関数型プログラミング
第12回　モナド

萩野　達也
hagino@sfc.keio.ac.jp

Slide URL

https://vu5.sfc.keio.ac.jp/slide/

モナドのクラス

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

- Monad クラスのインスタンスがモナド
  - 2つの関数を実装する必要がある.
  - (>>=) は**バインド**（bind）と呼ばれる

- 2つの関数は次の規則を満たしている必要がある.
  - **モナド則**

```
1. (return x) >>= f    = f x
2. m >>= return        = m
3. (m >>= f) >>= g     = m >>= (\x -> f x >>= g)
```

Maybeモナド

```
data Maybe a = Nothing | Just a   deriving (Eq, Ord)

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

- 「Maybe a」は失敗を扱うためによく用いられる.
  - 「Just x」は成功した場合の値を表している.
  - 「Nothing」は失敗を表している.
- f:: a -> Maybe b
  - f は「b」の型の値を返すかもしれない.
  - 「b」の型の値を返すことができない場合には「Nothing」を返す.

例

```
lookup::(Eq a) => a -> [(a, b)] -> Maybe b
```

lookup

```
lookup::(Eq a) => a -> [(a,b)] -> Maybe b
```

- **lookup** は2つの引数を取る:
  - インデックス
  - 連想リスト(タプルのリスト)
- lookup は次の値を返す:
  - 与えられたインデックスのタプルがあった場合には，対応する値を「**Just x** 」として返す.
  - 対応するタプルがなかった場合には，「**Nothing**」を返す.

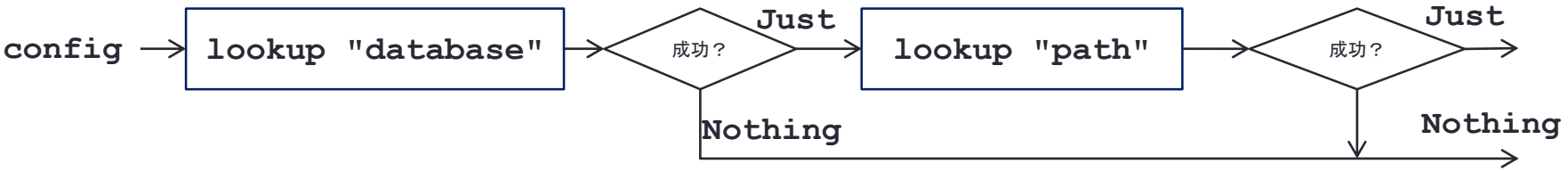
```
lookup "three" [("one", 1), ("two", 2), ("three", 3)] ⇒ Just 3
lookup "four"  [("one", 1), ("two", 2), ("three", 3)] ⇒ Nothing

lookup "path" [("type", "cgi"), ("path", "/var/app")] ⇒ Just "/var/app"
lookup "url"  [("type", "cgi"), ("path", "/var/app")] ⇒ Nothing
```

lookupを組み合わせる

```
config :: [(String, [(String, String)])]
config =
  [ ("database",  [("path", "/var/app/db"), ("encoding", "euc-jp")]),
    ("urlmapper", [("cgiurl", "/app"), ("rewrite", "True")]),
    ("template",  [("path", "/var/app/template")]) ]
```

- lookupの結果にさらにlookupを適用したい.
  - 最初のlookupが成功したかどうかを確認する必要がある.




```
case (lookup "database" config) of
  Just entries -> lookup "encoding" entries
  Nothing      -> Nothing
```

モナド則を使う

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

- Maybeがモナドであることから:

```
case (lookup "database" config) of
  Just entries -> lookup "encoding" entries
  Nothing      -> Nothing
```

```
lookup "database" config >>= lookup "encoding"
```

```
return config >>= lookup "database" >>= lookup "encoding"
```

練習問題12－1

- 次のプログラムは **x** を2で割るが，偶数でない時には失敗する．

```
div2::Int -> Maybe Int
div2 x = if even x then Just (x `div` 2)
                   else Nothing
```

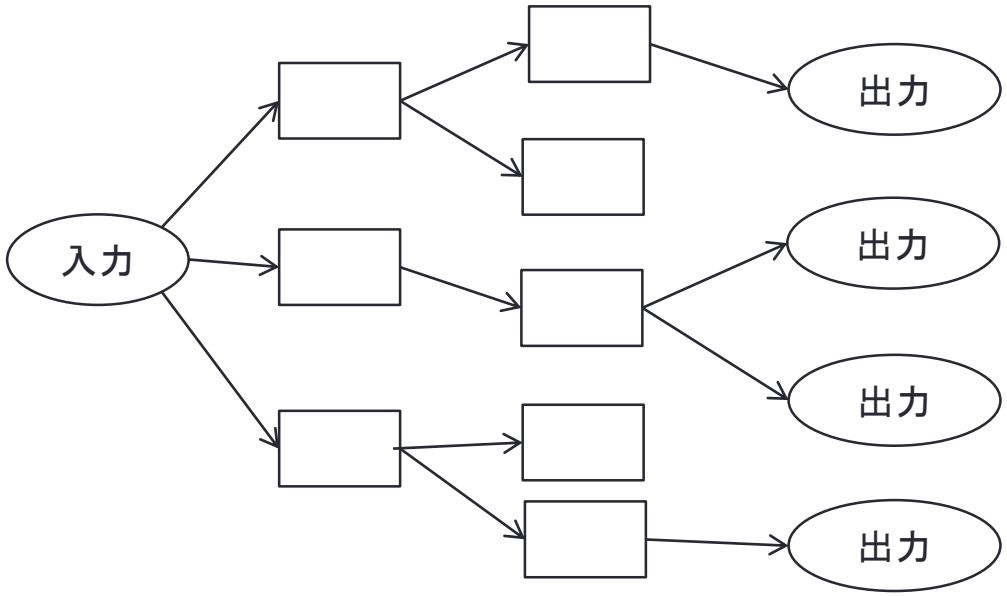
- 例
  - **div2 4 ⇒ Just 2**
  - **div2 3 ⇒ Nothing**

- **div2** を3回使うことによって，与えられた数字を8で割るが，8で割れない場合には失敗する関数 **div8** を定義しなさい．
  - **div8 24 ⇒ Just 3**
  - **div8 20 ⇒ Nothing**

```
div8::Int -> Maybe Int
div8 x = ...
```

Listモナド

- Maybeモナド
  - 失敗などして値が存在しない場合を扱うことができる.
- List Monad
  - 扱う値の数が増えたり減ったりする場合を扱う.



Listモナド

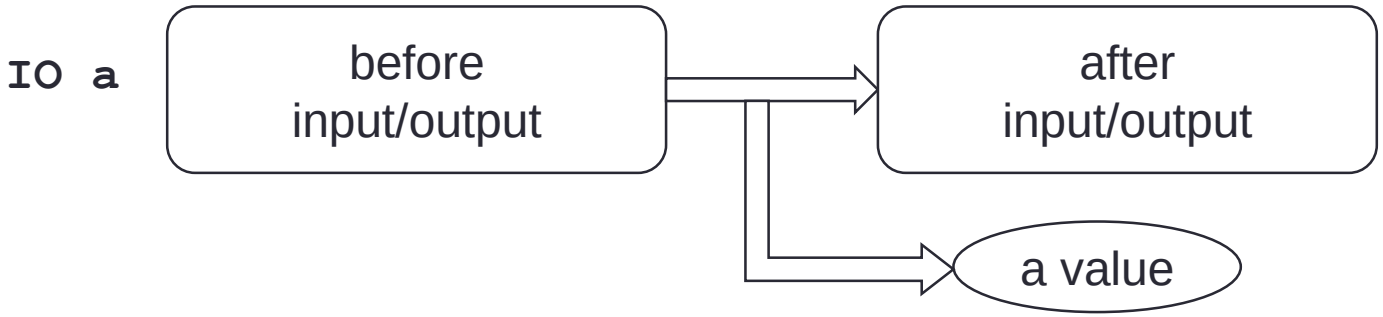
```
instance Monad [] where
  xs >>= f    = concatMap f xs
  return x    = [x]
```

- 例
  - ファイル名の展開
    - `expandCharClass "img[012].png"`
      `⇒ ["img0.png", "img1.png", "img2.png"]`
    - `expandAltWorlds "img.{png,jpg}"`
      `⇒ ["img.png", "img.jpg"]`
  - 2つの展開関数を組み合わせる
    - `expandPattern::String -> [String]`
    - `expandPattern pattern`
      `= expandCharClass pattern >>= expandAltWords`
    - `expandPattern "img[012].{png,jpg}"`
      `⇒ ["img0.png", "img0.jpg", "img1.png",`
      `"img1.jpg", "img2.png", "img2.jpg"]`

IOモナド

- 入出力には順番がある.
  - e.g. プロンプトは入力の前に出力する.
  - e.g. "Sunday"を"Monday"の前に出力する.

- 「`IO a`」の値は入出力アクションを表している.



- 「`(>>=)`」および「`return`」はシステムで実装されている.
  - `x >>= y`
  - アクション「`x`」がうまくいった場合には，その結果をアクション「`y`」の渡す.
  - そのため，アクション「`x`」はアクション「`y`」の前に行う必要がある.

IOモナドの例

cat.hs

```
main = do cs <- getContents
          putStr cs
```

- **getContents::IO String**
  - コンソールから入力するアクション
- **putStr::String -> IO ()**
  - 文字列をコンソールに出力するアクション

- do 式は (>>=) で書くことができる.

```
main = getContents >>= putStr
```

IOモナドと (>>)

- 次のdo式は (>>=) を使って書き直すことができる.

```
do putStrLn "Hello, World!"
   putStrLn "Hello, agran!!!"
```

```
putStrLn "Hello, World!" >>= (\x -> putStrLn "Hello, agran!!!")
```

- 2つ目の **putStrLn** は変数 **x** を使わないので, **Monad** のクラスメソッド **(>>)** を使うことができる.

```
putStrLn "Hello, World!" >> putStrLn "Hello, agran!!!"
```

```
class Monad m where
  (>>)::m a -> m b -> m b
  f >> g = f >>= (\x -> g)
```

IOモナドのその他の利用

- IOモナドは入出力以外にも，副作用などがあり実行順序が重要な場合にも使われる.
  - 例：乱数の生成

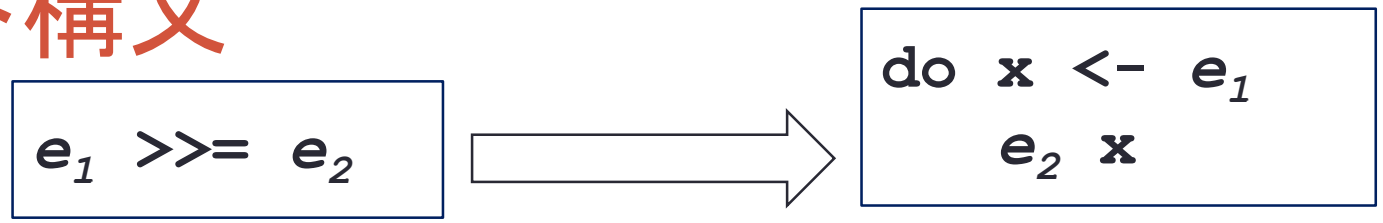
```
import System.Random

main = do g <- getStdGen
          let ns::[Int]
              ns = map (`mod` 100) $ take 100 $ randoms g
          print ns
```

- その他の利用：
  - 現在の時刻を取得する.
  - OSの機能を呼び出す.

モナド構文

- do 式



- let 節

```
do let n = 2 * 6
   in print n
```

do式のオフサイドルールの関係で正しく構文解析されない

```
do { let n = 2 * 6; in print n }
```

```
do let n = 2 * 6
   print n
```

or

```
do let n = 2 * 6
      in print n
```

例(1)

```
nameDo::IO ()
nameDo = do putStr "What is your first name? "
            first <- getLine
            putStr "And your last name? "
            last <- getLine
            let full = first ++ " " ++ last
            putStrLn ("Please to meet you, " ++ full ++ "!")
```

```
nameNoDo::IO ()
nameNoDo = putStr "What is your first name? " >>
           getLine >>= \first ->
           putStr "And your last name? " >>
           getLine >>= \last ->
           let full = first ++ " " ++ last
           in putStrLn ("Please to meet you, " ++ full ++ "!")
```

例(2)

- **lookup** を二重に行う場合を，do式で書いてみる．

```
case (lookup "database" config) of
  Just entries -> lookup "encoding" entries
  Nothing      -> Nothing
```

```
lookup "database" config >>= lookup "encoding"
```

```
do entries <- lookup "database" config
   lookup "encoding" entries
```

練習問題12－2

- 練習問題11－1の div8 をdo式を用いて書きなさい.

div8.hs

```
import System.Environment

div2::Int -> Maybe Int
div2 x = if even x then Just (x `div` 2)
                   else Nothing

div8::Int -> Maybe Int
div8 x = do y <- div2 x
            ...

main = do args <- getArgs
          print $ div8 $ read $ head args
```

練習問題12－3

- 前回の電卓では，0での割り算を行う式を入力するとおかしなことになります．

```
eval::ParseTree -> Rat
eval(Divide p1 p2) = ....
```

- eval を Rat を返す関数としてではなく Maybe Rat を返す関数として定義し，0での割り算があった時にはエラーメッセージを出力して，次の入力を受け付けるようにしなさい．
  - `eval::ParseTree -> Maybe Rat`

calcm.hs

```
import Data.Char
data Token = Num Int | Add | Sub | Mul | Div
data ParseTree = ...
type Parser = [Token] -> (ParseTree, [Token])
...

eval::ParseTree -> Maybe Rat
eval(Number x) = Just x
...
eval(Divide p1 p2) = ...

showResult::Maybe Rat -> String
showResult Nothing = "error: division by 0"
showResult Just x = show xpus

main = do cs <- getContents
          putStr $ unlines $
            map (showRresult . eval . fst . parseExpr . tokens) $ lines cs
```

実行例

```
% ./calcm
5+4*3/2
11
5+2/0-3
error: division by 0
1/2-1/6
1/3
```