Introdução a Programação Funcional com Haskell

Aplicações em data science e na web

Dia 03

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.

Construtores de Tipos

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:

```haskell
dobra :: Int  -> Int
dobra x = 2*x

flip :: Coin -> Coin
flip Cara  = Coroa
flip Coroa = Cara
```

Criando funções para Tipos Paramétricos

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
```

Criando funções para Tipos Paramétricos

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`.

Functors de lista

UFABC

- Para as listas nós já temos o functor:

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

Functors de Maybe

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
```

Maybe como resultados que podem falhar

UFABC

- O tipo Maybe não promete entregar nada, apenas tenta entregar um valor do tipo a, mas se algo der errado, ele retorna Nada:

```haskell
safeDiv :: Int -> Int -> Maybe Int
safeDiv x y | y /= 0     = Just (x `div` y)
            | otherwise = Nothing
```

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:

```
> x = 36 `safeDiv` 0
> fmap (*2) x
Nothing
> y = 21 `safeDiv` 3
> fmap (*2) y
Just 14
```

- Nesse caso, se a computação de `x` ou de `y` 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)
```

Functors de Árvores

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) <$> [1,2,3]
2  [2,3,4]
3  > (+1) `fmap` [1,2,3]
4  [2,3,4]
5  -- Para o Functor [] é o mesmo que
6  > (+1) `map` [1,2,3]
7  [2,3,4]
8  > map (+1) [1,2,3]
9  [2,3,4]
```

Exercício 1

- Considere um tipo descrevendo Pokémons que só podem atacar ou defender, o ataque/defesa pode ser descrito por diversos tipos: numérico descrevendo a força, string descrevendo o efeito, tuplas descrevendo ambos, etc.:

```haskell
data Pokemon a = ATK a | DEF a | AD a
               deriving (Show, Eq)
```

- Escreva a instância de Functor para esse tipo.

Para saber mais...

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

Applicative Functors

Functors para funções de múltiplos argumentos

UFABC

- Ok, digamos que eu queira fazer:

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

Família fmap

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
```

Família fmap

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

- Isso é denominado 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
```

UFABC

- 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.)

Applicative Maybe

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 = safeDiv x y
r2 = safeDiv y x

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

Maybe - Tratamento de Exceções

UFABC

```
1 > pure (+) <*> safeDiv 1 0 <*> safeDiv 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

```
1  > pure (++) <*> ["ha","heh","hmm"] <*> ["?","!","."]
2  ["ha?","ha!","ha.","heh?","heh!","heh."
3    ,"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 []
```

Operações com Listas

UFABC

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

```
1 -- Tb não dá esse resultado
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 (dependendo da função sendo avaliada).

Sequenciamento

- 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
```

Sequenciamento

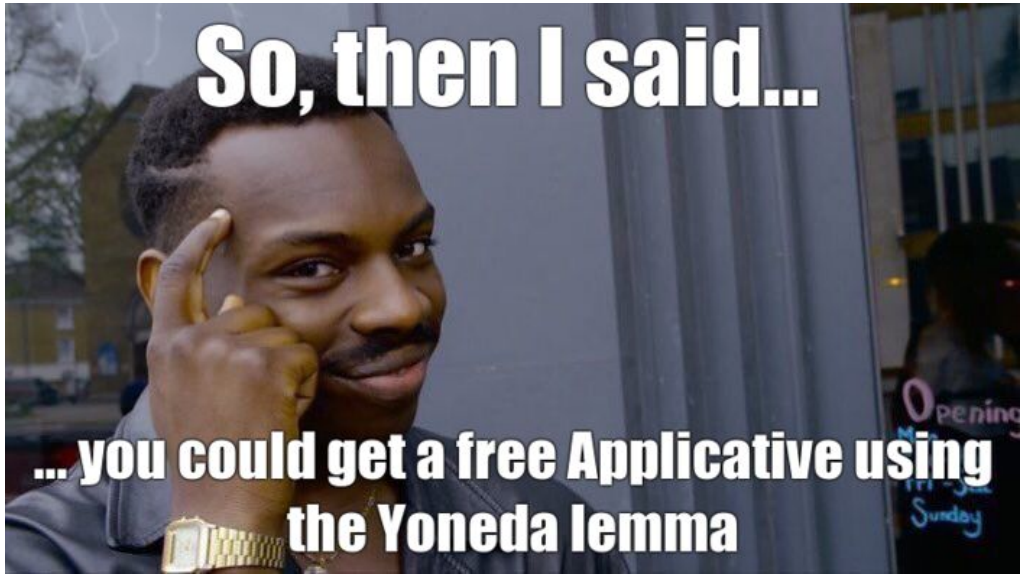
UFABC

```
> 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],[]]
[]
```

UFABC

- 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

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:

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

UFABC

- Podemos resolver isso usando `safeDiv` 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  -> safeDiv n m
```

UFABC

- Agora temos:

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

- Mas nosso código está confuso...

Applicative?

UFABC

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

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

- Mas `safeDiv` 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.

UFABC

- 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`.

- 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 ->
                 safeDiv n m
```

UFABC

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

UFABC

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

UFABC

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

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

```haskell
pares xs ys = [(x,y) | x <- xs, y <- ys]
  == do x <- xs
        y <- ys
        return (x,y)
```

Exercício 2

UFABC

- Escreva a instância de Monads para o tipo Pokémon:

```haskell
data Pokemon a = ATK a | DEF a | AD a
               deriving (Show, Eq)
```

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
```

UFABC

- Também temos a versão monádica de `filter`:

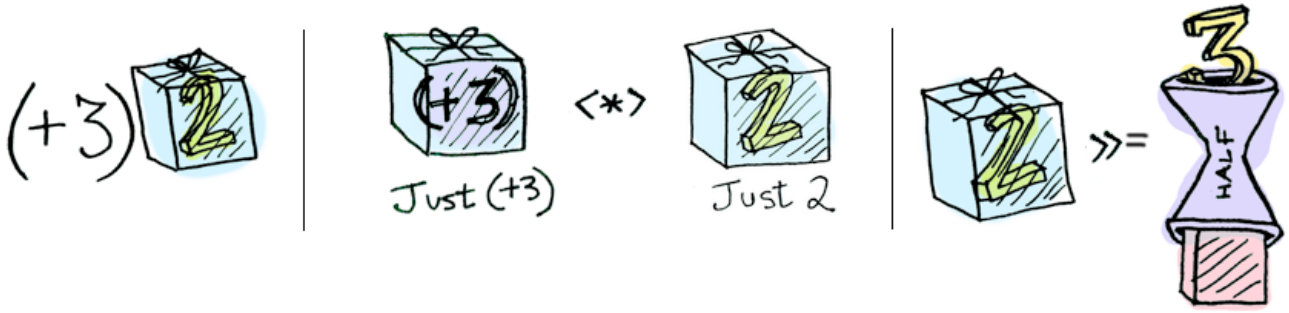
```haskell
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
filterM p []     = return []
filterM p (x:xs) = do b  <- p x
                      ys <- filter M p xs
                      return (if b then x:ys else ys)
```

UFABC

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

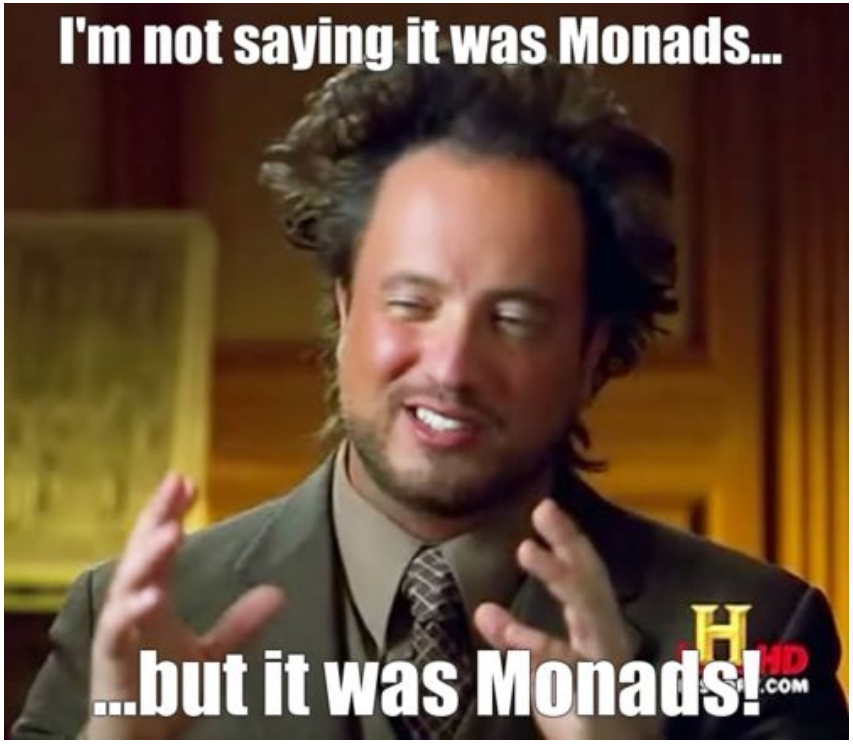
```
> filterM (\x -> [True, False]) [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

- 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.

UFABC

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 ()
```

Entrada e Saída puras

Se eu fizer:

```
1 do putChar 'a'
2    putChar 'a'
```

Na verdade ele estará fazendo algo como:

```
1 (_, env')  = putChar 'a' env
2 (_, 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

UFABC

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.

Exercício 3

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)
```

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'
```

Exercício 4

UFABC

Escreva a função **putStrLn** usando Applicative.

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

Para saber mais

UFABC

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

Tarefa para casa

Para entregar

UFABC

- Lista 3
  - Prazo: 02/11
  - Lista 3: `https://classroom.github.com/a/4po6laAz`
- Não deixe de escolher o seu nome na lista para que possamos relacionar o usuário GitHub à você!

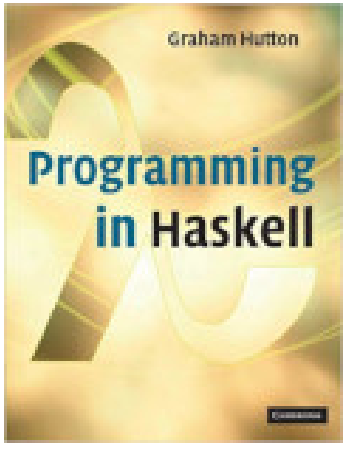
Referências

UFABC

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

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

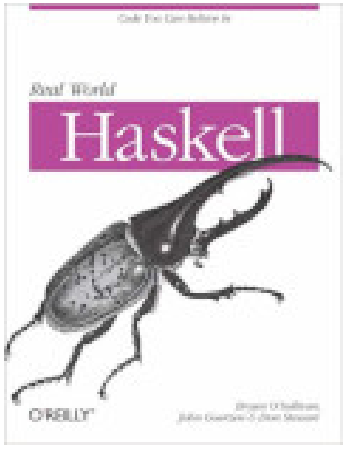
A primeira edição (antiga), que tem boa parte do conteúdo da segunda edição, está disponível na biblioteca:



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

UFABC

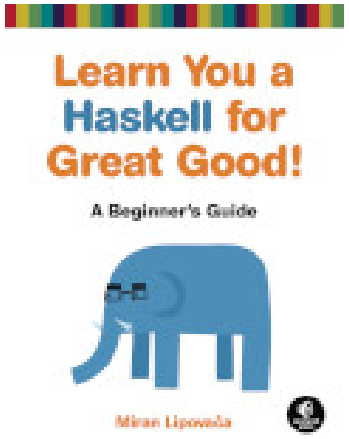
- [SGS]



- 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/`