Introdução a Programação Funcional com Haskell

Aplicações em data science e na web

Dia 01

Fabrício Olivetti e Emilio Francesquini
folivetti@ufabc.edu.br e.francesquini@ufabc.edu.br

28/09/2019

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

Disclaimer

UFABC

- 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.

UFABC

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

UFABC

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

}
```

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

Muitas linguagens de programação estão incorporando elementos de paradigma funcional.

UFABC

O Python possui alguns elementos do paradigma funcional:

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

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

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

UFABC

```java
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).

UFABC

- **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.

```haskell
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!

UFABC

- **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.

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

```
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!

Haskell: Características

- **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.

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

UFABC

- **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.

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

print (f 10)
```

UFABC

- **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]
```

UFABC

- **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.

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

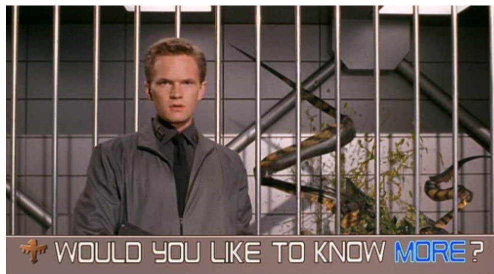
UFABC

- **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)
```

Para saber mais

UFABC



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

Ambiente de Programação

UFABC

- 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

UFABC

- **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/

Verificando a instalação

UFABC

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

Editores recomendados

UFABC

- 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.

Editores Recomendados - VSCode

UFABC

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.

Para saber mais

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

Primeiro Projeto

Primeiro projeto compilável

UFABC

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.

Stack

UFABC

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

UFABC

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

UFABC

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

main :: IO ()
main = do                          -- início da função principal
  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
```

UFABC

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))
```

Informação de operadores

UFABC

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:

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

UFABC

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

Matemática	Haskell
$f(x)$	f x
$f(x, y)$	f x y
$f(g(x))$	f (g x)
$f(x, g(y))$	f x (g y)
$f(x)g(y)$	f x * g y

UFABC

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

```haskell
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.

UFABC

Acrescentem a seguinte linha no arquivo fonte:

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

e no GHCi:

```
> :reload
> fatorial 5
120
```

Outros comandos do GHCi

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

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

> :q
$
```

UFABC

- `: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
```

Para saber mais

UFABC

- Funções
- [GH] 4
- [SGS] 2
- [ML] 4

Convenções

Requisitos dos nomes

- 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'
```

UFABC

- 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
```

Convenção para listas

UFABC

- 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

UFABC

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

```haskell
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` e `b` fazem parte da cláusula `where`. A definição de `z` não faz parte de `f`.

Tabs vs Espaço

- 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. 😛

https://www.youtube.com/watch?v=SsoOG6ZeyUI

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 dados

UFABC

- 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

UFABC

- Em Haskell, os tipos são definidos pela notação

```haskell
v :: T
```

- Significando que v define um valor do tipo T.

```haskell
False :: Bool
True :: Bool
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.

Tipos Básicos

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*[1] `'🕴'`.
- `String`: sequências de caracteres delimitados por aspas duplas: `"Olá Mundo"`.

[1] Este é o nome oficial do caracter na tabela Unicode v.7.0!

Tipos Básicos

UFABC

- `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` ou `Double`.

Pergunta

Qual tipo devemos atribuir a `x`?

UFABC

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

UFABC

- 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 `(T1, T2,...,Tn)`.

UFABC

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

UFABC

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

```haskell
not  :: Bool -> Bool
even :: Int -> Bool
```

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

```haskell
soma :: Int -> Int -> Int
soma x y = x + y

mult :: Int -> Int -> Int -> Int
mult x y z = x*y*z
```

Para saber mais

UFABC

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

Polimorfismo

UFABC

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]
```

Pergunta

Qual é então o tipo de `length`?

UFABC

- 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,...`.

UFABC

- 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.

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*).
- Uma restrição é escrita na forma `C a`, onde `C` é o nome da classe e `a` uma variável de tipo.

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

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

UFABC

- 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
```

UFABC

- 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:

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

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
```

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 1

UFABC

Escreva uma função que retorne a raíz 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`.

UFABC

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

```
1  raiz2Grau :: Floating a => a -> a -> a -> (a, a)
2  raiz2Grau a b c = ( ((-b) + sqrt (b^2 - 4*a*c)) / (2*a),
3                      ((-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

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

Exercício 2

Reescreva a função `raiz2Grau` utilizando `where`.

Resposta 2

UFABC

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

```haskell
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
```

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

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

ou

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

UFABC

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 3

UFABC

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)
```

UFABC

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

```haskell
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
```

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`.

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

```haskell
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 4

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

Para isso utilize a função `error`:

```
error "Raízes negativas."
```

UFABC

Escreva uma função que retorne a raíz 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
```

Nota sobre erro

UFABC

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:

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

Podemos reescreve-la utilizando guardas:

```haskell
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:

```haskell
not :: Bool -> Bool
not True  = False
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:

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

UFABC

Implemente a multiplicação utilizando Pattern Matching:

```haskell
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):

```
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
```

UFABC

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

```haskell
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
```

UFABC

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

```haskell
-- 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
```

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
```

UFABC

Sendo # um operador, temos que `(#), (x #), (# y)` são chamados de seções, e definem:

```
(#)   = \x -> (\y -> x # y)
(x #) = \y -> x # y
(# y) = \x -> x # y
```

Essas formas são também conhecidas como point-free notation:

```
> (/) 4 2
2
> (/2) 4
2
> (4/) 2
2
```

Exercício 5

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

```haskell
(&&&) :: Bool -> Bool -> Bool
True  &&& True  = True
True  &&& False = False
False &&& True  = False
False &&& False = False
```

Resposta 5

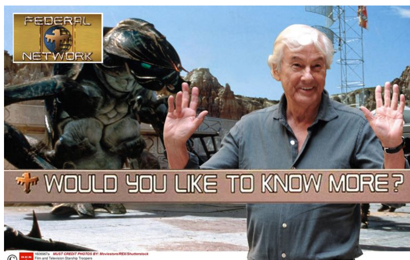
UFABC

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

```
1  (&&&) :: Bool -> Bool -> Bool
2  True &&& True = True
3  _    &&& _    = False
```

Para saber mais

UFABC



- 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 λ 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.

```
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:

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

UFABC

É uma lista ligada!!

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

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

Criando listas

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

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

Criando listas

Faixa de valores inclusivos:

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

Criando listas

Faixa de valores inclusivos com tamanho do passo:

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

UFABC

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:

```
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.

Para saber mais

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

COFFEE BREAK (não incluso)

Funções básicas para manipulação de listas

Recuperar elementos - `!!`

- 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
```

Recuperar elementos - `tail`

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 6

UFABC

O que a seguinte expressão retornará?

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

Exercício 6 - Solução

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:

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

Exercício 7

UFABC

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

UFABC

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

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

Tamanho da lista - `length`

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

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

UFABC

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

```
> sum [1..10]
55
> product [1..10]
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..3] ++ [4..10] == [1..10]
True
> 1 : [2..10] == [1..10]
True
```

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

Exercício 8

UFABC

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

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

```haskell
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 : [])` 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`:

```haskell
null :: [a] -> Bool
null [] = True
null _  = False
```

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

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

Implemente a função `take`. Se `n <= 0` deve retornar uma lista vazia.

UFABC

Implemente a função `take`. Se `n <= 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!

UFABC

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
  }
}
```

UFABC

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]
```

UFABC

Ela pode ser definida utilizando compreensão de listas:

```haskell
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 1s do mesmo tamanho da sua lista.

Exercício 10 - `length` - Solução

```
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]
```

UFABC

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

1. Qual a assinatura?

Exemplo: Divisores

UFABC

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

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

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

Exemplo: Divisores

UFABC

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?

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

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

UFABC

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?

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

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

Exercício 11

UFABC

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

Exercício 11 - Solução

UFABC

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

UFABC

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]
```

UFABC

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')]
```

Função pairs

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:

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

Exercício 12

UFABC

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
```

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

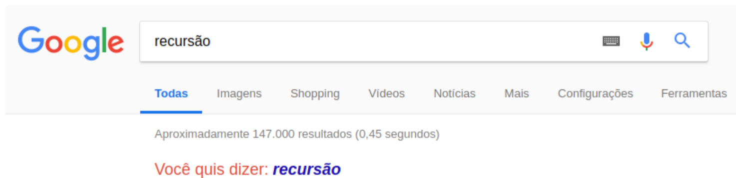
Para saber mais

UFABC

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

Recursão

Recursão



UFABC

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

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

UFABC

- Caso base:

$$1! = 0! = 1$$

- Para $n = 3$:

3! = 3 . 2! = 3 . 2 . 1! = 3 . 2 . 1 = 6

UFABC

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

UFABC

```haskell
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:

```
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
```

Fatorial

UFABC

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
```

Fatorial

UFABC

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:

```haskell
gcd :: Int -> Int -> Int
gcd a 0 = a
gcd a b = gcd b (a `mod` b)
```

Máximo Divisor Comum

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

Recursão em Listas

UFABC

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

```haskell
sum :: Num a => [a] -> a
sum [] = 0
sum ns = ???
```

UFABC

- 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:

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

Exercício 13

UFABC

Faça a versão caudal dessa função:

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

Exercício 13 - Solução

Faça a versão caudal dessa função:

```haskell
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
```

Produtória

UFABC

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`:

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

UFABC

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
```

UFABC

- 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.

Exercício 14

UFABC

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

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

Exercício 14 - Resposta

UFABC

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

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

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

Exercício 15 - Resposta

UFABC

```haskell
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 **múltiplas** vezes:

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

UFABC

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 | ???]
```

Exercício 16 - Resposta

UFABC

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

Para saber mais

UFABC

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

Tarefa para casa

Para entregar

UFABC

- 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!

Sugestões de exercícios para praticar

UFABC

Tentem fazer os exercícios da lista:

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

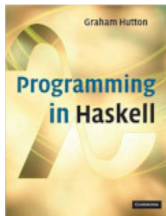
Livros

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

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

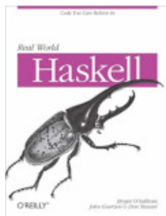
UFABC

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`

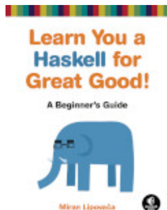
- [SGS]



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

Livros

- [ML]



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