Introdução a Programação Funcional com Haskell

Aplicações em data science e na web

Dia 02

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.

Funções de alta ordem

- As funções que...
  - recebem uma ou mais funções como argumento, ou
  - devolvem 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.

Funções com funções

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

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

UFABC

Essa função é aplicável em diversas situações:

```
> duasVezes (*2) 3
12

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

Aplicação parcial

- O Haskell permite que uma função de $n > 1$ argumentos seja aplicada

parcialmente definindo uma função com $m < n$ argumentos:

```
1   soma x y z = x + y + z
2
3   soma2 x y  = soma 2
4
5   soma23 x   = soma2 3
6
7   > soma2 1 2
8   5
9
10  > soma23 1
11  6
```

Funções com funções

UFABC

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

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

- Considere o padrão

```
[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.

Map

- Podemos definir a função `map` como:

```
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`.

UFABC

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

```
1  > map (+1) [1,2,3]
2  [2,3,4]
3
4  > map even [1,2,3]
5  [False, True, False]
6
7  > map reverse ["ola", "mundo"]
8  ["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 1

UFABC

- Implemente uma versão recursiva de `map`.

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]
```

UFABC

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

```
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`.

Filter

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?

- 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 ($)

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

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

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

```
1  sum []     = 0
2  sum (x:xs) = x + sum xs
3
4  product []     = 1
5  product (x:xs) = x * product xs
6
7  length []     = 0
8  length (_:xs) = 1 + length xs
```

Função foldr

- Podemos generalizar essas funções da seguinte forma:

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

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.

```
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`

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

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

UFABC

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

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

Função foldr

Ou seja:

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

se torna:

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

Que é nossa função `sum`:

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

Pergunta

Cadê o parâmetro da função `sum`?

Defina `product` utilizando `foldr`.

Como podemos implementar `length` utilizando `foldr`?

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

Para a lista:

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

devemos obter:

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

Da assinatura de `foldr`:

```
1 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.

UFABC

- Dessa forma podemos utilizar a seguinte função anônima:

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

Pergunta

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

Exercício 3

UFABC

Reescreva a função `reverse` utilizando `foldr`:

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

Folding caudal

- Na aula passada implementamos diversas funções em sua versão caudal

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

Função foldl

UFABC

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
```

Função foldl

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

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

Função `foldl`

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

```
1  sum = foldl (+) 0
2
3  product = foldl (*) 1
```

Função `length`

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:

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

E a função **reverse**?

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

Resposta

UFABC

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

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

Resposta

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 (:)) []
```

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

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 `(.)`:

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

Composição de funções

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**.

```
1 (.) :: (b -> c) -> (a -> b) -> (a -> c)
2 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`:

```haskell
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. (.)

```
1  > :t ($)
2  ($) :: (a -> b) -> a -> b
3  > :t (.)
4  (.) :: (b -> c) -> (a -> b) -> a -> c
```

- Observe o código abaixo

```
1  f x = sin $ abs x
2  g = sin . abs
3  h x = (sin . abs) x
4  i x = sin . abs $ x
```

Pergunta

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

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

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

Para saber mais

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

Definindo novos tipos

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

Declaração de tipo

UFABC

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

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

Declaração de tipo

UFABC

- 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:

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

UFABC

- Porém, não podemos definir tipos recursivos...

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

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

UFABC

- A declaração de tipos pode conter variáveis de tipo:

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

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

- Com isso podemos definir funções utilizando esses tipos:

```
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 5

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

- 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.

Tipos de Dados Algébricos

Tipos de Dados Algébricos

UFABC

- Tipos completamente novos.
- Pode conter tipos primitivos.
- Permite expressividade.
- Permite checagem em tempo de compilação

Tipos Soma

Tipo soma:

```haskell
data Bool = True | False
```

- `data`: declara que é um novo tipo
- `Bool`: nome do tipo
- `True | False`: poder assumir ou True ou False

Exemplo

Vamos criar um tipo que define a direção que quero andar:

```
data Dir = Norte | Sul | Leste | Oeste
```

Com isso podemos criar a função `para`:

```haskell
data Dir = Norte | Sul | Leste | Oeste

para :: Dir -> Trans
para Norte (x,y) = (x, y + 1)
para Sul   (x,y) = (x, y - 1)
para Leste (x,y) = (x + 1, y)
para Oeste (x,y) = (x - 1, y)
```

UFABC

E a função `caminhar`:

```
1  caminhar :: [Dir] -> Trans
2  caminhar []     coord = coord
3  caminhar (d:ds) coord = caminhar ds (para d coord)
```

Tipos Produto

- Tipo produto:

```haskell
data Ponto = Ponto Double Double
```

- `data`: declara que é um novo tipo
- `Ponto`: nome do tipo
- `Ponto`: construtor (ou envelope) - declaração implícita de uma função usada para criar um valor do tipo `Ponto`
- `Double Double`: tipos que ele encapsula

Atenção

O nome do tipo e o nome do construtor apesar de PODEREM ser os mesmos NÃO são a mesma coisa.

UFABC

Para ser possível imprimir esse tipo:

```haskell
data Ponto = Ponto Double Double
                      deriving (Show)
```

- `deriving`: derivado de outra classe
- `Show`: tipo imprimível
- Isso faz com que o Haskell crie automaticamente uma instância da função `show` para esse tipo de dado.

UFABC

- Para usá-lo em uma função devemos sempre envelopar a variável com o construtor.

```
1  dist :: Ponto -> Ponto -> Double
2  dist (Ponto x y) (Ponto x' y') = sqrt
3                                 $ (x-x')^2 + (y-y')^2
4
5  > dist (Ponto 1 2) (Ponto 1 1)
6  1.0
```

Tipos Produto

- Podemos misturar os tipos soma e produto:

```
data Forma = Circulo Ponto Double
           | Retangulo Ponto Double Double

-- um quadrado é um retângulo com os dois lados iguais
quadrado :: Ponto -> Double
quadrado p n = Retangulo p n n
```

UFABC

- `Circulo` e `Retangulo` são funções construtoras:

```
> :t Circulo
Circulo :: Ponto -> Double -> Forma

> :t Retangulo
Retangulo :: Ponto -> Double -> Double -> Forma
```

- As declarações de tipos também podem ser parametrizados, considere o tipo `Maybe`:

```haskell
data Maybe a = Nothing | Just a
```

- A declaração indica que um tipo `Maybe a` pode não ser nada ou pode ser apenas o valor de um tipo `a`.

UFABC

- Esse tipo pode ser utilizado para ter um melhor controle sobre erros e exceções:

```haskell
-- talvez a divisão retorne um Int
safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv m n = Just (m `div` n)

safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead xs = Just (head xs)
```

UFABC

- Eses erros podem ser capturados com a expressão `case`:

```
divComErro :: Int -> Int -> Int
divComErro m n = case (safeDiv m n) of
                    Nothing -> error "divisão por 0"
                    Just x  -> x
```

- A expressão case nos permite fazer pattern matching dentro do código da função com quaisquer expressões e não apenas nos seus parâmetros

Either

- Um outro tipo interessante é o `Either` definido como:

```
data Either a b = Left a | Right b
```

- Esse tipo permite que uma função retorne dois tipos diferentes, dependendo da situação.

UFABC

```haskell
-- ou retorna uma String ou um Int
safeDiv' :: Int -> Int -> Either String Int
safeDiv' _ 0 = Left "divisão por 0"
safeDiv' m n = Right (m `div` n)

> safeDiv' 2 2
1
> safeDiv' 2 0
"divisão por 0"
```

Exercício 6

UFABC

- Crie um tipo `Fuzzy` que pode ter os valores `Verdadeiro`, `Falso`, `Pertinencia Double`, que define um intermediário entre `Verdadeiro` e `Falso`.
- Crie uma função `fuzzifica` que recebe um `Double` e retorna `Falso` caso o valor seja menor ou igual a 0, `Verdadeiro` se for maior ou igual a 1 e `Pertinencia v` caso contrário.

Newtype

UFABC

- Uma terceira forma de criar um novo tipo é com a função `newtype`, que permite apenas um construtor:

```
1  newtype Nat = N Int
```

- A diferença entre `newtype` e `type` é que o primeiro define um novo tipo enquanto o segundo é um sinônimo.
- A diferença entre `newtype` e `data` é que o primeiro define um novo tipo até ser compilado, depois ele é substituído como um sinônimo. Isso ajuda a garantir a checagem de tipo em tempo de compilação.

Tipos Recursivos

- Um exemplo de tipo recursivo é a árvore binária, que pode ser definida como:

```
1  data Tree a = Leaf a | Node (Tree a) a (Tree a)
```

- Ou seja, ou é um nó folha contendo um valor do tipo a, ou é um nó contendo uma árvore à esquerda, um valor do tipo a no meio e uma árvore à direita.

- Desenhe a seguinte árvore:

```
t :: Tree Int
t = Node (Node (Leaf 1) 3 (Leaf 4)) 5
         (Node (Leaf 6) 7 (Leaf 9))
```

Podemos definir uma função `contem` que indica se um elemento `x` está contidado em uma árvore `t`:

```
contem :: Eq a => Tree a -> a -> Bool
contem (Leaf y) x     = x == y
contem (Node l y r) x = x == y || l `contem` x
                               || r `contem` x

> t `contem` 5
True
> t `contem` 0
False
```

Exercício 7

UFABC

- Altere a função `contem` levando em conta que essa é uma árvore de busca, ou seja, os nós da esquerda são menores ao nó atual, e os nós da direita são maiores.

Classes de Tipo

- Aprendemos em uma aula anterior sobre as classes de tipo, classes que definem grupos de tipos que devem conter algumas funções especificadas.
- Para criar uma nova classe de tipos utilizamos a palavra reservada `class`:

```
class Eq a where
    (==), (/=) :: a -> a -> Bool

    x /= y = not (x == y)
```

- Essa declaração diz: *para um tipo* `a` *pertencer a classe* `Eq` *deve ter uma implementação das funções* `(=`*)= e* `(/`*)=.*

```
1  class Eq a where
2     (==), (/=) :: a -> a -> Bool
3
4     x /= y = not (x == y)
```

UFABC

- Além disso, ela já define uma definição padrão da função (/)=, então basta definir (=)=.

```
1  class Eq a where
2     (==), (/=) :: a -> a -> Bool
3
4     x /= y = not (x == y)
```

UFABC

- Para definirmos uma nova instância de uma classe basta declarar:

```
instance Eq Bool where
    False == False = True
    True  == True  = True
    _     == _     = False
```

- Apenas tipos definidos por `data` e `newtype` podem ser instâncias de alguma classe.

UFABC

- Uma classe pode estender outra para forma uma nova classe. Considere a classe `Ord`:

```
1  class Eq a => Ord a where
2     (<), (<=), (>), (>=) :: a -> a -> Bool
3     min, max             :: a -> a -> a
4
5     min x y | x <= y     = x
6             | otherwise = y
7
8     max x y | x <= y     = y
9             | otherwise = x
```

- Ou seja, antes de ser uma instância de `Ord`, o tipo deve ser **também** instância de `Eq`.

Instância de Ord

Seguindo nosso exemplo de Booleano, temos:

```
instance Ord Bool where
    False < True = True
    _     < _    = False

    b <= c = (b < c) || (b == c)
    b > c  = c < b
    b >= c = c <= b
```

Lembrando:

- Tipo: coleção de valores relacionados.
- Classe: coleção de tipos que dão suporte a certas funções ou operadores.
- Métodos: funções requisitos de uma classe.
- Instância: um tipo que pertence a uma determinada classe

UFABC

Tipos que podem ser comparados em igualdade e desigualdade:

```
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
```

Eq - classe da igualdade

UFABC

```
> 1 == 2
False
> [1,2,3] == [1,2,3]
True
> "Ola" /= "Alo"
True
```

Ord - classe de ordem

A classe Eq acrescido de operadores de ordem:

```haskell
(<) :: a -> a -> Bool
(<=) :: a -> a -> Bool
(>) :: a -> a -> Bool
(>=) :: a -> a -> Bool
min :: a -> a -> a
max :: a -> a -> a
```

Ord - classe de ordem


```
1  > 4 < 6
2  > min 5 0
3  > max 'c' 'h'
4  > "Ola" <= "Olaf"
```

Show - classe imprimíveis

UFABC

A classe `Show` define como imprimir um valor de um tipo:

```
show :: a -> String
```

```
1  > show 10.0
2  > show [1,2,3,4]
```

A classe `Read` define como ler um valor de uma String:

```
read :: String -> a
```

Read - classe legíveis

UFABC

Precisamos especificar o tipo que queremos extrair da String:

```
1  > read "12.5" :: Double
2  > read "False" :: Bool
3  > read "[1,3,4]" :: [Int]
```

Num - classe numérica

UFABC

A classe Num define todos os tipos numéricos:

```
(+) :: a -> a -> a
(-) :: a -> a -> a
(*) :: a -> a -> a
negate :: a -> a
abs :: a -> a
signum :: a -> a
fromInteger :: Integer -> a
```

```
1  > 1 + 3
2  > 6 - 9
3  > 12.3 * 5.6
4  > 3 * (-2)
```

- Valores negativos devem ser escritos entre parênteses para remover ambiguidades com o operador de subtração.

Integral - classe de números inteiros

A classe Integral define todos os tipos numéricos inteiros:

```haskell
quot :: a -> a -> a
rem :: a -> a -> a
div :: a -> a -> a
mod :: a -> a -> a
quotRem :: a -> a -> (a, a)
divMod :: a -> a -> (a, a)
toInteger :: a -> Integer
```

```
> 10 `quot` 3
> 10 `rem` 3
> 10 `div` 3
> 10 `mod` 3
```

- As funções `quot` e `rem` arredondam para o 0, enquanto `div` e `mod` para o infinito negativo.

- A classe `Fractional` define todos os tipos numéricos fracionários

```
1 (/) :: a -> a -> a
2 recip :: a -> a
```

Fractional - classe de números inteiros

```
1  > 10 / 3
2  > recip 10
```

UFABC

Qual a diferença entre esses dois operadores de exponenciação?

```
(^) :: (Num a, Integral b) => a -> b -> a
(**) :: Floating a => a -> a -> a
```

Floating - classe de números de ponto flutuante

```
1  class Fractional a => Floating a where
2    pi :: a
3    exp :: a -> a
4    log :: a -> a
5    sqrt :: a -> a
6    (**) :: a -> a -> a
7    logBase :: a -> a -> a
```

Floating - classe de números de ponto flutuante

```
sin :: a -> a
cos :: a -> a
tan :: a -> a
asin :: a -> a
acos :: a -> a
atan :: a -> a
sinh :: a -> a
cosh :: a -> a
tanh :: a -> a
asinh :: a -> a
acosh :: a -> a
atanh :: a -> a
```

Info

- No ghci, o comando `:info` mostra informações sobre os tipos e as classes de tipo:

```
> :info Integral
class (Real a, Enum a) => Integral a where
  quot :: a -> a -> a
  rem :: a -> a -> a
  div :: a -> a -> a
  mod :: a -> a -> a
  quotRem :: a -> a -> (a, a)
  divMod :: a -> a -> (a, a)
  toInteger :: a -> Integer
  {-# MINIMAL quotRem, toInteger #-}
```

UFABC

- No ghci, o comando `:info` mostra informações sobre os tipos e as classes de tipo:

```
1  > :info Bool
2  data Bool = False | True     -- Defined in ‘GHC.Types’
3  instance Eq Bool -- Defined in ‘GHC.Classes’
4  instance Ord Bool -- Defined in ‘GHC.Classes’
5  instance Show Bool -- Defined in ‘GHC.Show’
6  instance Read Bool -- Defined in ‘GHC.Read’
7  instance Enum Bool -- Defined in ‘GHC.Enum’
8  instance Bounded Bool -- Defined in ‘GHC.Enum’
```

- Em muitos casos o Haskell consegue inferir as instâncias das classes mais comuns, nesses casos basta utilizar a palavra-chave `deriving` ao definir um novo tipo:

```haskell
data Bool = False | True
               deriving (Eq, Ord, Show, Read)
```

- Implementa as funções:

```
succ, pred, toEnum, fromEnum
```

UFABC

```haskell
data Dias = Dom | Seg | Ter | Qua | Qui | Sex | Sab
                   deriving (Show, Enum)
```

Enum é enumerativo:

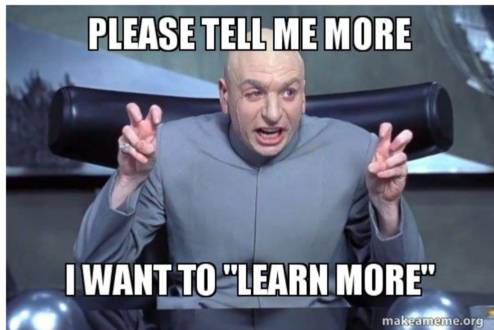
```haskell
succ Seg == Ter
pred Ter  == Seg
fromEnum Seg == 0
toEnum 1 :: Dias == Ter
-- E pode-se fazer
[Seg .. Sex] == [Seg, Ter, Qua, Qui, Sex]
```

Exercício 8

UFABC

- Defina um tipo para jogar o jogo Pedra, Papel e Tesoura e defina as funções `ganhaDe`, `perdeDe`.
- Defina também uma função denominada `ganhadores` que recebe uma lista de jogadas e retorna uma lista dos índices das jogadas vencedoras.

UFABC



- Tipos de dados algébricos
- [GH] 8
- [SGS] 3
- [ML] 8

Exercício

Análise de dados e plots simples com Haskell.

Clique aqui para baixar a tarefa.

Tarefa para casa

Para entregar

UFABC

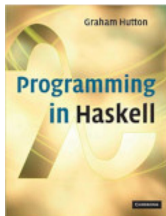
- Lista 2 e lista 2B
  - Prazo: 26/10
  - Lista 2: `https://classroom.github.com/a/yEyYld9m`
  - Lista 2B: `https://classroom.github.com/a/n3SYYixQ`
- Não deixe de escolher o seu nome na lista para que possamos relacionar o usuário GitHub à você!

Referências

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

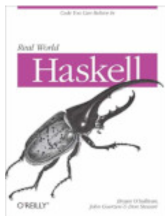
- **Programming in Haskell**. 2nd Edition.
  - Por *Graham Hutton*.

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



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

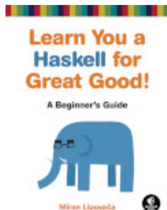
- [SGS]



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

UFABC

- [ML]



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