関数型プログラミング
第10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)
```

- 「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
```

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

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.

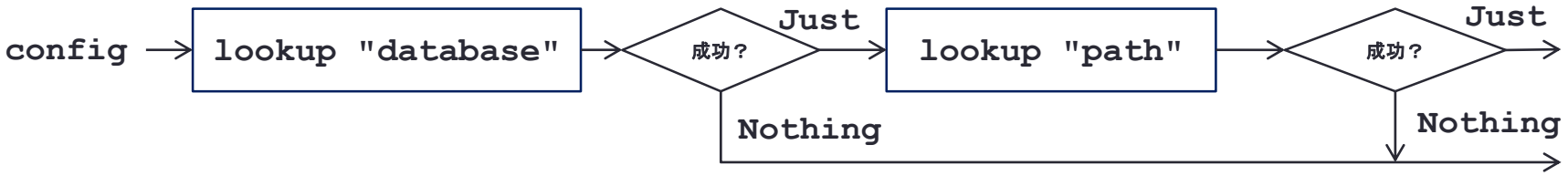
```
lookup "three" [("one", 1), ("two", 2), ("three", 3)] ⇒ Just 3
lookup "four"  [("one", 1), ("two", 2), ("three", 3)] ⇒ Nothing

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10－1

- 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．

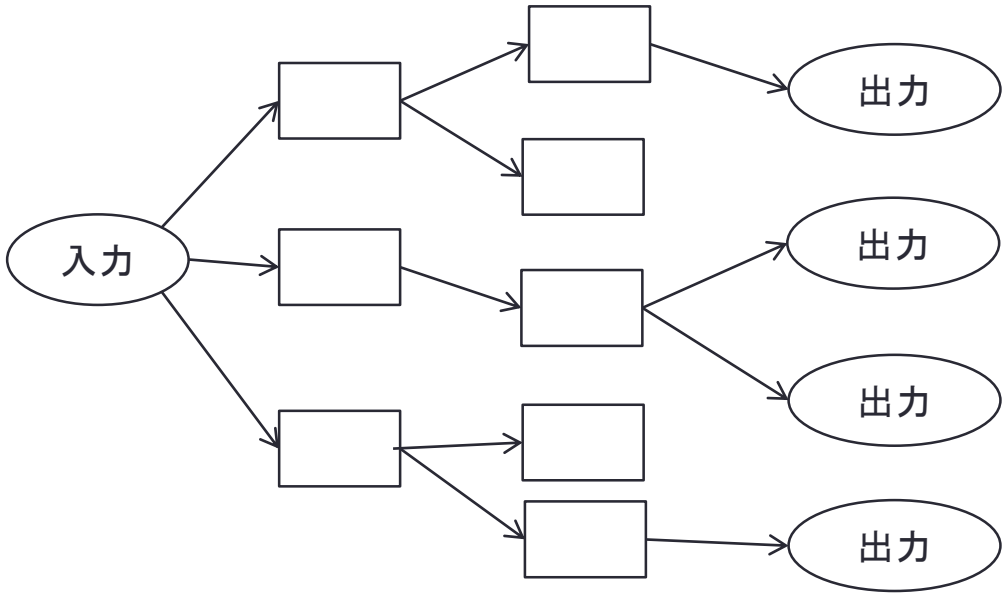
```
div2::Int -> Maybe Int
div2 x = if even x then Just (x `div` 2)
                   else Nothing
```

- 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モナド
  - 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    = concat $ map 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"]`

練習問題10－2

- **replicate** は与えられた要素を与えられた回数コピーしたリストを返す関数です.
  - `replicate :: Int -> a -> [a]`
  - `replicate 3 1 ⇒ [1,1,1]`
  - `replicate 5 'a' ⇒ "aaaaa"`
- **replicate** を使って，与えられたリストのそれぞれの要素を2重にして返す関数 **double** をListモナドを使って定義しなさい.
  - `double [1,2,3] ⇒ [1,1,2,2,3,3]`
  - `double "SFC" ⇒ "SSFFCC"`

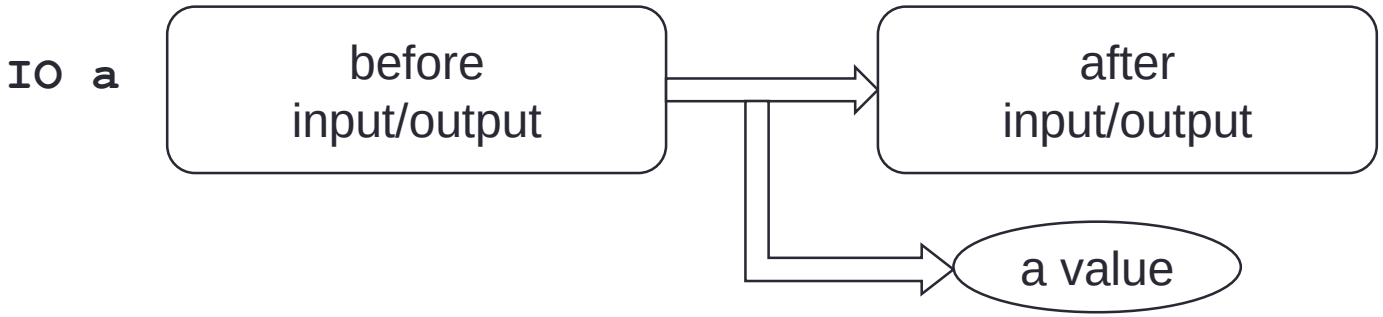
double.hs

```
double xs = ...
```

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 = getContents >>= putStr
```

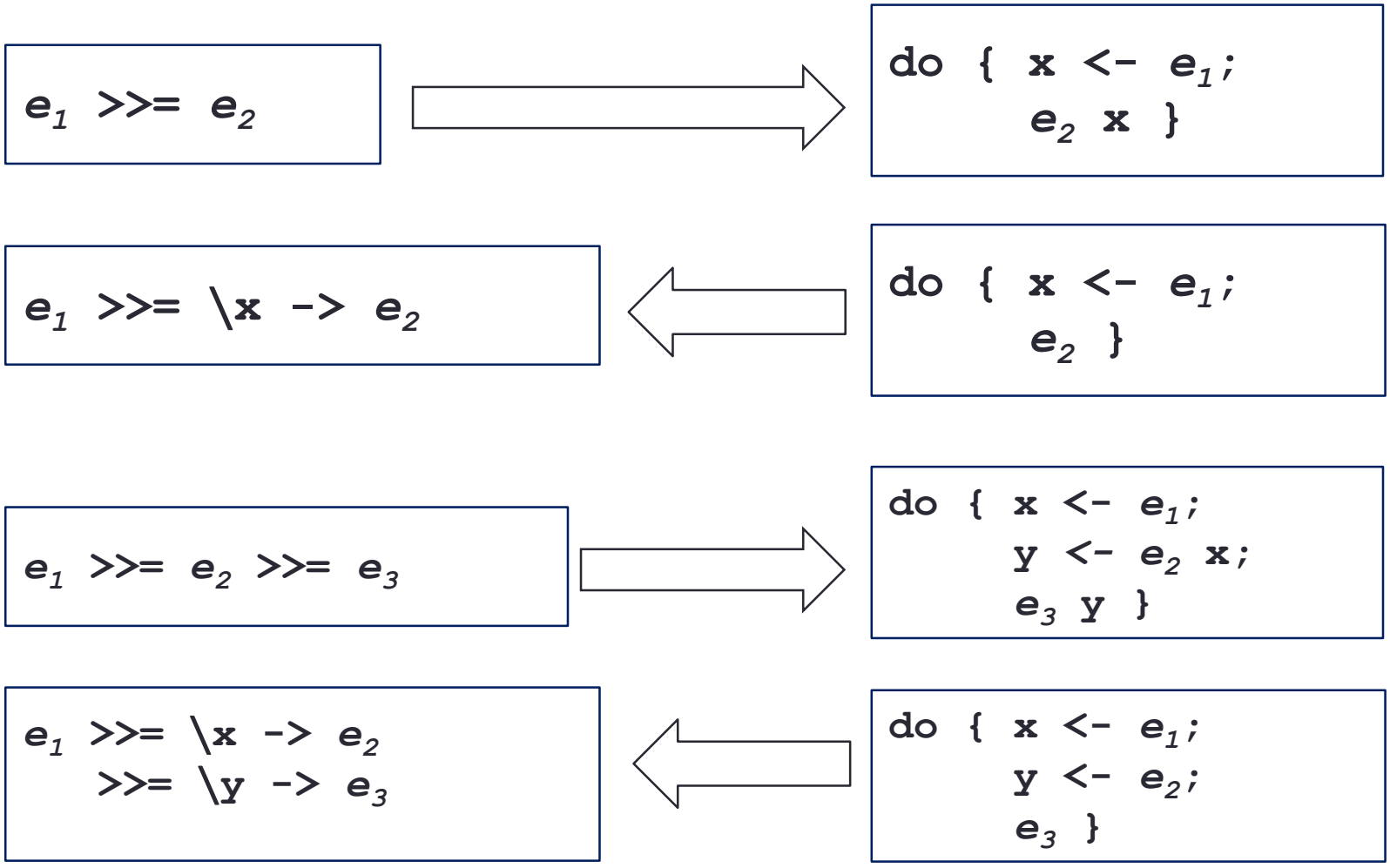
- `getContents::IO String`
  - コンソールから入力するアクション
- `putStr::String -> IO ()`
  - 文字列をコンソールに出力するアクション

- do 式を使うと次のように書くこともできる.

```
main = do { cs <- getContents ;
            putStr cs }
```

モナド構文

- do 式


e1 >>= e2
do { x <- e1;
e2 x }
e1 >>= \x -> e2
do { x <- e1;
e2 }
e1 >>= e2 >>= e3
do { x <- e1;
y <- e2 x;
e3 y }
e1 >>= \x -> e2
>>= \y -> e3
do { x <- e1;
y <- e2;
e3 }

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, again!!!" }
```

```
putStrLn "Hello, World!" >>= \x -> putStrLn "Hello, again!!!"
```

- 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, again!!!"
```

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

例(1)

```
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 ++ "!")
```

```
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 ++ "!") }
```

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 }
```

例(3)

- リスト内包表記は do 構文で書くことができる.

```
[ x * 2 | x <- [1..5] ]
```

```
[1..5] >>= \x -> return(x * 2)
```

```
do { x <- [1..5];
     return (x * 2) }
```

```
[ (x,y) | x <- [1,2],
          y <- [3,4] ]
```

```
[1,2] >>= \x ->
[3,4] >>= \y -> return (x,y)
```

```
do { x <- [1,2];
     y <- [3,4];
     return (x,y)}
```

練習問題10－3

- 練習問題10－1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 }
```

IOアクションと標準入力と出力

- 標準入力(端末)から入力

	型	説明
`getChar`	`IO Char`	1文字読み込む
`getLine`	`IO String`	1行読み込む
`getContents`	`IO String`	EOFまで読み込む

- 標準出力(端末)に出力

	型	説明
`putChar`	`Char -> IO ()`	1文字出力する
`putStr`	`String -> IO ()`	文字列を出力する
`putStrLn`	`String -> IO ()`	文字列と改行を出力する
`print`	`Show a => a -> IO ()`	文字列に変換して出力する

IOアクションとファイル入出力

- 単純なファイル入出力

関数	型	説明
`readFile`	`FilePath -> IO Sring`	ファイルから読み込む
`writeFile`	`FilePath -> String -> IO ()`	ファイルを出力する
`appendFile`	`FilePath -> String -> IO ()`	ファイルを追加書きする

- `FilePath` は `String` の別名

cat3.hs

```
import System.Environment

main = do { args <- getArgs;
            cs <- readFile $ head args;
            putStr cs}
```

```
% stack runghc cat3.hs a.txt
...
...
```

高度なファイル入出力

- 高度なファイル入出力

関数	型	説明
`openFile`	`FilePath -> IOMode -> IO Handle`	ファイルをオープンする
`hClose`	`Handle -> IO ()`	ファイルをクローズする
`hGetChar`	`Handle -> IO Char`	1文字読み込む
`hGetLine`	`Handle -> IO String`	1行読み込む
`hGetContents`	`Handle -> IO String`	EOFまで読み込む
`hPutChar`	`Handle -> Char -> IO ()`	1文字書き込む
`hPutStr`	`Handle -> String -> IO ()`	文字列を書き込む
`hPutStrLn`	`Handle -> String -> IO ()`	文字列と改行を書き込む
`hPrint`	`Show a => Handle -> a -> IO ()`	文字列に変換して書き込む

- **`IOMode`** は **`ReadMode`**, **`WriteMode`**, **`AppendMode`** または **`ReadWriteMode`**.
- import **`System.IO`** を利用前に行う

練習問題10－4

- **cat3.hs** を高度なファイル入出力を使って書き換えなさい.

cat3.hs

```
import System.Environment

main = do { args <- getArgs;
            cs <- readFile $ head args;
            putStr cs}
```

cat4.hs

```
import System.Environment
import System.IO

main = do { args <- getArgs;
            h <- openFile (head args) ReadMode;
            ...
            ...
          }
```

練習問題10－5

- 通常のUNIXの **cat** コマンドと同じように，複数のファイルを引数として指定し，それらのファイルを結合したファイルを標準出力に出力するプログラム **cat5.hs** を書きなさい．

cat5.hs

```
import System.Environment

main = do { args <- getArgs;
            outputFiles args }

outputFiles::[String] -> IO ()
outputFiles [] = ...
outputFiles (x:xs) = ...
```

```
% stack runghc cat5.hs a.txt b.txt c.txt
...
... (output a.txt b.txt c.txt in this order)
```