Programação Funcional — BCC222

Aulas 17,18

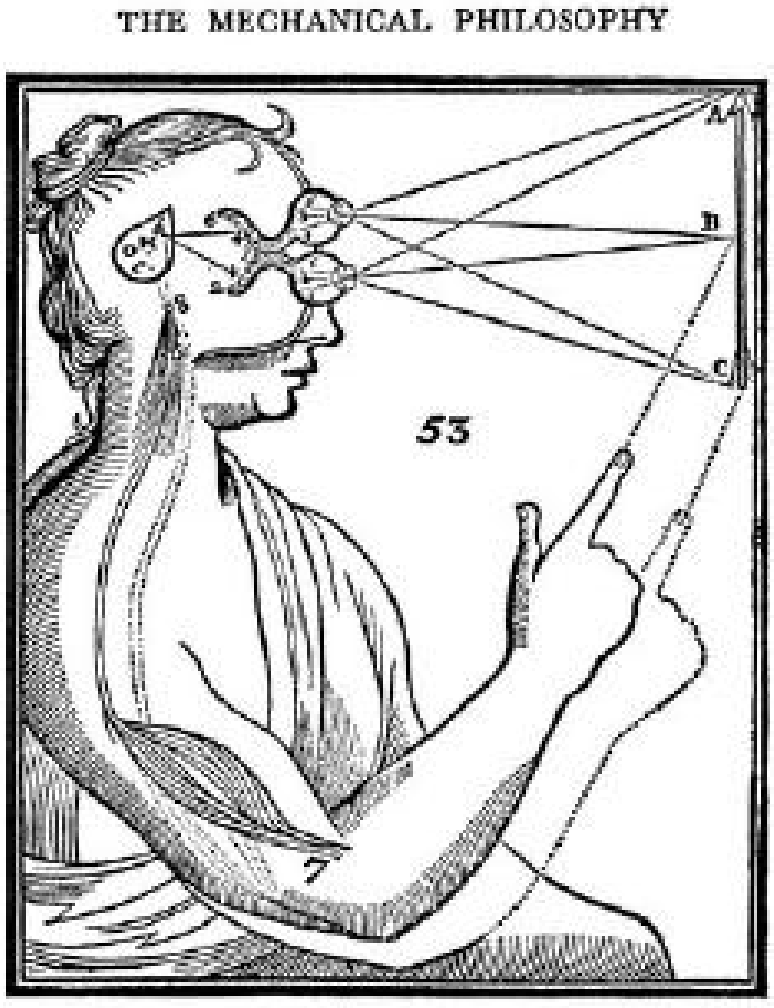
IO e Mônadas

Lucília Camarão de Figueiredo
Departamento de Ciência da Computação
Universidade Federal de Ouro Preto

Parte I

O problema Corpo e Mente

O problema Corpo e Mente



Parte II

Comandos

Imprimir um caractere

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

Por exemplo,

```
putChar '!'
```

denota o comando (ou ação) que, *se for executado*, imprime um ponto de exclamação.

Combinando dois comandos

```
(») ::  IO () -> IO () -> IO ()
```

Por exemplo,

```
putChar '?'  » putChar '!'
```

denota o comando (ou ação) que, *se for executado*, imprime um ponto de interrogação seguido de um ponto de exclamação.

Comando nulo

```
done ::  IO ()
```

O termo `done` apenas especifica um comando que, se for executado, não faz nada. (Compare pensar em não fazer nada com de fato não fazer nada: são coisas distintas.)

Imprimir um string

```
putStr ::  String -> IO ()
putStr []     = done
putStr (x:xs) = putChar x » putStr xs
```

Portanto, `putStr "?!"` é equivalente a

```
putChar '?'  » (putChar '!'  » done
```

e ambos denotam o comando que, *se for executado*, imprime um ponto de interrogação seguido de um ponto de exclamação.

Funções de ordem superior

Podemos definir `putStr` de forma mais compacta do seguinte modo:

```
putStr ::  String -> IO ()
putStr = foldr (») done .  map putChar
```

O operador » é associativo e tem como identidade `done`.

```
m » done     = m
done » m     = m
(m » n) » o = m » (n » o)
```

Main

Você já deve estar desesperado para saber "quando um comando é finalmente executado?"

Aqui está o arquivo `Confused.hs`:

```
module Confused where
main ::  IO ()
main = putStr "!?"
```

Executar esse programa imprime um indicador de perplexidade:

```
lucilia> runghc Confused.hs
?!  lucilia>
```

Portanto, `main` é a ligação entre a mente de Haskell e o corpo de Haskell — o análogo da glândula pineal de Descartes.

Imprimir um string seguido de *newline*

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

Aqui está o arquivo `ConfusedLn.hs`:

```
module ConfusedLn where
main ::  IO ()
main = putStrLn "!?"
```

Isso imprime o resultado de maneira mais apropriada:

```
lucilia> runghc Confused.hs
?!
lucilia>
```

Parte III

Raciocínio equacional

Raciocínio equacional perdido

Esse programa em Standard ML imprime `"haha"` como efeito colateral.

```
output(std_out,"ha"); output(std_out,"ha")
```

Mas o seguinte programa Standar ML apenas imprime `"ha"` como efeito colateral.

```
let val x = output(std_out,"ha") in x; x end
```

E o seguinte programa Standard ML novamente imprime `"haha"` como efeito colateral.

```
let fun f () = output(std_out,"ha") in f (); f () end
```

Raciocínio equacional recuperado

Em Haskell, o termo

```
(1+2) * (1+2)
```

e o termo

```
let x = 1+2 in x * x
```

são equivalentes (e ambos avaliam para 9).

Em Haskell, o termo

```
putString "ha" » putString "ha"
```

e o termo

```
let m = putString "ha" in m » m
```

também são equivalentes (e ambos imprimem `"haha"`).

Parte IV

Comandos com valores

Ler um caractere

Anteriormente, escrevemos `IO ()` para o tipo de comandos que não têm nanhum valor como resultado. Em Haskell, `()` é o tipo trivial que contém apenas um valor *non-bottom*, que é também escrito como `()`.

Escrevemos `IO Char` para o tipo de comandos que têm como resultado um valor do tipo `Char`.

Veja por exemplo o tipo da função para ler um caractere:

```
getChar ::  IO Char
```

A execução do comando `getChar` quando a entrada padrão contém `"abc"` resulta no valor `'a'` e a entrada restante será `"bc"`.

Não fazer nada e retornar um valor

De maneira geral, escrevemos `IO a` para o tipo de comandos que retornam um valor de tipo `a`.

O comando

```
return ::  a -> IO a
```

é similar a `done`, no sentido de que não ele faz nada, mas ele entretanto retorna o valor dado.

A execução do comando

```
return [] ::  IO String
```

quando a entrada contém `"def"` resulta no valor `[]` e a entrada permanece inalterada (ou seja, permanece `"def"`).

Combinando comandos com valores

Combinamos comandos por meio do operador »= (pronuncia-se *bind*).

```
(»=) ::  IO a -> (a -> IO b) -> IO b
```

Por exemplo, a execução do comando

```
getChar »= \x -> putChar (toUpper x)
```

quando a entrada é `"abc"` produz como saída `"A"`, e a entrada restante passa a ser `"bc"`.

O operador *bind* em detalhe

```
(»=) ::  IO a -> (a -> IO b) -> IO b
```

Se

```
m ::  IO a
```

é um comando que retorna um valor de tipo `a`, e

```
k ::  a -> IO b
```

é uma função de um valor de tipo `a` em um comando que produz como resultado um valor de tipo `b`, então

```
m »= k ::  IO b
```

é um comando que, se for executado, se comporta do seguinte modo:

primeiro executa o comando `m` dando como resultado um valor `x` de tipo `a`;
em seguida executa o comando `k x` produzindo como resultado um valor `y` de tipo `b`;
então produz como resultado final o valor `y`.

Lendo uma linha

O programa abaixo lê a os caracteres da entrada até que seja encontrado *newline*, e retorna a lista dos valores lidos.

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

Por exemplo, dada a entrada `"abc\ndef"`, isso retorna o string `"abc"` e a entrada restante passa a ser `"def"`.

Comandos como um caso especial

As operações mais fundamentais sobre comandos são:

```
return ::  a -> IO a
(»=) ::  IO a -> (a -> IO b) -> IO b
```

O comando `done` é um caso especial de `return`, e o operador » é um caso especial de »=.

```
done ::  IO ()
done  = return ()

(») ::  IO () -> IO () -> IO ()
m » n = m »= \() -> n
```

Um análogo de “let”

Embora possa parecer estranho à primeira vista, o combinador `(>>=)` é muito simelhante à familiar expressão “let” de Haskell. Aqui está a regra de tipo para o “let”.

$$\frac{\texttt{E} \vdash \texttt{m :: a} \qquad \texttt{E, x :: a} \vdash \texttt{n :: b}}{\texttt{E} \vdash \texttt{let x = m in n :: b}}$$

Tipicamente,*bind* é combinado com expressões lambda de maneira que lembra expressões “let”. Aqui está a regra de tipo correspondente.

$$\frac{\texttt{E} \vdash \texttt{m :: IO a} \qquad \texttt{E, x :: a} \vdash \texttt{n :: IO b}}{\texttt{E} \vdash \texttt{m >>= \textbackslash x -> n :: IO b}}$$

Ecoando a entrada para a saída

O programa abaixo ecoa sua entrada para a saída, convertendo todo texto de entrada em maiúscula, até que seja encontrada uma linha em branco.

```
echo ::  IO ()
echo = getLine »= \line ->
         if line == "" then
           return ()
         else
           putStrLn (map toUpper line) »
           echo

main ::  IO ()
main = echo
```

Testando o programa

```
lucilia> runghc Echo.hs
One line
ONE LINE
And, another line!
AND, ANOTHER LINE!
lucilia>
```

Parte V

Notação “do”

Lendo uma linha — usando notação “do”

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

é equivalente a

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

Eco — usando notação “do”

```
echo ::  IO ()
echo = getLine »= \line ->
       if line ==  then
         return ()
       else
         putStrLn (map toUpper line) »
         echo
```

é equivalente a

```
echo ::  IO ()
echo = do {
         line <- getLine;
         if line == "" then
           return ()
         else do {
           putStrLn (map toUpper line);
           echo
         }
       }
```

Notação "do" — forma geral

Cada linha `x <- e; ...` é interpretada como `e >>= \x -> ...`

Cada linha `e; ...` é interpretada como `e >> ...`

Por exemplo:

```
do { x1 <- e1;
     x2 <- e2;
     e3;
     x4 <- e4;
     e5;
     e6 }
```

é equivalente a

```
e1 >>= \x1 ->
e2 >>= \x2 ->
e3 >>
e4 >>= \x4 ->
e5 >>
e6
```

Parte VI

Mônadas

Monoides

Um monoide é um par constituído por um operador `()` e um valor `u`, tais que o operador `()` é associativo e tem `u` como identidade.

```
u @@ x           =  x
x @@ u           =  x
(x @@ y) @@ z    =  x @@ (y @@ z)
```

Exemplos de monoides:

```
(+)    e  0
(*)    e  1
(||)   e  False
(&&)   e  True
(++)   e  []
(»)    e  done
```

Mônadas

Sabemos que `(»)` e `done` satisfazem as propriedades de monoide.

```
done » m      = m
m » done      = m
(m » n) » o = m » (n » o)
```

De maneira análoga, `(»=` e `return` também satisfazem essas propriedades.

```
return v »= \x -> m          = m[x:=v]
m »= \x -> return x          = m
(m »= \x -> n) »= \y -> o  = m »= \x -> (n »= \y -> o)
```

leis do “let”

Sabemos que `(»)` e `done` satisfazem as propriedades de monoide.

```
done » m     = m
m » done     = m
(m » n) » o = m » (n » o)
```

De maneira análoga, `(»=` e `return` também satisfazem essas propriedades.

```
return v »= \x -> m          = m[x:=v]
m »= \x -> return x          = m
(m »= \x -> n) »= \y -> o = m »= \x -> (n »= \y -> o)
```

As três leis de mônadas têm análogas na notação “let”.

```
let x = v in m = m[x:=v]
let x = m in x = m
let y = (let x = m in n) in o
                = let x = m in (let y = n in o)
```

“Let” em linguagens com e sem efeito colateral

```
let x = v in m = m[x:=v]
let x = m in x = m
let y = (let x = m in n) in o
                = let x = m in (let y = n in o)
```

Essas leis valem mesmo em uma liguagem tal como SML, na qual a presença de efeito colateral invalida várias formas de raciocínio sobre programas.

Para que a primeira lei seja verdadeira, `v` não pode ser um termo arbitrário, mas deve ser um valor, tal como uma constante. Um valor é um termo que avalia imediatamente para si próprio, não tendo, portanto, efeito colaeral.

Enquanto em SML apenas as três leis acima são válidas para “`let`”, em Haskell vale uma lei ainda mais forte, na qual uma variável pode ser substituída por um termo qualquer, e não apenas por uma constante.

```
let x = m in n = n[x:=m]
```

Parte VII

Usando Mônadas — IO

A classe Monad

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

Minha mônada de IO (1)

```
module MyIO(MyIO, myPutChar, myGetChar, convert) where

type Input = String
type Remainder = String
type Output = String

data MyIO a = MyIO (Input -> (a, Remainder, Output))

apply ::  MyIO a -> Input -> (a, Remainder, Output)
apply (MyIO f) inp = f inp
```

Note que o tipo `MyIO` é abstrato. As únicas operações sobre valores desse tipo são as operações monádicas `myPutChar`, `myGetChar` e `convert`. A operação `apply` não é exportada pelo módulo.

Minha mônada de IO (2)

```
myPutChar ::  Char -> MyIO ()
myPutChar c = MyIO (\inp -> ((), inp, [c]))

myGetChar ::  MyIO Char
myGetChar = MyIO (\(ch:rem) -> (ch, rem, ))
```

Por exemplo,

```
apply myGetChar "abc" == ('a', "bc", "")
apply myGetChar "bc"  == ('b', "c", "")
apply (myPutChar 'A') "def" == ((), "def", "A")
apply (myPutChar 'B') "def" == ((), "def", "B")
```

Minha mônada de IO (3)

```
instance Monad MyIO where
  return x = MyIO (\inp -> (x, inp, ))
  m >>= k  = MyIO (\inp ->
               let (x, rem1, out1) = apply m inp in
               let (y, rem2, out2) = apply (k x) rem1 in
               (y, rem2, out1++out2))
```

Por exemplo

```
apply
  (myGetChar >>= \x -> myGetChar >>= \y -> return [x,y])
  "abc"
== ("ab", "c", "")
apply
  (myPutChar 'A' >> myPutChar 'B')
  "def"
== ((), "def", "AB")
apply
  (myGetChar >>= \x myPutChar (toUpper x))
  "abc"
== ((), "bc", "A")
```

Minha mônada de IO (4)

```
convert ::  MyIO () -> IO ()
convert m = interact (\inp ->
            let (x, rem, out) = apply m inp in
            out)
```

Neste caso

```
interact ::  (String -> String) -> IO ()
```

é parte da biblioteca padrão Prelude.hs. Toda a entrada é convertida em um string (de modo lazy) e passada como argumento para a função, sendo o resultado da função impresso como saída (também de modo lazy).

Usando minha mônada de IO (1)

```
module MyEcho where

import Char
import MyIO

myPutStr ::  String -> MyIO ()
myPutStr = foldr (») (return ()) .  map myPutChar

myPutStrLn ::  String -> MyIO ()
myPutStrLn s = myPutStr s » myPutChar '\n'
```

Usando minha mônada de IO (1)

```
myGetLine ::  MyIO String
myGetLine = myGetChar »= \x ->
            if x == '\n' then
              return []
            else
              myGetLine »= \xs ->
              return (x:xs)

myEcho ::  MyIO ()
myEcho = myGetLine »= \line ->
         if line ==  then
           return ()
         else
           myPutStrLn (map toUpper line) »
           myEcho

main ::  IO ()
main = convert myEcho
```

Testando

```
lucilia> runghc MyEcho
This is a test.
THIS IS A TEST.
It is only a test.
IT IS ONLY A TEST.
Were this a real emergency, you'd be dead now.
WERE THIS A REAL EMERGENCY, YOU'D BE DEAD NOW.
lucilia>
```

Você pode também usar a notação “do”

```
myGetLine ::  MyIO String
myGetLine = do {
              x <- myGetChar;
              if x == '\n' then
                return []
              else do {
                xs <- myGetLine;
                return (x:xs)
              }
            }

myEcho ::  MyIO ()
myEcho = do {
           line <- myGetLine;
           if line == "" then
             return ()
           else do {
             myPutStrLn (map toUpper line);
             myEcho
           }
         }
```

Parte VIII

A Mônada de Listas

A Mônada de Listas

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

-- instance Monad [] where
--   return   ::  a -> [a]
--   return x = [ x ]

--   (>>=)      ::  [a] -> (a -> [b]) -> [b]
--   m >>= k    = [ y | x <- m, y <- k x ]
```

Equivalentemente, podemos definir:

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

ou

```
-- m >>= k = concat (map k m)
```

Noatação "do" para a mônada de listas

```
pairs ::  Int -> [(Int, Int)]
pairs n = [ (i,j) | i <- [1..n], j <- [(i+1)..n] ]
```

é equivalente a

```
pairs' ::  Int -> [(Int, Int)]
pairs' n = do {
             i <- [1..n];
             j <- [(i+1)..n];
             return (i,j)
           }
```

Por exemplo,

```
lucilia> ghci Pairs
GHCi, version 6.10.4:  http://www.haskell.org/ghc/ :?  for h
Pairs> pairs 4
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
Pairs> pairs' 4
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
```

Mônadas com adição

```
-- class Monad m => MonadPlus m where
--   mzero ::  m a
--   mplus ::  m a -> m a -> m a

-- instance MonadPlus [] where
--   mzero ::  [a]
--   mzero = []

--   mplus ::  [a] -> [a] -> [a]
--   mplus = (++)

-- guard ::  MonadPlus => Bool -> m ()
-- guard False = mzero
-- guard True  = return ()
-- msum ::  MonadPlus => [m a] -> m a
-- msum = foldr mplus mzero
```

Usando guardas

```
pairs'' ::  Int -> [(Int, Int)]
pairs'' n = [ (i,j) | i <- [1..n], j <- [1..n], i < j ]
```

é equivalente a

```
pairs''' ::  Int -> [(Int, Int)]
pairs''' n = do {
               i <- [1..n];
               j <- [1..n];
               guard (i < j);
               return (i,j)
             }
```

Por exemplo,

```
lucilia> ghci Pairs GHCi, version 6.10.4:  http://www.haskel
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
Pairs> pairs''' 4
[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
```

Parte IX

A Mônada de Parsers

Módulo ParseMonad

```
module ParseMonad(Parser,apply,parse,char,spot,token,
  star,plus,parseInt) where

import Char
import Monad

-- O tipo de parsers
data Parser a = Parser (String -> [(a, String)])

-- Aplica um parser
apply ::  Parser a -> String -> [(a, String)]
apply (Parser f) s = f s

-- Retorna valor obtido pelo parser, supondo pelo menos sucesso
parse ::  Parser a -> String -> a
parse m s = head [ x | (x,t) <- apply m s, t == "" ]
```

Parser é uma mônada

```
-- Parsers formam uma mônada

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

-- return substitui succ
--   (>>=) substitui (***)

instance Monad Parser where
   return x = Parser (\s -> [(x,s)])
   m >>= k  = Parser (\s ->
                  [ (y, u) |
                    (x, t) <- apply m s,
                    (y, u) <- apply (k x) t ])
```

Parser é uma mônada com adição

```
-- Algumas mônadas possuem estrutura aditiva

-- class MonadPlus m where
--    mzero ::  m a
--    mplus ::  m a -> m a -> m a

-- mzero substitui fail
-- mplus substitui (+++)

instance MonadPlus Parser where
  mzero       = Parser (\s -> [])
  mplus m n   = Parser (\s -> apply m s ++ apply n s)
```

Parsing de um caractere

```
-- Cria um parser a partir de um predicado (e.g.  isDigit)
spot ::  (Char -> Bool) -> Parser Char
spot p = Parser f
  where
  f []     = []
  f (c:s)  | p c = [(c, s)]
           | otherwise = []

-- Cria um parser para um caractere particular
token c = spot (==c)
```

Parsing de um caractere

```
- Parse um único caractere
char ::  Parser Char
char = Parser f
  where
  f []     = []
  f (c:s)  = [(c,s)]

-- Parse caractere que satisfaz um predicado (p.e., isDigit)
spot ::  (Char -> Bool) -> Parser Char
spot p = do { c <- char; guard (p c); return c }

-- Parse um dado caractere
token ::  Char -> Parser Char
token c = spot (== c)
```

Parsing de uma lista

```
-- casa zero ou mais ocorrências
star ::  Parser a -> Parser [a]
star p = plus p `mplus` return []

-- casa uma ou mais ocorrências
plus ::  Parser a -> Parser [a]
plus p = do { x <- p;
              xs <- star p;
              return (x:xs) }
```

Parsing de uma lista

```
-- casa com um número natural
parseNat ::  Parser Int
parseNat = do { s <- plus (spot isDigit);
                return (read s) }

-- casa com um número negativo
parseNeg ::  Parser Int
parseNeg = do { token '-';
                n <- parseNat
                return (-n) }

-- casa com um inteiro
parseInt ::  Parser Int
parseInt = parseNat `mplus` parseNeg
```

Módulo ExprMonad

```
module ExprMonad where

import Monad
import ParseMonad

data Expr = Con Int
          | Expr :+:  Expr
          | Expr :*:  Expr
          deriving (Eq,Show)

eval ::  Expr -> Int
eval (Con i)     = i
eval (e :+:  f) = eval e + eval f
eval (e :*:  f) = eval e * eval f
```

Parsing de uma expressão

```
expr ::  Parser Expr
expr = parseCon `mplus` parseAdd `mplus` parseMul
  where
  parseCon = do { i <- parseInt;
                  return (Con i) }

  parseAdd = do { token '(';
                  d <- expr;
                  token '+';
                  e <- expr;
                  token ')';
                  return (d :+:  e) }

  parseMul = do { token '(';
                  d <- expr;
                  token '*';
                  e <- expr;
                  token ')';
                  return (d :*:  e) }
```

Testando o parser

```
lucilia> ghci ExprMonad.hs
GHCi, version 6.10.4:  http://www.haskell.org/ghc/ :?  for h
[1 of 2] Compiling ParseMonad ( ParseMonad.hs, interpreted
[2 of 2] Compiling ExprMonad ( ExprMonad.hs, interpreted Ok,
ExprMonad> parse expr "(1+(2*3))"
Con 1 :+:  (Con 2 :*:  Con 3)
ExprMonad> eval (parse expr "(1+(2*3))")
7
ExprMonad> parse expr "((1+2)*3)"
(Con 1 :+:  Con 2) :*:  Con 3
ExprMonad> eval (parse expr "((1+2)*3)")
9
ExprMonad>
```

Parte X

Mônada de Estado

Mônada de Estado

```
module StateMonad where

data State s a = State (s -> (a,s))

apply ::  State s a -> s -> (a,s)
apply (State f) s = f s

instance Monad (State s) where
  return x = State (\s -> (x,s))
  m >>= k  = State (\s ->
               let (x,t) = apply m s in
               let (y,u) = apply (k x) t in
               (y,u))
```

Números Aleatórios

```
module RandomState where

import StateMonad
import Random

-- data StdGen
-- next ::  StdGen -> (Int, StdGen)

chooseOne ::  State StdGen Int
chooseOne = State next

chooseMany ::  Int -> State StdGen [Int]
chooseMany 0     = return []
chooseMany (n+1) = do {
                      x <- chooseOne;
                      xs <- chooseMany n;
                      return (x:xs)
                    }
```

Convertendo entre monadas

```
-- newStdGen ::  IO StdGen
io ::  State StdGen a -> IO a
io m = do {
           stdgen <- newStdGen;
           let (x, stdgen') = apply m stdgen in
           return x
         }
```

Juntando tudo

```
main ::  IO ()
main = do {
           xs <- io (chooseMany 5)
           print xs;
           ys <- io (chooseMany 5)
           print ys
         }
```

Um exemplo de execução:

```
lucilia> runghc RandomState.hs
[615674669,1843321250,709512427,880597852,433062387]
[560955837,1086298589,1424808266,959935653,780335811]
lucilia>
```

Parte XI

Sequence

Mônada de Estado

Isso é parte da biblioteca padrão:

```
-- sequence ::  Monad m => [m a] -> m [a]
-- sequence []     = []
-- sequence (m:ms) = do {
--                      x <- m;
--                      xs <- sequence ms;
--                      return (x:xs)
--                    }
```

Mônada Parser — parsing de um dado string

```
match ::  String -> Parser String
match []     = return []
match (x:xs) = do {
                   y <- token x;
                   ys <- match xs;
                   return (y:ys)
                 }
```

é equivalente a

```
match' ::  String -> Parser String
match' xs = sequence (map token xs)
```

Mônada de Estado — lista de números aleatórios

```
chooseMany ::  Int -> State StdGen [Int]
chooseMany 0      = return []
chooseMany (n+1) = do {
                       x <- chooseOne;
                       xs <- chooseMany n;
                       return (x:xs)
                      }
```

é equivalente a

```
chooseMany' ::  Int -> State StdGen [Int]
chooseMany' n = sequence (replicate n chooseOne)
```