Introdução a Programação Funcional com Haskell

Dia 01

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

29/02/2020

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de **Introdução a Programação Funcional com Haskell** da 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.

Paradigma Estruturado

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

```
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.toUpper());
9          }
10     }
11     public getAprovados() {
12         return aprovados.sort();
13     }
14
15 }
```

UFABC

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

- Muitas linguagens de programação estão incorporando elementos de paradigma funcional por conta de seus benefícios.

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

```
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.

```
take 100 [x | x <- nat, primo 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. Como consequência não precisamos nos preocupar se uma variável foi passada como referência ou não.

```haskell
x = 1.0
x = 2.0
```

ERRO!

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.

```
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.

```haskell
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]
```

- **Tipos polimórficos:** permite definir funções genéricas que funcionam para classes de tipos. Por exemplo, a função `fst` retorna o primeiro elemento de uma tupla, os tipos dos elementos não importam.

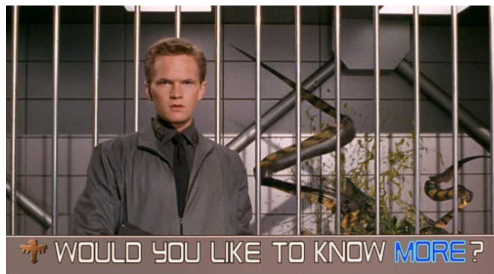
```
1  fst :: (a,b) -> a
2  fst (x,y) = x
```

- 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 infinitas.

```haskell
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

Olá Mundo

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

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

- Em Haskell, a aplicação de função é definida como:
  - o nome da função...
  - ... seguido dos parâmetros separados por espaço
  - A aplicação de funções tem maior precedência

```
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 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

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:
- Apesar de haver suporte para caracteres Unicode, o seu uso é controverso pela dificuldade envolvida na sua digitação.

```
funcao, ordenaLista, soma1, x'
```

Nomes reservados

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

UFABC

- 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. (embora Vim seja superior 😛)

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

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

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

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

UFABC

- Funções
- [GH] 4
- [SGS] 2
- [ML] 4

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

- Em Haskell, os tipos são definidos pela notação

```
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.

UFABC

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!

- `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.

UFABC

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`?

UFABC

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

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

UFABC

Uma lista de tipo T tem tipo [T]:

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

UFABC

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

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)`.

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 escrito como `T1 -> T2`, ou seja, o mapa do tipo `T1` para o tipo `T2`:

```
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:

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

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

Para saber mais

- 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:

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

Pergunta

Qual é então o tipo de `length`?

- Qual o tipo de `length`?

```haskell
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.

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

```
(+) :: 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:

```haskell
3 :: Num a => a
```

O que resolve nosso problema anterior.

- Funções podem 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
```

- Funções podem 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
```

Exemplos de funções

- Funções podem 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:

```
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 raíz 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)
                  )
```

UFABC

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

```
f x = y + z
  where
    y = e1
    z = e2
```

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

Exercício 2

UFABC

- 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)
```

Condicionais

UFABC

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)
```

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)
```

Resposta 3

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 = (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`.

UFABC

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"
```

- Utilizando guards, reescreva a função `raiz2Grau` para retornar um erro com raízes negativas.
- Para isso utilize a função `error`:

```
1 error "Raízes negativas."
```

Resposta 4

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
  | 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
```

UFABC

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
```

UFABC

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
```

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.

```
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:

```
-- 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:

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

ou

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

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

```
> mod 10 3
1
> 10 `mod` 3
1
```

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
```

Operadores

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:

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

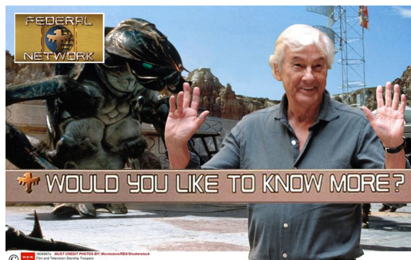
UFABC

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

```
(&&&) :: Bool -> Bool -> Bool
True &&& True = True
_    &&& _    = 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

UFABC

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

UFABC

- **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:

```
1  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!

UFABC

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

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

Criando listas

UFABC

Faixa de valores inclusivos:

```
[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:

```
[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:

```
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 - `!!`

UFABC

- 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!

Recuperar elementos - head

UFABC

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

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

UFABC

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]
```

UFABC

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

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

Exercício 7

UFABC

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`:

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

- 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 `++`

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

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

```
1   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* `_`.

Implementando a função nulo

UFABC

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

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

Exercício 9

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

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`:

```
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:

```
> [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.*

UFABC

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:

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

UFABC

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:

```
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`):

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

Exemplo: concat

Ela pode ser definida utilizando compreensão de listas:

```
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.

```
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`.

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

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?

```haskell
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?

```
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`.

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

```haskell
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:

```haskell
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)]
```

Função pairs

A assinatura será:

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

E a definição será:

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

Função and

A função `and` recebe uma lista de `Bool` e devolve `True` se todos os elementos são `True` e `False` caso contrário.

```
1  and :: [Bool] -> Bool
2  and []         = True
3  and (True:xs)  = and xs
4  and _          = False
```

Também existe a função `or` que devolve verdadeiro se ao menos um dos elementos da lista for verdadeiro.

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
```

Exercício 12 - Solução

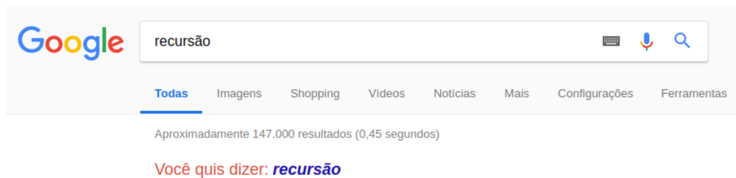
UFABC

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

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

Recursão

UFABC



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

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

- Caso base:

$$1! = 0! = 1$$

UFABC

- Para $n = 3$:

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

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

UFABC

```
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:

```
> 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:

```
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
```

- Note que a pilha tem tamanho constante.
- Em outras linguagens isto é chamado de recursão de cauda (*en: tail recursion*).

Recursão em Listas

UFABC

- 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 = ???
```

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 = (head ns) + sum (tail ns)
```

- Por que não usar Pattern Matching?

UFABC

- 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

UFABC

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
```

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`?

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

Tamanho

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.
- Em breve 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:

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

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
```

Exercício 15

UFABC

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

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

Exercício 15 - Resposta

```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:

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

Exercício 16

UFABC

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

```haskell
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:

```haskell
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]
```

Para saber mais

UFABC

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

Funções de alta ordem

Funções de alta ordem

UFABC

- As funções que...
  - recebem uma ou mais funções como argumento, ou
  - retornam uma função
- ... são denominadas **Funções de alta ordem** (*high order functions*).
- O uso de funções de alta ordem permitem aumentar a expressividade do Haskell quando confrontamos padrões recorrentes.

- Considerem a função `duasVezes` que recebe uma função `f` e um argumento qualquer `x` e aplica `f` em `x` duas vezes seguidas

```
duasVezes :: (a -> a) -> a -> a
duasVezes f x = f (f x)
```

Essa função é aplicável em diversas situações:

```
> duasVezes (*2) 3
12

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

Aplicação parcial

UFABC

- O Haskell permite que uma função de $n > 1$ argumentos seja aplicada parcialmente definindo uma função com $m < n$ argumentos:

```haskell
soma x y z = x + y + z

soma2 x y  = soma 2

soma23 x   = soma2 3

> soma2 1 2
5

> soma23 1
6
```

Funções com funções

- Com isso podemos criar novas funções a partir da nossa função `duasVezes`:

```
1  quadruplica = duasVezes (*2)
```

UFABC

- Considere o padrão

```haskell
[f x | x <- xs]
```

- Que é muito comum quando queremos gerar uma lista de números ao quadrado, somar um aos elementos de uma lista, etc.

UFABC

- Podemos definir a função `map` como:

```haskell
map :: (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs]
```

- Uma função que transforma uma lista do tipo `a` para o tipo `b` utilizando uma função `f :: a -> b`.

- `map` nos dá um novo vocábulo que deixa mais clara a interpretação das transformações feitas em listas

```
> map (+1) [1,2,3]
[2,3,4]

> map even [1,2,3]
[False, True, False]

> map reverse ["ola", "mundo"]
["alo", "odnum"]
```

Observações sobre a função `map`

UFABC

- `map` funciona para listas genéricas, de qualquer tipo
- `map` também funciona para qualquer função `f : : a -> b`
- Logo ela pode ser aplicada a ela mesma, ou seja, aplicável em listas de listas

```
1  > map (map (+1)) [[1,2],[3,4]]
2    => [ map (+1) xs | xs <- [[1,2],[3,4]] ]
3    => [ [x+1 | x <- xs] | xs <- [[1,2],[3,4]] ]
```

Exercício 17

UFABC

- Implemente uma versão recursiva de `map`.

UFABC

- Implemente uma versão recursiva de `map`.

```
mapRec :: (a -> b) -> [a] -> [b]
mapRec _ [] = []
```

UFABC

- Implemente uma versão recursiva de `map`.

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

Filter

UFABC

- Outro padrão recorrente observado é a filtragem de elementos utilizando guards nas listas

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

> [x | x <- [1..10], primo x]
[2,3,5,7]
```

Filter

UFABC

- Podemos definir a função de alta ordem `filter` da seguinte forma:

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]
```

- `filter` retorna uma lista de todos os valores cujo o predicado `p` de `x` retorna `True`.

UFABC

- Reescrevendo os exemplos anteriores:

```
> filter even [1..10]
[2,4,6,8,10]

> filter primo [1..10]
[2,3,5,7]
```

UFABC

- Também podemos passar funções parciais como argumento:

```
1  > filter (>5) [1..10]
2  [6,7,8,9,10]
3
4  > filter (/= ' ') "abc def ghi"
5  "abcdefghi"
```

Pergunta

Porque o exemplo acima `(>5)` funciona?

UFABC

- As duas funções `map` e `filter` costumam serem utilizadas juntas, assim como na compreensão de listas:

```
1  somaQuadPares :: [Int] -> Int
2  somaQuadPares ns = sum [n^2 | n <- ns, even n]
3
4  somaQuadPares :: [Int] -> Int
5  somaQuadPares ns = sum (map (^2) (filter even ns))
```

Operador pipe ($)

UFABC

- Podemos utilizar o operador $ para separar as aplicações das funções e remover os parênteses

```
1  > :t ($)
2  ($) :: (a -> b) -> a -> b
```

```
1  somaQuadPares :: [Int] -> Int
2  somaQuadPares ns = sum
3                   $ map (^2)
4                   $ filter even ns
```

- Diferentemente do pipe do Unix, aqui a execução é de baixo para cima.

Outras funções úteis durante o curso:

```
> all even [2,4,6,8]
True

> any odd [2,4,6,8]
False

> takeWhile even [2,4,6,7,8]
[2,4,6]

> dropWhile even [2,4,6,7,8]
[7,8]
```

Folding

Função foldr

Vamos recapitular algumas funções recursivas que escrevemos

```
sum []     = 0
sum (x:xs) = x + sum xs

product []     = 1
product (x:xs) = x * product xs

length []     = 0
length (_:xs) = 1 + length xs
```

Função foldr

UFABC

- Podemos generalizar essas funções da seguinte forma:

```
f [] = v
f (x:xs) = g x (f xs)
```

Função foldr

Essa função é chamada de **foldr**:

```
1  foldr :: (a -> b -> b) -> b -> [a] -> b
2  foldr f v [] = v
3  foldr f v (x:xs) = f x (foldr f v xs)
```

Função foldr

- O nome dessa função significa dobrar, pois ela justamente dobra a lista aplicando a função `f` em cada elemento da lista e um resultado parcial.

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

Função `foldr`

- Pense nessa lista não-recursivamente a partir da definição de listas:

```
a1 : (a2 : (a3 : []))
```

- Trocando `:` pela função `f` e `[]` pelo valor `v`:

```
a1 `f` (a2 `f` (a3 `f` v))
```

UFABC

Ou seja:

```
foldr (+) 0 [1,2,3]
```

se torna:

```
1 + (2 + (3 + 0))
```

Que é nossa função `sum`:

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

Pergunta

Cadê o parâmetro da função `sum`?

Exercício 18

UFABC

Defina `product` utilizando `foldr`.

UFABC

```haskell
product = foldr (*) 1
```

Função `length`

Como podemos implementar `length` utilizando `foldr`?

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

Para a lista:

```
1 : (2 : (3 : []))
```

devemos obter:

```
1 + (1 + (1 + 0))
```

UFABC

Da assinatura de `foldr`:

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

Percebemos que na função `f` o primeiro argumento é um elemento da lista e o segundo é o valor acumulado.

Função `length`

- Dessa forma podemos utilizar a seguinte função anônima:

```haskell
length = foldr (\_ n -> 1+n) 0
```

Pergunta

`length = foldr (+1) 0` funciona?

Exercício 19

UFABC

Reescreva a função `reverse` utilizando `foldr`:

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

```haskell
1 : (2 : (3 : []))
  => (([] ++ [3]) ++ [2]) ++ [1]

snoc x xs = xs ++ [x]
reverse = foldr snoc []
```

- Uma recursão caudal é quando a chamada recursiva contém apenas a chamada para a função e nada mais

```haskell
recNaoCaudal x   = x + recNaoCaudal (x-1)

recCaudal x res = recCaudal (x-1) (res+x)
```

- A função de somatória de uma lista em sua versão caudal pode ser escrita como:

```
sum :: Num a => [a] -> a
sum ns = sum' 0 ns
  where
    sum' v [] = v
    sum' v (x:xs) = sum' (v+x) xs
```

Função foldl

Esse padrão é capturado pela função `foldl`:

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

- Da mesma forma podemos pensar em `foldl` não recursivamente invertendo a lista:

```
1 : (2 : (3 : []))
  => (([] : 1) : 2) : 3
  => ((0 + 1) + 2) + 3
```

- Quando `f` é associativa, ou seja, os parênteses não fazem diferença, a aplicação de `foldr` e `foldl` não se altera:

```
sum = foldl (+) 0

product = foldl (*) 1
```

UFABC

Como ficaria a função `length` utilizando `foldl`?

```
1 length = foldr (\_ n -> 1+n) 0
2 length = foldl (??) 0
```

Função `length`

Basta inverter a ordem dos parâmetros:

```
length = foldr (\_ n -> 1 + n) 0
length = foldl (\n _ -> n + 1) 0
```

UFABC

E a função **reverse**?

```
snoc x xs = xs ++ [x]
reverse = foldr snoc []
```

UFABC

```haskell
1 : (2 : (3 : []))
  => (([] f 3) f 2) f 1

f xs x = ???
reverse = foldl f []
```

UFABC

```haskell
1 : (2 : (3 : []))
  => (([] f 3) f 2) f 1

f xs x = x:xs
reverse = foldl f []

-- ou se quiser usar uma lambda
reverse = foldl (\xs x -> x:xs) []

-- ou usando a função flip
-- flip :: (a -> b -> c) -> b -> a -> c
reverse = foldl (flip (:)) []
```

`foldr` vs `foldl`

UFABC

Uma regra do *dedão* para trabalharmos por enquanto é:

- Se a lista passada como argumento é infinita, use `foldr`
- Se o operador utilizado pode gerar curto-circuito, use `foldr`
- Se a lista é finita e o operador não irá gerar curto-circuito, use `foldl`
- Se faz sentido trabalhar com a lista invertida, use `foldl`

(na verdade, ao invés de `foldl` devemos utilizar `foldl'` que é a versão não preguiçosa)

Composição de funções

Composição de funções

UFABC

- Na matemática a composição de função $f \circ g$ define uma nova função z tal que $z(x) = f(g(x))$.
- No Haskell temos o operador `(.)`:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

UFABC

- A assinatura da função `(.)` merece um pouco mais de atenção
- Dada uma função que mapeia do tipo `b` para o tipo `c`, e outra que mapeia do tipo `a` para o tipo `b`, gere uma função que mapeie do tipo `a` para o tipo `c`.

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

- A composição de função é associativa:

```
(f . g) . h == f . (g . h)
```

UFABC

- E tem um elemento neutro que é a função `id`:

```
1  f . id = id . f = f
```

UFABC

- Essas duas propriedades são importantes durante a construção de programas, pois elas permitem o uso do `foldr` (e dentre outras funções de alta ordem):

```
1  -- cria uma função que é a composição de uma lista de
   ↪  funções
2  compose :: [a -> a] -> (a -> a)
3  compose = foldr (.) id
```

($) vs. (.)

UFABC

```
> :t ($)
($) :: (a -> b) -> a -> b
> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

- Observe o código abaixo

```
f x = sin $ abs x
g = sin . abs
h x = (sin . abs) x
i x = sin . abs $ x
```

Pergunta

Quais são os tipos de `f`, `g`, `h` e `i`?

UFABC

- Altere a função abaixo para utilizar o operador `(.)` no lugar do pipe `($)`.

```haskell
somaQuadPares :: [Int] -> Int
somaQuadPares ns = sum
                 $ map (^2)
                 $ filter even ns
```

UFABC

- Altere a função abaixo para utilizar o operador `(.)` no lugar do pipe `($)`.

```haskell
somaQuadPares :: [Int] -> Int
somaQuadPares = sum . (map (^2)) . filter even
```

- Funções de alta ordem
- [GH] 7
- [SGS] 4
- [ML] 6

Definindo novos tipos

Novos tipos de dados

UFABC

- A definição de novos tipos de dados, além dos tipos primitivos, permite manter a legibilidade do código e facilita a organização de seu programa.

UFABC

- A forma mais simples de definir um novo tipo é criando *apelidos* para tipos existentes:

```
1    type String = [Char]
```

- Todo nome de tipo deve começar com uma letra maiúscula. As definições de tipo podem ser encadeadas!
- Suponha a definição de um tipo que armazena uma coordenada e queremos definir um tipo de função que transforma uma coordenada em outra:

```
type Coord = (Int, Int)
type Trans = Coord -> Coord
```

UFABC

- Porém, não podemos definir tipos recursivos...

```
1  type Tree = (Int, [Tree])
```

- ... mas temos outras formas de definir tais tipos.

UFABC

- A declaração de tipos pode conter variáveis de tipo:

```haskell
type Pair a = (a, a)

type Assoc k v = [(k,v)]
```

UFABC

- Com isso podemos definir funções utilizando esses tipos:

```haskell
find :: Eq k => k -> Assoc k v -> v
find k t = head [v | (k',v) <- t, k == k']

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

Exercício 21

UFABC

- Crie uma função `paraCima` do tipo `Trans` definido anteriormente que ande para cima dado uma coordenada (some +1 em y).

```
type Coord = (Int, Int)
type Trans = Coord -> Coord
```

Exercício 21 - Resposta

```haskell
type Coord = (Int, Int)
type Trans = Coord -> Coord

paraCima :: Trans
paraCima (x,y) = (x, y + 1)
```

- Como esses tipos são apenas apelidos, eu posso fazer:

```
array = [(1,3), (5,4), (2,3), (1,1)] :: [(Int, Int)]
> find 2 array
3

array' = [(1,3), (5,4), (2,3), (1,1)] :: Assoc Int Int
> find 2 array
3
```

- O compilador não distingue um do outro.

Tarefa para casa

Para entregar

UFABC

- Lista 1
  - Prazo: 14/03
  - Link para submissão: XXXXXXXXXXXXXXXXX
- Não deixe de escolher o seu nome na lista para que possamos relacionar o usuário GitHub à você!

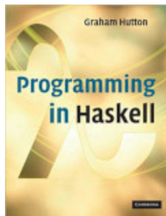
Referências

Livros

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

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

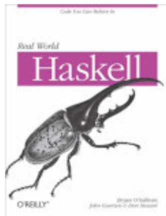
A primeira edição (antiga), que tem boa parte do conteúdo da segunda edição, está disponível na biblioteca:



- Link Biblioteca: http://biblioteca.ufabc.edu.br/index.php?codigo_sophia=15287

Livros

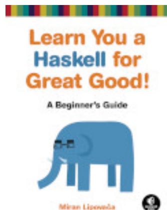
- [SGS]



- 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/`