

Монада

монада – моноид в категории эндифункторов (ц) википедия

Пусть у нас есть какой-нибудь эндифунктор на категории Hask (т.е. тип `f`, являющийся экземпляром класса `Functor`). Дополним его структурой моноида.

Обобщённый моноид M в моноидальной категории C — это морфизм $(++)$ в этой категории из $M \otimes M$ в M , а также морфизм `empty` из I в M .

```
class MonoidCat (~>) ( ⊗ ) I => Monoid (~>) ( ⊗ ) I M where  
  (++) :: M M -> M   (join :: Monad m => m (m a) -> m a)  
  empty :: I -> M
```

Монады. Зачем?



return

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

```
> :t return
return :: Monad m => a -> m a
> return "Hello" == Just "Hello"
?
> return "Hello" == ["Hello"]
?
```

return

Согласованность

$(p \gg= \text{return}) = p$

$\text{do } \{x \leftarrow p; \text{return } x\} = \text{do } \{p\}$

$(\text{return } e \gg= f) = f \ e$

$\text{do } \{x \leftarrow \text{return } e; f \ x\} = \text{do } \{f \ e\}$

Ассоциативность ($\gg=$) :

$((p \gg= f) \gg= g) = p \gg= (\lambda x \rightarrow (f \ x \gg= g))$

Этот закон позволяет воспринимать последовательность $a ; b ; c ; \dots$ как монолитную и не заботиться о расстановке скобок в ней.

Композиция Клейсли

$(\gg=) :: \text{Monad } m \Rightarrow (a \rightarrow m \ b) \rightarrow (b \rightarrow m \ c) \rightarrow (a \rightarrow m \ c)$

$(f \gg= g) \ x = f \ x \gg= g$

Монады. Зачем?

```
> let x = [1,2,3]  
> (x >>= return) == x  
?
```

Монады. Зачем?

```
> let x = [1,2,3]
> (x >>= return) == x
True
```

правая единица $(x \gg= \text{return}) = x$

Можно переписать этот закон как
 $(x \gg= (\lambda a \rightarrow \text{return } a)) = x$,

то есть, связывание монадного вычисления x с вычислением, зависящим от параметра и просто-напросто возвращающим этот параметр, есть тождественная функция

Стандартные монады

Монада Identity (тождественная монада), не меняет ни тип значений, ни стратегию связывания вычислений.

```
data Identity a = Identity a
return a = Identity a
(Identity a) >>= f = f a
```

Стандартные монады

Maybe (монада вычислений с обработкой отсутствующих значений)

```
data Maybe a = Nothing | Just a
```

Реализация тривиальна:

```
return a = Just a
```

```
Nothing >>= f = Nothing
```

```
(Just a) >>= f = f a
```


Стандартные монады

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

```
case lookup 'f' [('f','u'),('n','n'),('y','?')] of
  Nothing -> Nothing
  Just y -> case lookup y [('o','l'),('u','c'),('k','y')] of
    Nothing -> Nothing
    Just z -> lookup z [('d','u'),('c','k')]
```

```
lookup 'f' [('f','u'),('n','n'),('y','?')] >=>
  (\a -> lookup a [('o','l'),('u','c'),('k','y')]) >=>
    (\b -> lookup b [('d','u'),('c','k')])
```

`lookupM list a = lookup a list -- lookupM list = \a -> lookup a list`

`Just 'f' >=> lookupM [('f','u'),('n','n'),('y','?')] >=> lookupM [('o','l'),('u','c'),('k','y')] >=> lookupM [('d','u'),('c','k')]`

`do`

```
y <- lookup 'f' [('f','u'),('n','n'),('y','?')]
z <- lookup y [('o','l'),('u','c'),('k','y')]
return (lookup z [('d','u'),('c','k')])
```

Стандартные монады (Maybe)

```
maybeHalf :: Int -> Maybe Int
```

```
maybeHalf a
```

```
    | even a = Just (div a 2)
```

```
    | otherwise = Nothing
```

```
> Just 10 >>= maybeHalf
```

```
Just 5
```

```
> Just 10 >>= maybeHalf >>= maybeHalf
```

```
Nothing
```

```
> Just 10 >>= maybeHalf >>= maybeHalf >>= maybeHalf
```

```
Nothing
```

Стандартные монады (Maybe)

```
maybeHalfN :: Int -> Int -> Maybe Int  
?
```

Стандартные монады (Maybe)

```
maybeHalfN :: Int -> Int -> Maybe Int
```

```
maybeHalfN 0 a = Just a
```

```
maybeHalfN n a = maybeHalf a >>= (\x -> maybeHalfN (n-1) x)
```

```
maybeHalfN n a = do
```

```
  result <- maybeHalf a
```

```
  maybeHalfN (n-1) a
```

Стандартные монады

Монада List (вычисления, которые могут возвращать 0 или более результатов)

В этой монаде значения представляют собой списки, которые можно интерпретировать как несколько возможных результатов одного вычисления. Если одно вычисление зависит от другого, то второе вычисление производится для каждого результата первого, и полученные результаты (второго вычисления) собираются в список.

```
return a = [a]  
params >>= f = concat [f x | x <- params]
```

Стандартные монады

Таким образом, `params >>= f :: [b]`, как и следовало ожидать – это список возможных результатов применения функции к каждому из вариантов входного аргумента.

Например,

```
let x = [1 .. 5] >>= (\x -> [x .. x+3 ])
```

```
pipe, F#  
[1,2,3,4] |> filter (fun x -> odd x)
```

```
["~/music", "~/work"] >>= getDirectoryContents =  
[  
  "~/music/Bach",  
  "~/music/Beethoven",  
  "~/music/Rammstein",  
  "~/work/projects",  
  "~/work/documents"  
]
```

Стандартные монады

```
(>>=) :: [a] -> (a -> [b]) -> [b]
```

```
hiAll :: Maybe String  
hiAll = Just "Hello" >>= (\x ->  
    Just " all" >>= (\y ->  
        Just (x ++ y)))
```

```
hiAll :: Maybe String  
hiAll = do  
    x <- Just "Hello"  
    y <- Just " all"  
    return (x ++ y) -- Just (x ++ y)
```

Стандартные монады

$(\gg=) :: [a] \rightarrow (a \rightarrow [b]) \rightarrow [b]$

$[(x, y) \mid x \leftarrow [1, 2, 3], y \leftarrow [1, 2, 3], x \neq y]$

```
do x <- [1, 2, 3]
  y <- [1, 2, 3]
  True <- return (x /= y)
  return (x, y)
```

```
[1, 2, 3] >>= (\x -> [1, 2, 3] >>= (\y -> return (x /= y) >>=
  (\r -> case r of
    True -> return (x, y)
    _     -> fail ""))))
```

\Rightarrow

Do notation

$\text{do } \{p\} = p$

$\text{do } \{p; \text{stmts}\} = p \gg \text{do } \{\text{stmts}\}$

$\text{do } \{x \leftarrow p; \text{stmts}\} = p \gg= \lambda x \rightarrow \text{do } \{\text{stmts}\}$

Стандартные монады

Монада IO

определяет операцию ($>>=$) как последовательное выполнение двух её операндов, а результат выполнения первого операнда последовательно передаётся во второй

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

m >> k = m >>= \_ -> k
```

return

numIO :: IO Int

numIO = return 42

IO

`putChar :: Char -> IO ()`

`putChar 'x'`

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

`putChar 'x' >> putChar '\n'`

`return :: a -> IO a`

`println :: String -> IO ()`

`println xs = ???`

IO

`putChar :: Char -> IO ()`

`putChar 'x'`

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

`putChar 'x' >> putChar '\n'`

`return :: a -> IO a`

`println :: String -> IO ()`

`println xs = foldr (>>) (return ()) (map putChar xs) >> putChar '\n'`

IO

`getChar :: IO Char`

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

`gl :: IO [Char]`

`gl = getChar >>= \x ->`

`if x == '\n' then return []`

`else getLine >>= \xs -> return (x:xs)`

do-notation

gl = do

 x <- getChar

 if x == '\n' then return []

 else do

 xs <- gl

 return (x:xs)

IO – функтор

```
instance Functor IO where
```

```
    fmap f action = do
```

```
        result <- action
```

```
        return (f result)
```

```
main = do
```

```
    line <- getLine
```

```
    let line' = (++ "!") line
```

```
    putStrLn line
```

```
main = do
```

```
    line <- fmap (++ "!") getLine -- line <- fmap ((++ "!") . reverse) getLine
```

```
    putStrLn line
```


IO – аппликативный функтор

```
instance Applicative IO where
```

```
  pure = return
```

```
  a <*> b = do
```

```
    f <- a
```

```
    x <- b
```

```
    return (f x)
```

```
concatLines :: IO String
```

```
concatLines = do
```

```
  a <- getLine
```

```
  b <- getLine
```

```
  return $ a ++ b
```

```
concatLines :: IO String
```

```
concatLines = (++) <$> getLine <*> getLine
```

IO

```
import Control.Monad  
import System.Random
```

```
randomRIO (1,6)
```

```
rollDiceIO :: IO (Int, Int)
```

```
rollDiceIO = liftM2 (,) (randomRIO (1,6)) (randomRIO (1,6))
```

```
rollDiceIO = (,) <$> (randomRIO (1,6)) <*> (randomRIO (1,6))
```

```
rollNDiceIO ?
```

IO

```
rollNDiceIO :: Int -> IO [Int]
rollNDiceIO 0 = return []
rollNDiceIO n = do
  randomRIO (1, 6) >>=
    (\num -> rollNDiceIO (n-1) >>=
      (\list -> return (num:list)))
```

IO

```
rollNDiceIO :: Int -> IO [Int]
```

```
rollNDiceIO 0 = return [0]
```

```
rollNDiceIO n = do
```

```
    num <- randomRIO (1, 6)
```

```
    list <- rollNDiceIO (n-1)
```

```
    return (num:list)
```

IO

КОММУТАТИВНОСТЬ

```
num <- randomRIO (1, 6)
```

```
list <- rollNDiceIO (n-1)
```

```
return (num:list)
```

```
list <- rollNDiceIO (n-1)
```

```
num <- randomRIO (1, 6)
```

```
return (num:list)
```

IO

```
rollNDiceIO :: Int -> IO [Int]
```

```
rollNDiceIO n = replicateM n (randomRIO (1, 6))
```

```
rollNDiceIO n = sequence $ map (\a -> randomRIO(1,6)) [1..n]
```

Control.Monad

`sequence :: Monad m => [m a] -> m [a]`

`mapM, mapM_ :: Monad m => (a -> m b) -> [a] -> m [b]`

`putStr :: String -> IO ()`

`foldM :: Monad m => (a -> b -> m a) -> a -> [b] -> m a`

`liftM :: Monad m => (a1 -> r) -> m a1 -> m r`

`ap :: Monad m => m (a -> b) -> m a -> m b`

`return f `ap` x1 `ap` ... `ap` xn`

`liftMn f x1 x2 ... xn`

`fail :: String -> m a`

Моноиды State

```
import Control.Monad.State

fact3 :: Integer -> State Integer Integer

fact3 0 = return 1

fact3 n = do
    let n' = n-1
    modify (+1)
    m <- fact3 n'
    let r = n*m
    modify (+1)
    return r

runState (fact3 10) 0
```


State

`type State s a = s -> (a,s)`

`put :: s -> State s ()`

`get :: State s s`

`state :: (s -> (a,s)) -> State s a`

`runState State s a -> (s -> (a, s))`

`evalState State s a -> s -> a`

State

```
newtype State s a = State { runState :: s -> (a, s) }
```

```
instance Monad (State s) where
```

```
return :: a -> State s a
```

```
return x = State ( \ st -> (x, st) )
```

```
(>>=) :: State s a -> (a -> State s b) -> State s b
```

```
processor >>= processorGenerator = State $ \ st ->
```

```
    let (x, st') = runState processor st
```

```
    in runState (processorGenerator x) st'
```

State

```
runState (return 'X') 1
```

```
runState get 1
```

```
runState (put 5) 1
```

```
runState (do { put 5; return 'X' }) 1
```

```
runState (put 5 >=> \_ -> return 'X') 1
```

```
postincrement :: State Int Int
```

```
postincrement = do { x <- get; put (x+1); return x }
```

```
runState postincrement 1
```

```
predecrement = do { x <- get; put (x-1); get }
```

```
runState predecrement 1
```

```
runState (modify (+1)) 1
```

```
runState (gets (+1)) 1
```

```
evalState (gets (+1)) 1
```

```
execState (gets (+1)) 1
```

State

`return :: a -> State s a`

`return x s = (x,s)`

`get :: State s s`

`get s = (s,s)`

`put :: s -> State s ()`

`put x s = ((),x)`

`modify :: (s -> s) -> State s ()`

`modify f = do { x <- get; put (f x) }`

`gets :: (s -> a) -> State s a`

`gets f = do { x <- get; return (f x) }`

State

```
import Control.Monad.State
list2tree xs = evalState (build (length xs)) xs
build :: Int -> State [a] (Tree a)
build 0 = return Empty
build 1 = do
    x:xs <- get
    put xs
    return (Node x Empty Empty)
build n = do
    x:xs <- get
    put xs
    let m = div (n-1) 2
    u <- build m
    v <- build (n-1-m)
    return (Node x u v)
```

State

data Tree a = Leaf a | Node (Tree a) (Tree a) ?

Random

```
import System.Random  
import Data.Char
```

```
type GeneratorState = State StdGen
```

```
type Generator = (GeneratorState Int, GeneratorState Int)
```

```
randomChar :: GeneratorState Char
```

```
randomChar = do
```

```
  gen <- get
```

```
  let (charInt, newGenerator) = randomR (97, 122) gen
```

```
  put newGenerator
```

```
  return (chr charInt)
```

```
evalState randomChar (mkStdGen 0)
```

Random

```
randomString :: GeneratorState String
```

```
randomString = do
```

```
  gen <- get
```

```
  let (v10, newGen) = randomR (1, 9) gen :: (Int, StdGen)
```

```
  case v10 of
```

```
    1 -> put newGen >> return []
```

```
    _ -> do
```

```
      let (x, newGen') = runState randomChar newGen
```

```
      let (xs, newGen'') = runState randomString newGen'
```

```
      put newGen''
```

```
      return (x:xs)
```