xs)    = x : take (n-1) xs
```

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

```
> "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:

```
> [x^2 | x <- [1..5]]  
[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:

```
> [toLower c | c <- "OLA MUNDO"]
```

```
"ola mundo"
```

```
> [(x, even x) | x <- [1,2,3]]
```

```
[(1, False), (2, True), (3, False)]
```

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

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

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

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

Equivalente a:

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

## Exemplo: concat

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

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

Ela pode ser definida utilizando compreensão de listas:

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

## Exercício 10: length

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



```
length xs = sum [1 | _ <- xs]
```

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

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

Vamos criar uma função chamada `divisores` que retorna uma lista de todos os divisores de  $n$ . Qual a assinatura?

Vamos criar uma função chamada `divisores` que retorna uma lista de todos os divisores de `n`. Quais os parâmetros?

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

Vamos criar uma função chamada `divisores` que retorna uma lista de todos os divisores de `n`. Qual o gerador?

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

Vamos criar uma função chamada `divisores` que retorna uma lista de todos os divisores de `n`. Qual o guard?

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

Vamos criar uma função chamada `divisores` que retorna uma lista de todos os divisores de `n`. Qual o guard?

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

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



## Exercício 11

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

## Resposta 11

```
primo :: Int -> Bool
primo n = divisores n == [1,n]
```

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`:

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

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

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

## A função zip

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

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

```
> zip [1,2,3] ['a', 'b', 'c']  
[(1,'a'),(2,'b'),(3,'c')]
```

```
> zip [1,2,3] ['a', 'b', 'c', 'd']  
[(1,'a'),(2,'b'),(3,'c')]
```

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

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

A assinatura será:

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

E a definição será:

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



## Exercício 12

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.

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

```
sorted :: Ord a => [a] -> Bool
sorted xs = and [x <= y | (x, y) <- pairs xs]
```

# 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  $n = 3$ :

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

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

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

Casos bases primeiro!!



O Haskell avalia as expressões por substituição:

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

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

```
> fatorial 4
=> 4 * fatorial 3
=> 4 * (3 * fatorial 2)
=> 4 * (3 * (2 * fatorial 1))
=> 4 * (3 * (2 * 1))
=> 4 * (3 * 2)
=> 4 * 6
=> 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.

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

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

```
gcd :: Int -> Int -> Int
```

```
gcd a 0 = a
```

```
gcd a b = gcd b (a `mod` b)
```

```
> gcd 48 18  
=> gcd 18 12  
=> gcd 12 6  
=> gcd 6 0  
=> 6
```

A multiplicação Etíope de dois números  $m, n$  é dado pela seguinte regra:

- Se  $m$  for par, o resultado é a aplicação da multiplicação em  $m/2, n * 2$ .
- Se  $m$  for ímpar, o resultado a aplicação da multiplicação em  $m/2, n * 2$  somados a  $n$ .
- Se  $m$  for igual a 1, retorne  $n$ .

Exemplo:

m	n	r
14	12	0
7	24	24
3	48	72
1	96	168

Implemente o algoritmo recursivo da Multiplicação Etíope.



```
etiope :: Integer -> Integer -> Integer
```

```
etiope m n
```

```
  | m==1      = n
```

```
  | par m      = etiope (m `div` 2) (n * 2)
```

```
  | otherwise = n + (etiope (m `div` 2) (n * 2))
```

## **Recursão em Listas**

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

```
sum :: Num a => [a] -> a
```

```
sum [] = 0
```

```
sum ns = ???
```

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

```
sum :: Num a => [a] -> a
sum [] = 0
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:

```
sum :: Num a => [a] -> a
```

```
sum [] = 0
```

```
sum (n:ns) = n + sum ns
```

## Exercício 13

Faça a versão caudal dessa função:

```
sum :: Num a => [a] -> a
```

```
sum [] = 0
```

```
sum (n:ns) = n + sum ns
```

## Exercício 13

Faça a versão caudal dessa função:

```
sum :: Num a => [a] -> a
```

```
sum [] = 0
```

```
sum ns = sum' ns 0
```

```
  where
```

```
    sum' [] s      = s
```

```
    sum' (n:ns) s = sum' ns (n+s)
```

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

```
sum :: Num a => [a] -> a
```

```
sum [] = 0
```

```
sum (n:ns) = n + sum ns
```



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

```
product :: Num a => [a] -> a
```

```
product [] = 0
```

```
product (n:ns) = n + sum ns
```

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

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

E a função `length`?

```
sum :: Num a => [a] -> a
```

```
sum [] = 0
```

```
sum (n:ns) = n + sum ns
```

E a função `length`?

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

```
length [] = 0
```

```
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.

Considere a função reverse:

```
> :t reverse
```

```
reverse :: [a] -> [a]
```

```
> reverse [1,2,3]
```

```
[3,2,1]
```

Como poderíamos implementá-la?

Vamos começar pelo caso base, o inverso de uma lista vazia, é vazia:

```
reverse :: [a] -> [a]
```

```
reverse [] = []
```

## Invertendo uma lista

Vamos começar pelo caso base, o inverso de uma lista com um elemento, é ela mesma:

```
reverse :: [a] -> [a]
```

```
reverse [] = []
```

```
reverse [x] = [x]
```



## Invertendo uma lista

Vamos começar pelo caso base, o inverso de uma lista com dois elementos é:

```
reverse :: [a] -> [a]
```

```
reverse [] = []
```

```
reverse [x] = [x]
```

```
reverse [x,y] = [y,x]
```

## Invertendo uma lista

Vamos começar pelo caso base, o inverso de uma lista com três elementos é:

```
reverse :: [a] -> [a]
reverse []      = []
reverse [x]     = [x]
reverse [x,y]   = [y,x]
reverse [x,y,z] = [z,y,x]
```

Esse último caso base nos dá uma ideia de como generalizar! Note que:

```
> reverse [1,2,3] == reverse [2,3] ++ [1]
```

## Invertendo uma lista

Vamos começar pelo caso base, o inverso de uma lista com três elementos é:

```
reverse :: [a] -> [a]
```

```
reverse [] = []
```

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

## Exercício 14

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

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

## Resposta 14

```
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) | x <= y    = x:y:ys
                 | otherwise = y : insert x ys
```

## Exercício 15

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

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

## Resposta 15

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



Em alguns casos o retorno da função recursiva é a chamada dela mesma em múltiplas instâncias:

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

## Exercício 16

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

```
qsort :: Ord a => [a] -> [a]
qsort []      = []
qsort (x:xs) = qsort menores ++ [x] ++ qsort maiores
  where
    menores = [a | ???]
    maiores = [b | ???]
```

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

```
qsort :: Ord a => [a] -> [a]
qsort []      = []
qsort (x:xs) = qsort menores ++ [x] ++ qsort maiores
  where
    menores = [a | a <- xs, a <= x]
    maiores = [b | b <- xs, b > x]
```

## Quickcheck

Uma parte importante da Engenharia de Software é o teste de seu produto final. Dado que o programa compilou sem erros, ele faz o que é esperado?

O Haskell permite, em algumas situações, provar matematicamente que seu programa está correto (usando indução).

Outra forma de verificar a corretude é fazer testes de entrada e saída das funções criadas e verificar se elas apresentam as propriedades esperadas.

Se você criou um novo algoritmo de ordenação, que propriedades são esperadas?

- A lista de saída está ordenada
- A lista de saída tem o mesmo tamanho da lista de entrada
- A lista de saída contém os mesmos elementos da lista de entrada

Edite o arquivo *.cabal* e acrescente o seguinte ao final da linha *build-depends*:

```
build-depends: base >= 4.7 && <5, QuickCheck
```

## Testando QuickSort

Vamos trocar o sinal `<=` pelo sinal de `<` no nosso código do `qsort`:

```
import Test.QuickCheck
```

```
qsort :: Ord a => [a] -> [a]
```

```
qsort [] = []
```

```
qsort (x:xs) = qsort menores ++ [x] ++ qsort maiores
```

```
  where
```

```
    menores = [a | a <- xs, a <= x]
```

```
    maiores = [b | b <- xs, b > x]
```



Vamos testar uma primeira propriedade de algoritmos de ordenação:  
**idempotência.**

Queremos mostrar que `qsort (qsort xs) == qsort xs`:

```
prop_idempotencia :: Ord a => [a] -> Bool
```

```
prop_idempotencia xs = qsort (qsort xs) == qsort xs
```

## Idempotência

Vamos testar essa função no ghci (use `stack ghci src/Main.hs`):

```
> prop_idempotencia [1]
True
> prop_idempotencia [1,2,3,4]
True
> prop_idempotencia [3,2,4,1]
True
> prop_idempotencia [4,3,2,1]
True
> prop_idempotencia []
True
```

Outra propriedade é que o tamanho da lista seja o mesmo após a execução do algoritmo:

```
prop_length :: Ord a => [a] -> Bool
```

```
prop_length xs = length (qsort xs) == length xs
```

## Tamanho da lista

```
> prop_length [1]
```

```
True
```

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

```
True
```

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

```
True
```

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

```
True
```

```
> prop_length []
```

```
True
```

Os casos de teste utilizado são representativos?

A biblioteca QuickCheck automatiza a geração de dados para testes (e faz outras coisas úteis também).

## Verificando nossas propriedades

```
> quickCheck prop_idempotencia
+++ OK, passed 100 tests.
> quickCheck prop_length
*** Failed! Falsifiable (after 4 tests):
[(),()]
```

Oops!

A biblioteca QuickCheck gera casos de testes progressivos, começando de casos simples até casos mais complexos em busca de erros.

Ao encontrar um erro, ele retorna a instância mais simples que deu errado.

Para entender melhor vamos executar essa função para listas de inteiros:

```
> quickCheck (prop_length :: [Int] -> Bool)
*** Failed! Falsifiable (after 5 tests and 1 shrink):
[1,1]
```

O que houve?



```
> qsort [1,1]  
[1]
```

Outra propriedade é que primeiro elemento da lista é igual ao mínimo:

```
prop_minimum :: Ord a => [a] -> Bool
```

```
prop_minimum xs = head (qsort xs) == minimum xs
```

## Mínimo == head

Vamos testar essa função no ghci (use `stack ghci src/Main.hs`):

```
> quickCheck prop_minimum
```

```
*** Failed! Exception:
```

```
'Prelude.head: empty list' (after 1 test):
```

```
[]
```

## Mínimo == head

Tanto a função `minimum` quanto a função `head` retornam erro em listas vazias, podemos especificar que não queremos testar instâncias nulas com o operador `==>`:

```
prop_minimum :: Ord a => [a] -> Property
prop_minimum xs = not (null xs)
                  ==> head (qsort xs) == minimum xs
```

Esse operador retorna uma propriedade interpretável pelo `quickCheck`.

## Mínimo == head

Vamos testar essa função no ghci (use *stack ghci src/Main.hs*):

```
> quickCheck prop_minimum  
+++ OK, passed 100 tests.
```

Finalmente, se temos um algoritmo que cumpre a mesma tarefa e temos certeza de que está correto, podemos usá-lo na comparação:

```
import Data.List -- sort

prop_model :: Ord a => [a] -> Bool
prop_model xs = qsort xs == sort xs
```

## **Tarefa para casa**

## Sugestões de exercícios para praticar

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.