FUNCTIONAL PROGRAMMING
NO.11 MONAD

Tatsuya Hagino
hagino@sfc.keio.ac.jp

Slide URL

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

Monad class

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

- Each monad is an instance of Monad class:
  - has to have two functions.
  - (>>=) is called bind.
- Two functions have to satisfy the following Monad law:

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

Maybe Monad

```
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`** is often used to handle failure case:
  - **`Just x`** is a value representing success.
  - **`Nothing`** is the value representing failure.
- **`f:: a -> Maybe b`**
  - **`f`** may return a value of **b**
  - It may return Nothing when it cannot return a value in **b**.

example

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

lookup

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

- **`lookup`** takes two arguments:
  - an index value
  - an association list (a list of tuples)

- lookup returns:
  - **`Just x`** if it finds a tuple of the given index.
  - **`Nothing`** when there is no matching tuple.

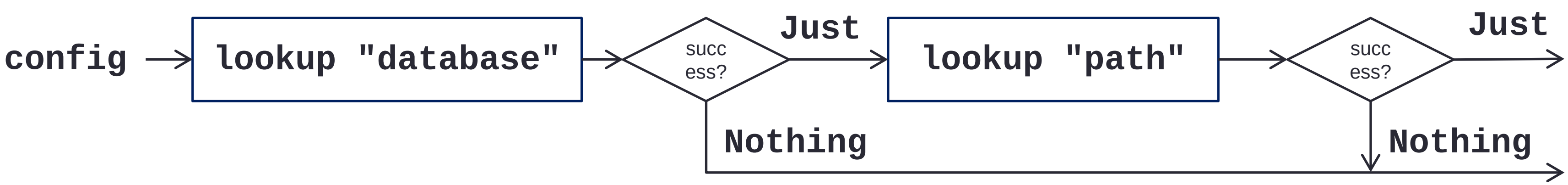
```
lookup "three" [("one", 1), ("two", 2), ("three", 3)] ⇒ Just 3
lookup "four"  [("one", 1), ("two", 2), ("three", 3)] ⇒ Nothing

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

Combining lookup

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

- Applying another lookup to the result of lookup.
  - Need to check whether the first lookup succeed or not.




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

Using Monad Law

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

- from Maybe monad declaration

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

⇩

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

⇩

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

Exercise 11-1

- The following program fails if **x** is not an even number:

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

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

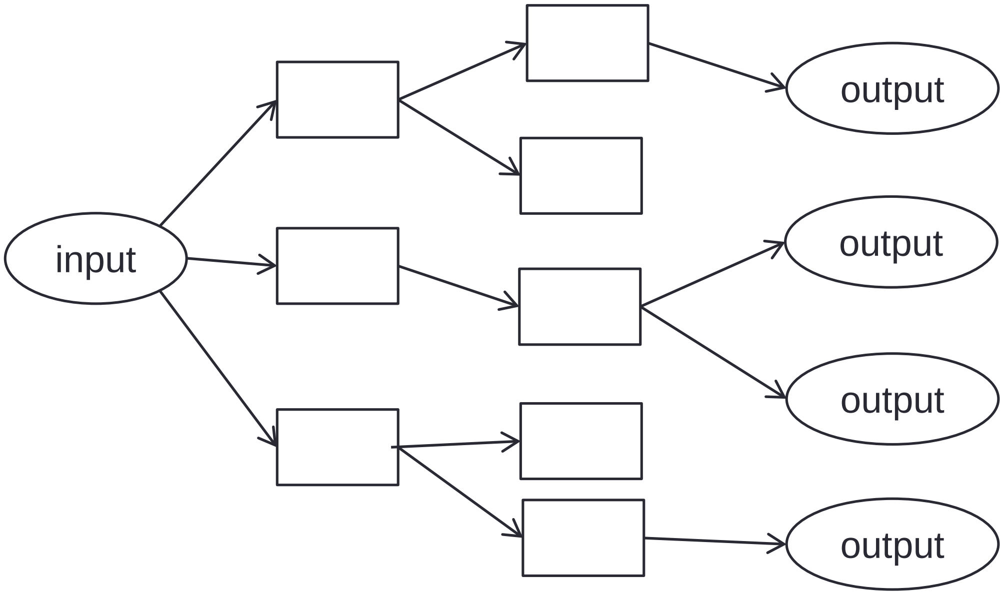
- Write **div8** which tries to divide the given number by 8 but it fails if it cannot. Write it using **div2** three times.
  - **div8 24** ⇒ **Just 3**
  - **div8 20** ⇒ **Nothing**

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

List Monad

- Maybe monad
  - To handle the case that the value may not exist.

- List Monad
  - To handle increase or decrease of the number of values.



List Monad

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

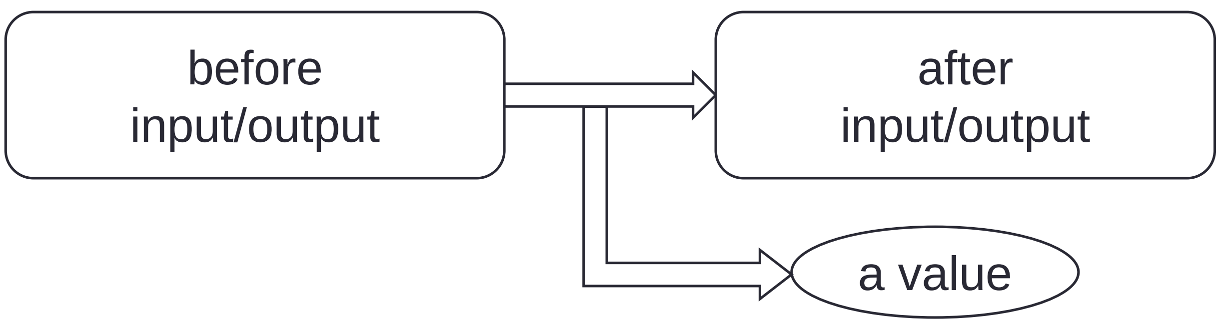
- Example
  - File name pattern expansion
    - `expandCharClass "img[012].png"`
      `⇒ ["img0.png", "img1.png", "img2.png"]`
    - `expandAltWorlds "img.{png,jpg}"`
      `⇒ ["img.png", "img.jpg"]`
  - Combine the two expansion functions:
    - `expandPattern::String -> [String]`
    - `expandPattern pattern`
      `= expandCharClass pattern >>= expandAltWords`
    - `expandPattern "img[012].{png,jpg}"`
      `⇒ ["img0.png", "img0.jpg", "img1.png", "img1.jpg", "img2.png", "img2.jpg"]`

IO Monad

- For input and output, executions need to be ordered.
  - e.g. Output a prompt before input.
  - e.g. Output "Sunday" before "Monday".

- A value of **`IO a`** represents an input/output action

`IO a`



- **`(>>=)`** and **`return`** are implemented by the system.
  - **`x >>= y`**
  - Pass the result of executing action **x** to action **y**
  - Therefore, action **x** needs to be executed before action **y**

IO Monad Example

cat.hs

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

- **`getContents::IO String`**
  - Action to input from console.
- **`putStr::String -> IO ()`**
  - Action to ouput s string to console.

- **do** expression is converted into **`(>>=)`**

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

IO Monad and (>>)

- The following **do** expression is also converted to **(>>=)**

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

⇩

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

- Since the second **putStrLn** does not use variable **x**, **Monad** has a class method **(>>)** for these cases.

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

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

Other Usage of IO Monad

- IO Monad is not only used for input/output, but also used when side effects are important.
  - e.g. Random number generation

```
import System.Random

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

- Other usages
  - Getting the current time
  - Calling Operating System functions

Monad Syntax

- **do** expression

e_1 `>>=` e_2

`do x <-` e_1
`   ` e_2 `x`

e_1 `>>= (\x ->` e_2`)`

`do x <-` e_1
`   ` e_2

- **let** clause

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

does not work, because of offside rule of do expression

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

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

or

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

Example (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 ++ "!")
```

Example (2)

- Write double **lookup** using do expression

```
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
```

Exercise 11-2

- Rewrite div8 of Exercise 11-1 using do syntax.

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
```

Exercise 11-3

- Last week calculator has a problem when it tries to divide by 0.

```
eval::ParseTree -> Int
eval(Divide p1 p2) = eval p1 `div` eval p2
```

- Change **`eval`** not to return **`Int`** but to return **`MayError Int`**.
  - **`eval::ParseTree -> MayError Int`**
- Output error message (division by 0) when it tries to divide by 0.

calcm.hs

```
import Data.Char
import Control.Applicative
data Token = Num Int | Add | Sub | Mul | Div | LPar | RPar
data ParseTree = ...
type Parser = [Token] -> (ParseTree, [Token])
...
data MayError a = Value a | Error String
instance (Show a) => Show(MayError a) where ...
...
instance Monad MayError where ...

eval::ParseTree -> MayError Int
eval(Number x) = Just x
...
eval(Divide p1 p2) = ...

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

Example

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

MayError: Monad for Error Handling

- Create a type for handling errors:

```
data MayError a = Value a | Error String
```

  - It maybe an actual value or an error string.
  - Similar to **Maybe** Monad.
- Firstly, in order to show the value of **MayError**, make it an instance of **Show**.

```
instance (Show a) => Show (MayError a) where
  show (Value x) = show x
  show (Error s) = "error: " ++ s
```

  - If it is a value, show the value.
  - If it is an error, show the error.

```
> show (Value 123)
123
> show (Error "division by 0")
error: division by 0
```

Functor MayError

- Before making it a monad, it need to be an instance of **`Functor`**.
- **`Functor f`** needs **`fmap`** class method:
  - **`fmap::(a -> b) -> f a -> f b`**

```
import Control.Applicative

instance Functor MayError where
  fmap f (Value x) = Value (f x)
  fmap f (Error s) = Error s
```

  - **`fmap`** of **`MayError`** applies the function to the value.
  - If it is an error, keep the error.

```
> fmap (+ 1) (Value 2)
3
> fmap (+ 1) (Error "division by 0")
error: division by 0
```

Applicative MayError

- Next, make it an instance of **`Applicative`**.
- **`Applicative f`** needs two class methods:
  - **`pure::a -> f a`**
  - **`(<*>)::f (a -> b) -> f a -> f b`**

```
instance Applicative MayError where
  pure x = Value x
  (Value f) <*> (Value x) = Value (f x)
  (Value f) <*> (Error s) = Error s
  (Error s) <*> _         = Error s
```

- **`pure`** creates a value in **`MayError`**.
- **`<*>`** is a function application in the level of **`MayError`**.

- **`pure`** and **`<*>`** of **`Applicative`** have to satisfy the following rules:
  - **`pure id <*> v = v`**
  - **`pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`**
  - **`pure f <*> pure x = pure (f x)`**
  - **`u <*> pure y = pure ($ y) <*> u`**

Monad MayError

- Now make the parser an instance of **`Monad`**.
- **`Monad m`** needs two class methods:
  - **`return::a -> m a`**
  - **`(>>=)::m a -> (a -> m b) -> m b`**

```
instance Monad MayError where
  (Value x) >>= f = f x
  (Error s) >>= f = Error s
  return x = Value x
```

- **`return`** is the same as **`pure`** of **`Applicative`**.
- **`(Value x) >>= f`** will pass the value to **`f`**.
- **`(Error s) >>= f`** will ignore **`f`**.
- Using **`Monad do`** expression, codes may become more readable.
  - **`p >>= (\x -> q)`**
  - **`do { x <- p; q }`**

eval

```
eval::ParseTree -> Int
eval(Number x) = x
eval(Plus p1 p2) = eval p1 + eval p2
...
eval(Divide p1 p2) = eval p1 `div` eval p2
```

- Change **`eval`** to return **`MayError Int`** instead of just **`Int`**.

```
eval::ParseTree -> MayError Int

eval(Number x) = Value x

eval(Plus p1 p2) = do x <- eval p1
                      y <- eval p2
                      return (x + y)
...

eval(Divide p1 p2) = do x <- eval p1
                        y <- eval p2
                        if y == 0 then Error "division by 0"
                                  else return (x `div` y)
```

Exercise 11-4

- Handle token error and parse error in the calculator.

```
tokens::String -> MayError [Token]
...
tokens (c:cs) | isDigit c = let (ds, rs) = span isDigit (c:cs)
                            in do ts <- tokens cs
                                  return (Num(read ds):ts)
              | isSpace c = tokens cs
              | otherwise = Error "unknown token"

type Parser = [Token] -> MayError(ParseTree, [Token])
parseFactor::Parser
parseFactor(Num x:l) = return (Number x, l)
...
parseFactor _ = Error "parse error"
parseTerm::Praser
...
parseExpr::Parser
...
parse::[Token] -> MayError ParseTree
parse ts = do (pt, rs) <- parseExpr ts
              if null rs then return pt else Error "extra token"

main = do cs <- getContents
          putStr $ unlines $ map show $
            map (\s -> tokens s >= parse >= eval) $ lines cs
```

Exercise 11-5

- Extend the calculator as you like.
- For example,
  - add other operators (e.g. **`2^3`** is 2 to the power of 3 which is 8)
  - calculate fractions (use **`Rat`** we defined)
  - calculate floating point numbers (use **`Double`**)
  - add trigonometric functions (e.g. **`sin`**, **`cos`**, **`tan`**)
  - introduce variables (i.e. memories) to store the result and refer it later on, like
    - **`x=(2+3)*2`** store 12 to **`x`**
    - **`y=x*x-4`** store 140 to **`y`**
    - **`x+y*3`**
  - and so on