Introdução a Programação Funcional com Haskell

Dia 02

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

07/03/2020

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.

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

UFABC

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

UFABC

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

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

Exemplo

UFABC

Com isso podemos criar a função `para`:

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

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`:

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

data Dir = Norte | Sul | Leste | Oeste

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

UFABC

- Tipo produto:

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

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

⚠ Warning

O nome do tipo e o nome do construtor podem ser (e tipicamente são) os mesmos! Apesar de PODEREM ser os mesmos nomes, vivem em espaços de nomes diferentes e NÃO são a mesma coisa.

UFABC

Para ser possível imprimir esse tipo:

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

UFABC

- Podemos misturar os tipos soma e produto:

```haskell
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
```

Tipos parametrizados

UFABC

- 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`.

Maybe

UFABC

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

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

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

UFABC

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

```haskell
divComErro :: Int -> Int -> Int
divComErro m n = case (maybeDiv 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

UFABC

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

```haskell
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
eitherDiv' :: Int -> Int -> Either String Int
eitherDiv' _ 0 = Left "divisão por 0"
eitherDiv' m n = Right (m `div` n)

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

Exercício 1

UFABC

- Crie um tipo `Fuzzy` que pode ter os valores `Verdadeiro`; `Falso`; ou `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.

Exercício 1 - Resposta

UFABC

```haskell
data Fuzzy = Falso | Pertinencia Double | Verdadeiro

fuzzifica :: Double -> Fuzzy
fuzzifica x | x <= 0     = Falso
            | x >= 1     = Verdadeiro
            | otherwise = Pertinencia x
```

Newtype

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

```haskell
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

Árvore Binária

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

```haskell
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.

UFABC

- Desenhe a seguinte árvore:

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

UFABC

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

```
1  contem :: Eq a => Tree a -> a -> Bool
2  contem (Leaf y) x      = x == y
3  contem (Node l y r) x = x == y || l `contem` x
4                                 || r `contem` x
5
6  > t `contem` 5
7  True
8  > t `contem` 0
9  False
```

Exercício 2

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.

Exercício 2 - Resposta

UFABC

```haskell
contem :: Ord a => Tree a -> a -> Bool
contem (Leaf y) x                  = x == y
contem (Node l y r) x | x == y     = True
                      | x < y      = l `contem` x
                      | otherwise  = r `contem` x
```

Classes de Tipo

Clases de Tipo

UFABC

- 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`:

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

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

UFABC

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

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

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

UFABC

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

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

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

UFABC

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

```
1  instance Eq Bool where
2      False == False = True
3      True  == True  = True
4      _     == _     = False
```

Instâncias da Classe

UFABC

- 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`:

```haskell
class Eq a => Ord a where
   (<), (<=), (>), (>=) :: a -> a -> Bool
   min, max             :: a -> a -> a

   min x y | x <= y     = x
           | otherwise  = y

   max x y | x <= y     = y
           | 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:

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

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

UFABC

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

Eq - classe da igualdade

UFABC

Tipos que podem ser comparados em igualdade e desigualdade:

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

Eq - classe da igualdade

UFABC

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

`Ord` - classe de ordem

UFABC

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

UFABC

```
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:

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

Show - classe imprimíveis

UFABC

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

Read - classe legíveis

UFABC

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

```haskell
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:

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

Num - classe numérica

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

Tip

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

`Integral` - classe de números inteiros

UFABC

A classe `Integral` define todos os tipos numéricos inteiros:

```
1  quot :: a -> a -> a
2  rem :: a -> a -> a
3  div :: a -> a -> a
4  mod :: a -> a -> a
5  quotRem :: a -> a -> (a, a)
6  divMod :: a -> a -> (a, a)
7  toInteger :: a -> Integer
```

```
1  > 10 `quot` 3
2  > 10 `rem` 3
3  > 10 `div` 3
4  > 10 `mod` 3
```

Tip

As funções **quot** e **rem** arredondam para o 0, enquanto **div** e **mod** para $-\infty$.

`Fractional` - classe de números racionais

UFABC

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

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

`Fractional` - classe de números racionais

```
> 10 / 3
> recip 10
```

UFABC

Qual a diferença entre esses dois operadores de exponenciação?

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

`Floating` - classe de números de ponto flutuante

UFABC

```haskell
class Fractional a => Floating a where
  pi :: a
  exp :: a -> a
  log :: a -> a
  sqrt :: a -> a
  (**) :: a -> a -> a
  logBase :: a -> a -> a
```

`Floating` - classe de números de ponto flutuante

UFABC

```haskell
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
```

UFABC

- 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:

```
> :info Bool
data Bool = False | True    -- Defined in ‘GHC.Types’
instance Eq Bool -- Defined in ‘GHC.Classes’
instance Ord Bool -- Defined in ‘GHC.Classes’
instance Show Bool -- Defined in ‘GHC.Show’
instance Read Bool -- Defined in ‘GHC.Read’
instance Enum Bool -- Defined in ‘GHC.Enum’
instance Bounded Bool -- Defined in ‘GHC.Enum’
```

Derivação de instâncias

UFABC

- 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)
```

UFABC

- 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 3

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.

Exercício 3 - Resposta

UFABC

```haskell
data Jogada = Pedra | Papel | Tesoura
              deriving (Show, Enum, Eq)

ganhaDe :: Jogada -> Jogada -> Bool
Pedra   `ganhaDe` Tesoura = True
Papel   `ganhaDe` Pedra   = True
Tesoura `ganhaDe` Papel   = True
j1      `ganhaDe` j2      | j1 == j2  = True
                          | otherwise = False
```

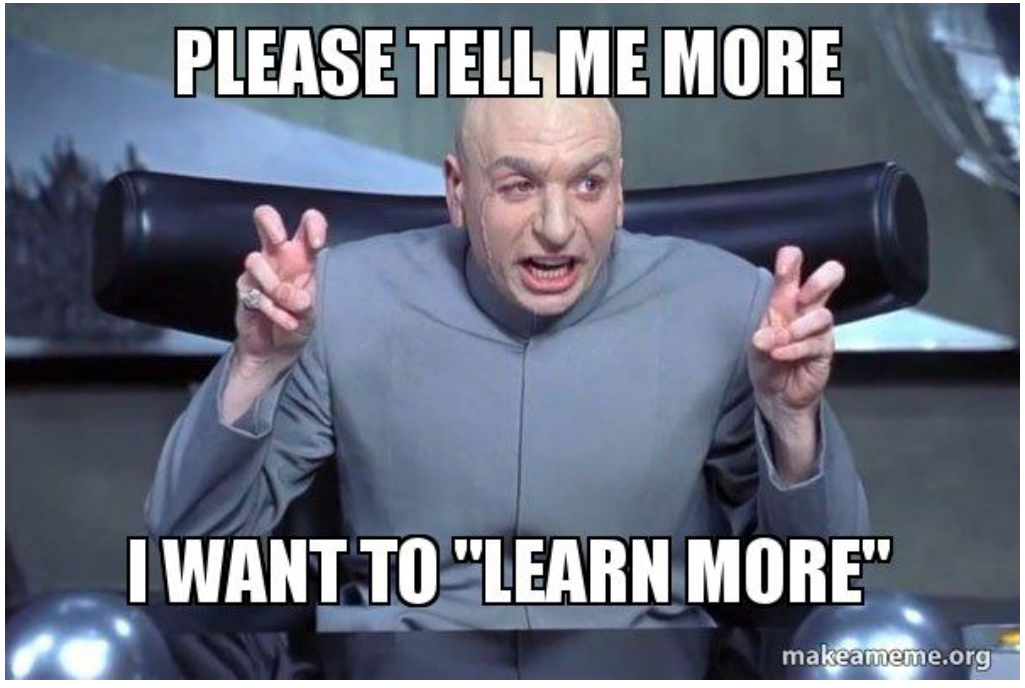
Exercício 3 - Resposta

UFABC

```haskell
data Jogada = Pedra | Papel | Tesoura
              deriving (Show, Enum, Eq)

perdeDe :: Jogada -> Jogada -> Bool
j1 `perdeDe` j2 = not $ j1 `ganhaDe` j2
```

Para saber mais...



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

Construtores de Tipos

UFABC

- Em aulas anteriores vimos o conceito de construtores de tipos, quando criamos novos tipos. Eles recebem um tipo como parâmetro e criam um novo tipo:

```haskell
listaDeDouble :: [Double]
talvezInt     :: Maybe Int
arvoreChar    :: Tree Char
```

UFABC

- Tipo paramétrico é todo tipo que possui um parâmetro de tipo:

```
1 [a], Maybe a, Tree a, ...
```

Criando funções para Tipos Paramétricos

UFABC

- Considere as seguintes funções:

```
1  dobra :: Int  -> Int
2  dobra x = 2*x
3
4  flip :: Coin -> Coin
5  flip Cara  = Coroa
6  flip Coroa = Cara
```

UFABC

- Se quisermos aplicar essas funções para uma lista desses tipos, basta:

```haskell
dobraLista :: [Int]  -> [Int]
dobraLista = map (*2)

flipLista :: [Coin] -> [Coin]
flipLista = map flip
```

UFABC

- E se quisermos aplicar essa função para um `Maybe Coin`, ou `Tree Int`?

```haskell
dobraArvore :: Tree Int -> Tree Int
dobraArvore = ??

flipMaybe :: Maybe Coin -> Maybe Coin
flipMaybe = ??
```

Functors

- Functors são *funções* que fazem com que as *funções* de um certo tipo sejam aplicáveis a um tipo paramétrico contendo esse tipo.

Functors e teoria das categorias

- Functors são morfismos que transformam os morfismos de uma categoria inteira (Tipos) em morfismos de outra (`[]`, `Maybe`, ...).
- No Haskell o que temos são endofunctors.
- Mais sobre isso no curso de teoria das categorias para programadores!

UFABC

- No Haskell um Functor é definido como uma classe de tipos:

```haskell
class Functor f where
   fmap :: (a -> b) -> f a -> f b
```

- Ou seja, se eu já tenho uma função $g : a \to b$, e tenho um tipo paramétrico `f`, eu posso aplicar a função `g` em `f a` para obter `f b`.

Tip

Em outras linguagens funcionais, e algumas imperativas, há uma operação chamada `flat map`. O `f` de `fmap` não tem nada a ver com *flat* e sim com *Functor*. Em breve veremos o equivalente ao `flat map` em outras linguagens.

Functors de lista

UFABC

- Para as listas nós já temos o functor:

```haskell
instance Functor [] where
  fmap = map
```

UFABC

- Para o Maybe definimos da seguinte forma:

```haskell
instance Functor Maybe where
  fmap _ Nothing =  Nothing
  fmap g (Just x) = Just (g x)
```

UFABC

- Agora podemos fazer:

```
1  > fmap chr Nothing
2  Nothing
3  > fmap chr (Just 65)
4  Just 'A'
5  > fmap (+1) (Just 65)
6  Just 66
```

UFABC

- Reforçando a ideia de promessa computacional, imagine que eu esteja aplicando a função `chr` em um valor proveniente de uma computação que pode falhar:

```
1 > x = (n + 36) `mod` y
2 > fmap chr x
```

- Nesse caso, se a computação de x falhar, a função não será aplicada e o programa não termina com erro.

UFABC

- Definimos um Functor de Árvores como:

```haskell
instance Functor Tree where
  fmap g (Leaf x)   = Leaf (g x)
  fmap g (Node l x r) = Node (fmap g l) (g x) (fmap g r)
```

UFABC

```
> fmap (*2) (Node (Leaf 1) 2 (Node (Leaf 3) 4 (Leaf 5))
(Node (Leaf 2) 4 (Node (Leaf 6) 8 (Leaf 10))
```

Operador Functor

UFABC

- Podemos utilizar o operador `(<$>)` no lugar do `fmap`:
- O operador `(<$>)` nada mais é que a definição de `fmap` infixa

```
> (+1) <$> [1,2,3]
[2,3,4]
> (+1) `fmap` [1,2,3]
[2,3,4]
-- Para o Functor [] é o mesmo que
> (+1) `map` [1,2,3]
[2,3,4]
> map (+1) [1,2,3]
[2,3,4]
```

Exemplo: Aritmética segura

UFABC

```
> 4 `div` 0
*** Exception: divide by zero
> maxBound :: Int
9223372036854775807
> (maxBound :: Int) + 1
-9223372036854775808
```

- Inteiros têm precisão de 64 bits.
- Caso qualquer operação resulte em um número que exija mais do que 64 bits, ocorre overfow e o número ¨*flipa*”
- Checar isto a cara operação pode ser uma tarefa muito tediosa. Vamos criar uma função!

Exemplo: Aritmética segura

UFABC

- Considere o seguinte tipo:

```haskell
data SafeNum a = NaN | NegInf| PosInf | SafeNum a
  ↪  deriving Show
```

- Este tipo serve para armazenar valores de números de maneira segura
- Os valores possíveis são:
  - `SafeNum a` - Um número propriamente dito
  - `NegInf` e `PosInf` - utilizado em caso de overflow
  - `NaN` - *Not a Number*: utilizado em caso de operações aritméticas inválidas ($\sqrt{-1}$, divisão, por 0, ...)
- Podemos então criar uma função que dado dois inteiros devolva uma versão segura do valor

Exemplo: Aritmética segura

UFABC

```haskell
safeAdd :: Int -> Int -> SafeNum Int
safeAdd x y
  | signum x /= signum y = SafeNum z
  | signum z /= signum x = if signum x > 0
                           then PosInf
                           else NegInf
  | otherwise = SafeNum z
  where z = x + y

safeDiv :: Int -> Int -> SafeNum Int
safeDiv 0 0 = NaN
safeDiv x 0
  | x > 0     = PosInf
  | otherwise = NegInf
safeDiv x y = SafeNum $ x `div` y
```

UFABC

E a partir daí podemos fazer:

```
> maxBound `safeAdd` 1
PosInf
> minBound `safeAdd` (-1)
NegInf
> 0 `safeDiv` (-1)
SafeNum 0
> (-1) `safeDiv` 0
NegInf
> 0 `safeDiv` 0
NaN
```

Exemplo: Aritmética segura

- Porem agora temos um problema. Queremos fazer: `negate (x + y)`

```
1  > x = 5
2  > y = 7
3  > z = x `safeAdd` y
4  > :t z
5  z :: SafeNum Int
6  > :t negate
7  negate :: Num a => a -> a
```

- Temos um valor dentro de um tipo paramétrico e desejamos aplicar uma função ao valor que ele contém!

Exemplo: Aritmética segura

UFABC

- Com um problema um pouquinho mais elaborado, a quantidade de boilerplate necessária para fazer uma operação que dependa de valores seguros é inaceitável. 😕

```haskell
-- Devolve (x / y) + (y / x)
-- Versão inicial
f0 :: Int -> Int -> SafeNum Int
f0 x y
  | isSafe xy && isSafe yx = safeAdd (unbox xy) (unbox
  ↪  yx) -- SafeNum Int
  | (not.isSafe) xy = xy
  | otherwise = yx -- (not.isSafe) yx
  where
    xy = safeDiv x y -- SafeNum Int
    yx = safeDiv y x -- SafeNum Int
    unbox (SafeNum x) = x
    isSafe (SafeNum _) = True
    isSafe _           = False
```

Exemplo: Aritmética segura

UFABC

Dá para fazer melhor?

Exemplo: Aritmética segura

UFABC

Dá para fazer melhor?

- Sim! Podemos transformar um `SafeNum` em um functor e usando `fmap` ter a certeza que não teremos problemas durante a execução.

Exemplo: Aritmética segura

UFABC

Se `SafeNum` fosse um functor, poderíamos fazer:

```
> z = 5 `safeAdd` 7

> :t z
z :: SafeNum Int

> :t negate
negate :: Num a => a -> a

> :t (<$>)
(<$>) :: Functor f => (a -> b) -> f a -> f b

> negate <$> z
SafeNum (-12)

> negate <$> safeDiv 10 0
PosInf
```

Exercício 4

Escreva a instância de Functor para o tipo `SafeNum`.

```haskell
data SafeNum a = NaN | NegInf| PosInf | SafeNum a
  deriving Show
```

Resposta - Exercício 4

UFABC

```haskell
boxedCoerce :: SafeNum a -> SafeNum b
boxedCoerce NaN = NaN
boxedCoerce NegInf = NegInf
boxedCoerce PosInf = PosInf
boxedCoerce _ = error "Não deveria ser usado para
↪  valores safe"

instance Functor SafeNum where
  fmap f (SafeNum n) = SafeNum $ f n
  fmap _ x = boxedCoerce x
```

Exemplo: Aritmética segura

UFABC

```haskell
flatten :: SafeNum (SafeNum a) -> SafeNum a
flatten (SafeNum sn) = sn
flatten v = boxedCoerce v

-- Versão com functors
f1 :: Int -> Int -> SafeNum Int
f1 x y =
  let xy = safeDiv x y
      yx = safeDiv y x
      -- :: SafeNum (Int -> SafeNum Int)
      safeAddXY = fmap safeAdd xy
      -- :: SafeNum (SafeNum (SafeNum Int))
      safeXYPlusYX = fmap (`fmap` yx) safeAddXY
  in
    (flatten.flatten) safeXYPlusYX
```

- Melhorou, mas ainda está ruim...

Para saber mais...

- Functors, Applicatives e Monads
- [GH] 12
- [SGS] 14
- [ML] 11, 12
- Teoria das categorias: Functors

Applicative Functors

UFABC

- Ok, digamos que eu queira fazer:

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

UFABC

- Idealmente teríamos:

```haskell
fmap0 :: a -> f a
fmap1 :: (a -> b) -> f a -> f b
fmap2 :: (a -> b -> c) -> f a -> f b -> f c
fmap3 :: (a -> b -> c -> d) -> f a -> f b -> f c -> f d
```

UFABC

- Com isso poderíamos:

```
> fmap2 (+) [1,2] [3,4]
[4,5]
> fmap2 (+) (Just 3) (Just 2)
Just 5
```

- Mas definir todas essas funções é um trabalho tedioso...

UFABC

- Podemos resolver isso através do uso de *currying*:

```haskell
pure  :: a -> f a
aplica :: f (a -> b) -> f a -> f b

fmap0 :: a -> fa
fmap0 = pure

fmap1 :: (a -> b) -> (f a -> f b)
fmap1 g x = aplica (pure g) x

fmap2 :: (a -> (b -> c)) -> (f a -> (f b -> f c))
fmap2 g x y = aplica (aplica (pure g) x) y
```

UFABC

- Este padrão é tão recorrente que ganhou um nome: Applicative Functor ou simplesmente Applicative, cuja classe de tipo é definida como:

```haskell
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

UFABC

- E com isso podemos fazer:

```
1 > pure (+) <*> [1,2] <*> [3,4] -- não dá esse resultado
2 [4,6]
3 > pure (+) <*> (Just 3) <*> (Just 2)
4 Just 5
```

- O significado de `pure` nesse contexto é a de que estamos transformando uma função pura em um determinado contexto computacional (de computação não determinística, de computação que pode falhar, etc.)

UFABC

- Para o tipo Maybe basta definirmos:

```haskell
instance Applicative Maybe where
  pure             = Just
  Nothing  <*> _   = Nothing
  (Just g) <*> mx  = fmap g mx
```

UFABC

- Essas definições nos ajudam a definir um modelo de programação em que funções puras podem ser aplicadas a argumentos que podem falhar, sem precisar gerenciar a propagação do erro:

```haskell
r1 = maybeDiv x y
r2 = maybeDiv y x

-- Se alguma divisão falhar, retorna Nothing
-- Não precisamos criar um maybeAdd!
somaResultados = pure (+) <*> r1 <*> r2
```

Maybe - Tratamento de Exceções

UFABC

```
1 > pure (+) <*> maybeDiv 1 0 <*> maybeDiv 0 1
2 Nothing
```

UFABC

- Para as listas, o uso de applicative define como aplicar um operador em todas as combinações de elementos de duas listas:

```haskell
instance Applicative [] where
  pure x    = [x]
  gs <*> xs = [g x | g <- gs, x <- xs]
```

UFABC

- Com isso temos:

```
> pure (+1) <*> [1,2,3]
[2,3,4]
> pure (+) <*> [1] <*> [2]
[3]
> pure (*) <*> [1,2] <*> [3,4]
[3,4,6,8]
```

Applicative List

UFABC

```
> pure (++) <*> ["ha","heh","hmm"] <*> ["?","!","."]
["ha?","ha!","ha.","heh?","heh!","heh."
  ,"hmm?","hmm!","hmm."]
```

Computação não-determinística

UFABC

- Imagine que queremos fazer a operação `x * y`, mas tanto `x` quanto `y` são não-determinísticos, ou seja, podem assumir uma lista de possíveis valores.
- Uma forma de tratar esse problema é através do Applicative listas que retorna todas as possibilidades:

```
1 > pure (*) <*> [1,2,3] <*> [2,3]
2 [2,3,4,6,6,9]
3 > pure (*) <*> [1,2,3] <*> []
4 []
```

UFABC

- Uma outra interpretação para o Applicative de listas é a operação elemento-a-elemento pareados. Ou seja:

```
1 -- Não dá esse resultado. Veja ZipList.
2 > pure (+) <*> [1,2,3] <*> [4,5]
3 [5,7]
```

UFABC

- Como só pode existir uma única instância para cada tipo, criaram a ZipList que é uma lista que terá essa propriedade na classe Applicative:

```
1 > import Control.Applicative
2 > pure (+) <*> ZipList [1,2,3] <*> ZipList [4,5]
3 ZipList [5,7]
```

UFABC

- Imagine que temos uma sequência de aplicações de uma função `g` a ser aplicada na ordem:

```haskell
g :: a -> Maybe a

[g x1, g x2, g x3]
```

- Na avaliação preguiçosa, quando avaliarmos uma lista cada elemento será avaliado em ordem arbitrária (dependendo da função sendo avaliada).

UFABC

- Como a sequência é importante, não queremos continuar computando no caso de falhas.
- Podemos construir uma lista de Applicative da seguinte forma:

```
1  pure (:) <*> g x1 <*>
2     (pure (:) <*> g x2 <*>
3        (pure (:) <*> g x3 <*> pure []))
```

UFABC

- Se uma aplicação falhar, não temos motivos para continuar computando, caso a aplicação `g x2` falhe, podemos retornar `Nothing` imediatamente.
- É possível generalizar essa função com:

```haskell
-- sequencia de Applicatives
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA []     = pure []
sequenceA (x:xs) = pure (:) <*> x <*> sequenceA xs
```

UFABC

```
> sequenceA [Just 3, Just 2, Just 1]
Just [3,2,1]
> sequenceA [Just 3, Nothing, Just 1]
Nothing
> sequenceA [[1,2,3],[4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
> sequenceA [[1,2,3],[4,5,6],[3,4,4],[]]
[]
```

- Sequenciamento é útil quando queremos ter controle da ordem das operações e tais operações podem gerar efeitos colaterais ou falhar. Ex.:
  - Capturar caracteres do teclado
  - Backtracking

E o `SafeNum` + `Applicative`?

UFABC

A instância de `Applicative` para `SafeNum` é bem simples:

```haskell
instance Applicative SafeNum where
  pure = SafeNum
  f <*> x = flatten $ fmap (`fmap` x) f
```

E o SafeNum + Applicative?

UFABC

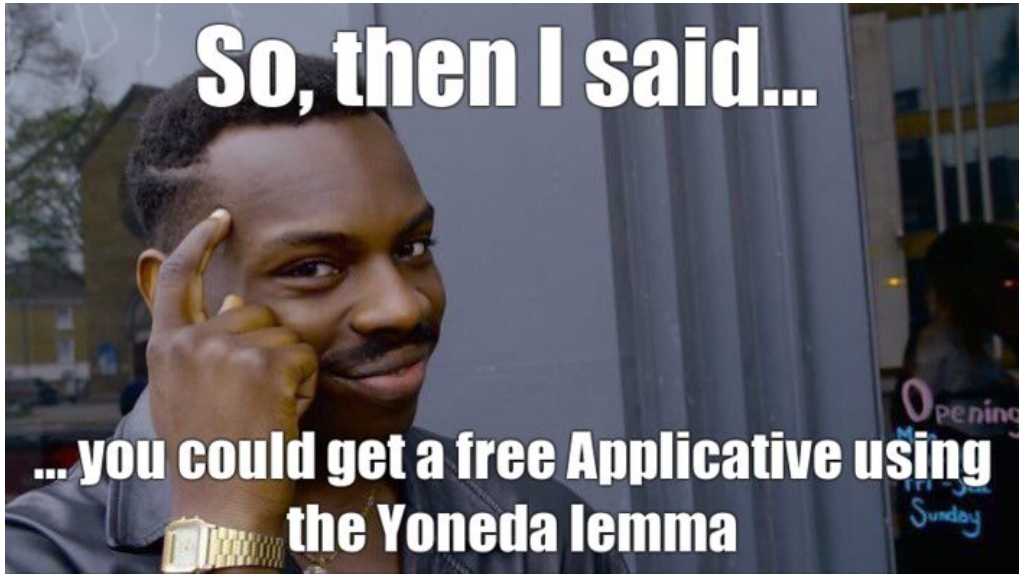
```haskell
-- Versão "pura"
f0 :: Int -> Int -> SafeNum Int
f0 x y
  | isSafe xy && isSafe yx = safeAdd
  ↪  (unbox xy) (unbox yx)
  | (not.isSafe) xy = xy
  | otherwise = yx
  where
    xy = safeDiv x y
    yx = safeDiv y x
    unbox (SafeNum a) = a
    isSafe (SafeNum _) = True
    isSafe _           = False
```

```haskell
-- Versão com functors
f1 :: Int -> Int -> SafeNum Int
f1 x y =
  let xy = safeDiv x y
      yx = safeDiv y x
      safeAddXY = fmap safeAdd xy
      safeXYPlusYX = fmap (`fmap` yx)
      ↪  safeAddXY
  in
    (flatten.flatten) safeXYPlusYX
```

```haskell
-- Versão com applicative
f2 :: Int -> Int -> SafeNum Int
f2 x y =
  let xy = safeDiv x y  -- SafeNum Int
      yx = safeDiv y x  -- SafeNum Int
  in
    flatten $ pure safeAdd <*> xy <*> yx
```

Será que dá pra fazer melhor?

Para saber mais...



- Functors, Applicatives e Monads
- [GH] 12
- [SGS] 14
- [ML] 11, 12
- Applicatives
- Lema de Yoneda

Monads

UFABC

- Vamos definir um tipo de dado que representa expressões matemáticas:

```haskell
data Expr = Val Int
          | Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
          | Div Expr Expr
```

UFABC

- Para avaliar essa expressão podemos definir:

```haskell
eval :: Expr -> Int
eval (Val n)   = n
eval (Add x y) = (eval x) + (eval y)
eval (Sub x y) = (eval x) - (eval y)
eval (Mul x y) = (eval x) * (eval y)
eval (Div x y) = (eval x) `div` (eval y)
```

UFABC

- Porém, se fizermos:

```
1 > eval (Div (Val 1) (Val 0))
2 *** Exception: divide by zero
```

UFABC

- Podemos resolver isso usando `maybeDiv` e `Maybe` (vamos focar apenas na divisão):

```haskell
eval :: Expr -> Maybe Int
eval (Val n)   = Just n
eval (Div x y) = case eval x of
                   Nothing -> Nothing
                   Just n  -> case eval y of
                                Nothing -> Nothing
                                Just m  -> maybeDiv n
                                ↪  m
```

UFABC

- Agora temos:

```
> eval (Div (Val 1) (Val 0))
Nothing
```

- Mas nosso código está confuso...

Applicative?

- O uso de Applicative pode resolver muitos problemas de encadeamento de funções com efeito
- Seria legal poder fazer:

```
1 > pure maybeDiv <*> eval x <*> eval y
```

- Mas `maybeDiv` tem tipo `Int -> Int -> Maybe Int` e deveria ser `Int -> Int -> Int` para o uso de applicativo.

- O problema aqui é que o uso de Applicative é para sequências de computações que podem ter efeitos mas que são independentes entre si.
- Queremos agora uma sequência de computações com efeito mas que uma computação dependa da anterior.

- Precisamos de uma função que capture nosso padrão de `case of`:

```haskell
vincular :: Maybe a -> (a -> Maybe b) -> Maybe b
vincular mx g = case mx of
                  Nothing -> Nothing
                  Just x  -> g x
```

- O nome significa que estamos vinculando o resultado da computação de `mx` ao argumento da função `g`.

UFABC

- No Haskell esse operador é conhecido como **bind** e definido como:

```haskell
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

- Qualquer semelhança com o logo de Haskell, é mera coincidência 😄

UFABC

- Com isso podemos reescrever `eval` como:

```haskell
eval :: Expr -> Maybe Int
eval (Val n)   = Just n
eval (Div x y) = eval x >>= \n ->
                 eval y >>= \m ->
                 maybeDiv n m
```

UFABC

```
> eval (Div (Val (Just 4)) (Val (Just 2)))
  => (Just 4) >>= \n ->
                 (Just 2) >>= \m -> maybeDiv n m
  => (Just 2) >>= \m -> maybeDiv 4 m
  => maybeDiv 4 2
```

UFABC

```
1  > eval (Div (Val (Nothing)) (Val (Just 2)))
2    => Nothing >>= \n ->
3                  (Just 2) >>= \m -> maybeDiv n m
4    => Nothing
```

UFABC

```
> eval (Div (Val (Just 4)) (Val (Nothing))
  => (Just 4) >>= \n ->
                 (Just 2) >>= \m -> maybeDiv n m
                 => Nothing >>= \m -> maybeDiv 4 m
  => Nothing
```

UFABC

- Generalizando, uma expressão construída com o operador `(>>=)` tem a seguinte estrutura:

```
1  m1 >>= \x1 ->
2  m2 >>= \x2 ->
3  ...
4  mn >>= \xn ->
5  f x1 x2 ... xn
```

- Indicando um encadeamento de computação sequencial para chegar a uma aplicação de função. Esse operador garante que se uma computação falhar, ela para imediatamente e reporta a falha (em forma de `Nothing`, `[]`, etc.)

UFABC

- Essa mesma expressão pode ser escrita com a notação chamada do-notation:

```
1  do x1 <- m1
2     x2 <- m2
3     ...
4     xn <- mn
5     f x1 x2 ... xn
```

UFABC

- Com isso podemos reescrever `eval` novamente como:

```haskell
eval :: Expr -> Maybe Int
eval (Val n)   = Just n
eval (Div x y) = do n <- eval x
                    m <- eval y
                    safeDiv n m
```

- Que captura uma sequência de computações que devem respeitar a ordem, são dependentes e podem falhar. Uma notação imperativa? 

UFABC

- Esse tipo de operação forma uma nova classe de tipos denominada Monads:

```haskell
class Applicative m => Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b

    return = pure
```

- Além do operador `bind` ela redefine a função `pure` com o nome de `return`.

UFABC

- Já escrevemos a definição de `Monad Maybe` mas podemos deixá-la mais clara utilizando Pattern Matching:

```haskell
instance Monad Maybe where
   Nothing  >>= _ = Nothing
   (Just x) >>= f = f x
```

UFABC

- Listas também fazem parte da classe Monad, inclusive já fizemos uso de *bind* para listas anteriormente:

```haskell
instance Monad [] where
    xs >>= f = [y | x <- xs, y <- f x]
```

UFABC

- Por exemplo, para gerar todas as combinações de elementos de duas listas pode ser escrito como:

```haskell
pares :: [a] -> [b] -> [(a,b)]
pares xs ys = xs >>= \x ->
              ys >>= \y ->
              return (x,y) -- [(x,y)]
```

UFABC

- Ou em *do-notation*:

```haskell
pares :: [a] -> [b] -> [(a,b)]
pares xs ys = do x <- xs
                 y <- ys
                 return (x,y)
```

Monad Lists

UFABC

```
1  > pares [1,2] [3,4]
2    => [1,2] >>= \x ->
3           [3,4] >>= \y ->
4                  [(x,y)]
5    => [x' | x  <- [1,2],
6             x' <- \x -> [3,4] >>= \y -> [(x,y)]]
7    => [x' | x  <- [1,2],
8             x' <- \x -> [y' | y <- [3,4], y' <- [(x,y)]]
9    => [x' | x  <- [1,2],
10            x' <- \x -> [y' | y' <- [(x,3), (x,4)]]
11   => [x' | x  <- [1,2],
12            x' <- \x -> [(x,3), (x,4)]]
13   => [x' | x' <- [(1,3),(1,4),(2,3),(2,4)]]
14   => [(1,3),(1,4),(2,3),(2,4)]
```

UFABC

- A compreensão de listas surgiu a partir da notação *do*:

```
1  pares xs ys = [(x,y) | x <- xs, y <- ys]
2    == do x <- xs
3          y <- ys
4          return (x,y)
```

`SafeNum` encontra uma `Monad` para chamar de sua

UFABC

```haskell
-- Versão com monad
f3' :: Int -> Int -> SafeNum Int
f3' x y =
  safeDiv x y >>= \xy ->
  safeDiv y x >>= \yx ->
  safeAdd xy yx

f3 :: Int -> Int -> SafeNum Int
f3 x y = do
  xy <- safeDiv x y
  yx <- safeDiv y x
  safeAdd xy yx
```

SafeNum - Comparação entre as versões

UFABC

```haskell
-- Versão "pura"
f0 :: Int -> Int -> SafeNum Int
f0 x y
  | isSafe xy && isSafe yx = safeAdd
  ↪  (unbox xy) (unbox yx)
  | (not.isSafe) xy = xy
  | otherwise = yx
  where
    xy = safeDiv x y
    yx = safeDiv y x
    unbox (SafeNum x) = x
    isSafe (SafeNum _) = True
    isSafe _           = False
```

```haskell
-- Versão com functors
f1 :: Int -> Int -> SafeNum Int
f1 x y =
  let xy = safeDiv x y
      yx = safeDiv y x
      safeAddXY = fmap safeAdd xy
      safeXYPlusYX = fmap (`fmap` yx)
      ↪  safeAddXY
  in
    (flatten.flatten) safeXYPlusYX
```

```haskell
-- Versão com applicative
f2 :: Int -> Int -> SafeNum Int
f2 x y =
  let xy = safeDiv x y
      yx = safeDiv y x
  in
    flatten $ pure safeAdd <*> xy <*> yx
```

```haskell
-- Versão com monad
f3 :: Int -> Int -> SafeNum Int
f3 x y = do
  xy <- safeDiv x y
  yx <- safeDiv y x
  safeAdd xy yx
```

Exercício 5

UFABC

- Escreva a instância de `Monad` para o tipo `SafeNum`.

```haskell
data SafeNum a = NaN | NegInf| PosInf | SafeNum a
  ↪  deriving Show
```

Resposta - Exercício 5

UFABC

```haskell
data SafeNum a = NaN | NegInf| PosInf | SafeNum a
  ↪  deriving Show
```

```haskell
instance Monad SafeNum where
  (SafeNum v) >>= f = f v
  v           >>= _ = boxedCoerce v
```

UFABC

- As funções de alta ordem possuem versões para Monads na biblioteca `Control.Monad`:

```haskell
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f []     = return []
mapM f (x:xs) = do y  <- f x
                   ys <- mapM f xs
                   return (y:ys)
```

UFABC

- Digamos que tenho a seguinte função:

```haskell
conv :: Char -> Maybe Int
conv c | isDigit c = Just (digitToInt c)
       | otherwise = Nothing
```

UFABC

- Podemos aplicar **mapM** para obter:

```
1 > mapM conv "1234"
2 Just [1,2,3,4]
3 > mapM conv "12a4"
4 Nothing
```

- Também temos a versão monádica de `filter`:

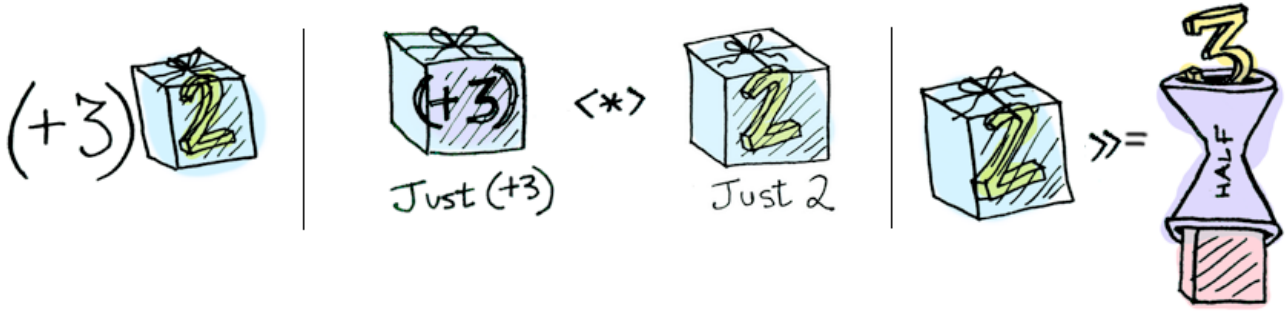
```haskell
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
filterM p []     = return []
filterM p (x:xs) = do b  <- p x
                      ys <- filterM p xs
                      return (if b then x:ys else ys)
```

UFABC

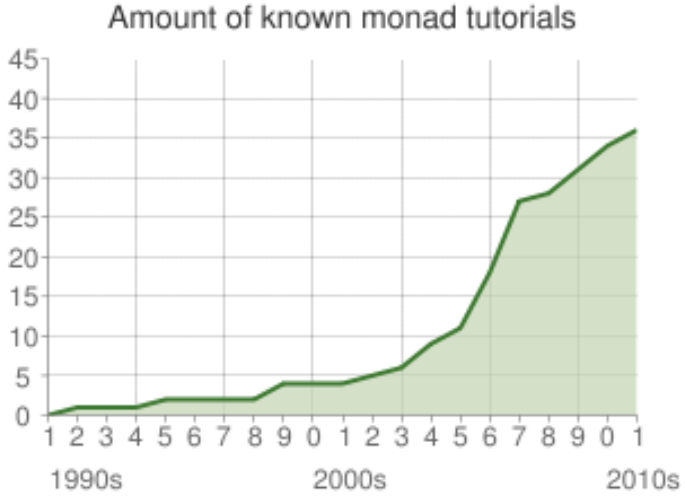
- Podemos gerar o conjunto das partes com essa função e o Monad List:

```
> filterM (\x -> [True, False]) [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

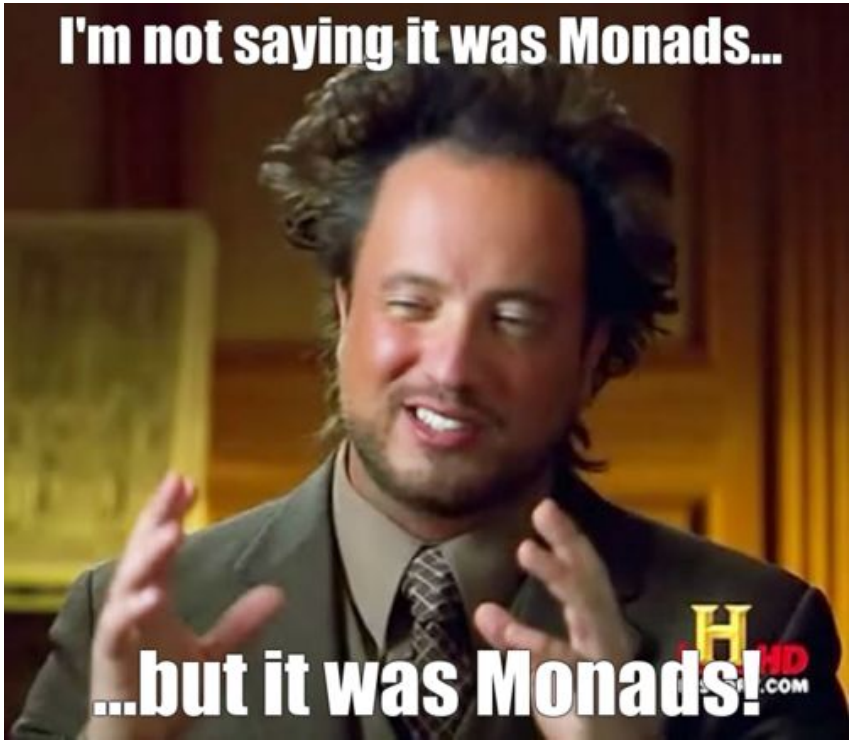
- Functors, Applicatives, And Monads In Pictures



- Abstraction, intuition, and the “monad tutorial fallacy”
- Monad tutorials timeline



Para saber mais - II



- Functors, Applicatives e Monads
- [GH] 12
- [SGS] 14
- [ML] 11, 12

State IO

UFABC

- Conforme discutimos anteriormente, funções de entrada e saída de dados são impuras pois alteram o estado atual do sistema.
- A função `getChar` captura um caracter do teclado. Se eu executar tal função duas vezes, o valor da função não necessariamente será igual.
- A função `putChar` escreve um caracter na saída padarão (ex.: monitor). Se eu executar duas vezes seguidas com a mesma entrada, a saída será diferente.

Basicamente, as funções de entrada e saída alteram estado, ou seja:

```haskell
newtype IO a = newtype ST a = State -> (a, State)
```

com a definição de estado sendo:

```haskell
type State = Environment
```

o estado sendo o ambiente, sistema operacional, o mundo computacional que ele vive.

UFABC

Com isso, tudo que fizemos até agora é suficiente para trabalharmos com IO sem afetar a pureza dos nossos programas:

```haskell
getchar :: IO Char

putChar :: Char -> IO ()
```

UFABC

Se eu fizer:

```haskell
do putChar 'a'
   putChar 'a'
```

Na verdade ele estará fazendo algo como:

```haskell
(_, env')  = putChar 'a' env
(_, env'') = putChar 'a' env'
```

Entrada e Saída - Ação

UFABC

- No Haskell chamamos as funções de entrada e saída como ações de IO (IO actions).
- As funções básicas são implementadas internamente de acordo com o Sistema Operacional

Ações básicas

Vamos trabalhar inicialmente com três ações básicas:

```haskell
-- recebe um caracter da entrada padrão
getChar :: IO Char

-- escreve um caracter na saída padrão
putChar :: Char -> IO ()

-- retorna um valor puro envolvido de uma ação IO
return :: a -> IO a
```

Funções auxiliares: `getLine`

UFABC

Em vez de capturar apenas um caracter, podemos capturar uma linha inteira de informação. Podemos escrever `getLine` da seguinte maneira:

```haskell
getLine :: IO String
getLine = do x <- getChar
             if x == '\n' then
                return []
             else
                do xs <- getLine
                   return (x:xs)
```

Atenção!

A função `return` não se comporta como em outras linguagens!

Lembre-se: `return` apenas pega um valor puro e o coloca no em um contexto. Ele não interrompe a execução.

UFABC

Escreva as instruções do **else** como Applicative

```haskell
getLine :: IO String
getLine = do x <- getChar
             if x == '\n' then
                return []
             else
                do xs <- getLine
                   return (x:xs)
```

Exercício 6 - Resposta

UFABC

```haskell
getLine :: IO String
getLine = do x <- getChar
             if x == '\n' then
                return []
             else
                pure (x:) <*> getLine
```

Funções auxiliares: `putStr`

UFABC

A função inversa escreve uma `String` na saída padrão:

```haskell
putStr :: String -> IO ()
putStr []     = return ()
putStr (x:xs) = do putChar x
                   putStr  xs

putStrLn :: String -> IO ()
putStrLn xs = do putStr xs
                 putChar 'n'
```

UFABC

Escreva a função **putStrLn** usando Applicative.

```haskell
putStrLn :: String -> IO ()
putStrLn xs = do putStr xs
                 putChar '\n'
```

Exercício 7 - Resposta

UFABC

```haskell
putStrLn :: String -> IO ()
putStrLn xs = do sequenceA [putStr' xs, putChar 'n']
                 return ()
```

Para saber mais

- [GH] 14
- [SGS] 13, 14, 15
- [ML] 11
- State IO e Leitura de Arquivos

Tarefa para casa

Para entregar

UFABC

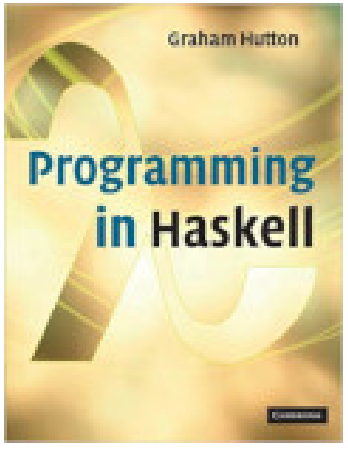
- Lista 2 no Github Classroom
  - Prazo: 21/03
- Não deixe de escolher o seu nome na lista para que possamos relacionar o usuário GitHub à você!

Referências

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

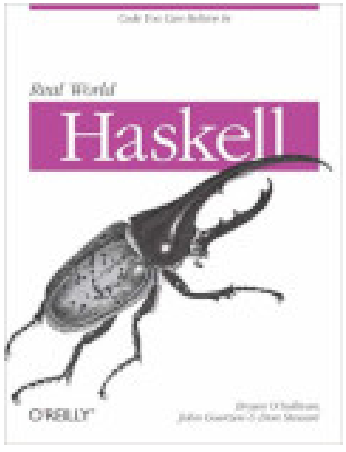
- **Programming in Haskell**. 2nd Edition.
  - Por *Graham Hutton*.

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`

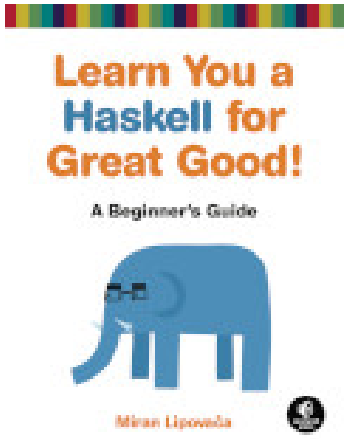
- [SGS]



- 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/`