

# Функциональное программирование

## Лекция 10. Стандартные монады

Денис Николаевич Москвин

Кафедра математических и информационных технологий  
Санкт-Петербургского академического университета

27.04.2012

- 1 Монада IO: ввод-вывод
- 2 Монада Reader: чтение из окружения
- 3 Монада Writer: запись в лог
- 4 Монада State: изменяемое состояние

- 1 Монада IO: ввод-вывод
- 2 Монада Reader: чтение из окружения
- 3 Монада Writer: запись в лог
- 4 Монада State: изменяемое состояние

# Проблема ввода-вывода

В чистых языках, где значение функции зависит только от её параметров, ввод-вывод представляет собой проблему.

Функция

```
getCharFromConsole :: Char
```

**всегда** должна возвращать одно и то же!

Как с этим справиться?

# Проблема ввода-вывода

В чистых языках, где значение функции зависит только от её параметров, ввод-вывод представляет собой проблему.

Функция

```
getCharFromConsole :: Char
```

**всегда** должна возвращать одно и то же!

Как с этим справиться?

```
getCharFromConsole :: RealWorld -> (RealWorld, Char)
```

При этом доступ к значениям типа `RealWorld` должен быть ограничен.

- Значение типа IO — это вычисление, которое при выполнении может осуществлять действие ввода-вывода.
- Реализация в GHC (слегка упрощено)

```
newtype IO a = IO (RealWorld -> (RealWorld, a))
```

- Про тип RealWorld в документации сказано «deeply magical», он не экспортируется из модуля, поэтому программист не имеет к нему доступа.
- И это очень хорошо, поскольку один и тот же RealWorld нельзя использовать два раза!
- Единственный способ выполнить действие ввода-вывода — связать его с функцией main программы.

Слегка сжульничаем

```
type IO a = RealWorld -> (RealWorld, a)
```

```
instance Monad IO where
  return    :: a -> IO a
  return a  = \w -> (w,a)

  (>>=)     :: IO a -> (a -> IO b) -> IO b
  (>>=) m k = \w -> case m w of (w',a) -> k a w'
```

Гарантии, которые должны выполняться:

- Побочный эффект каждого действия происходит один раз.
- Побочные эффекты происходят в заданном порядке.

- Ввод:

```
getChar      :: IO Char
getLine      :: IO String
getContents  :: IO String
```

- Вывод:

```
putChar      :: Char -> IO ()
putStr, putStrLn :: String -> IO ()
print        :: Show a => a -> IO ()
```

- Ввод-вывод:

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



# Пример ввода-вывода

```
main = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn $ "Nice to meet you, " ++ name ++ "!"
```

Какой тип имеет main?

## Пример ввода-вывода

```
main = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn $ "Nice to meet you, " ++ name ++ "!"
```

Какой тип имеет main?

Подсказка:

```
main =
  putStrLn "What is your name?" >>
  getLine >>= \name ->
  putStrLn $ "Nice to meet you, " ++ name ++ "!"
```

Как, имея `getChar`, сделать `getLine`?

```
getLine' :: IO String
getLine' = do
  c <- getChar
  if c == '\n' then
    return []
  else do
    cs <- getLine'
    return (c:cs)
```

В `if...then...else` конструкции повторный вызов `do` необходим (например, из соображений типизации).

Как, имея putChar, сделать putStr?

```
putStr'      :: String -> IO ()  
putStr' []   = return ()  
putStr' (x:xs) = putChar x >> putStr' xs
```

Можно выделить общий шаблон свёртки

```
sequence_    :: Monad m => [m a] -> m ()  
sequence_ = foldr (>>) (return ())
```

Тогда

```
putStr''     :: String -> IO ()  
putStr'' = sequence_ . map putChar
```

## Устройство putStr (2)

Реализация

```
putStr'' :: String -> IO ()  
putStr'' = sequence_ . map putChar
```

тоже содержит обобщаемый шаблон. Имеется

```
mapM_      :: Monad m => (a -> m b) -> [a] -> m ()  
mapM_ f    = sequence_ . map f
```

Используя её, получим

```
putStr'''  :: String -> IO ()  
putStr''' = mapM_ putChar
```

Есть более «полновесные» аналоги `sequence_` и `mapM_`

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

```
mapM         :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f       = sequence . map f
```

## Сессия GHCi

```
*Fp10> mapM_ putChar "Hello"
Hello*Fp10> mapM putChar "Hello"
Hello[(),(),(),(),()]
```

- 1 Монада IO: ввод-вывод
- 2 Монада Reader: чтение из окружения
- 3 Монада Writer: запись в лог
- 4 Монада State: изменяемое состояние

Вычисление, допускающее чтение значений из разделяемого окружения.

```
instance Monad ((->) r) where
  return    :: a -> (r -> a)
  return x  = \_ -> x

  (>>=)     :: (r -> a) -> (a -> (r -> b)) -> (r -> b)
  m >>= k   = \e -> k (m e) e
```

- `return` — просто игнорирует окружение;
- `(>>=)` — передаёт полученное окружение в оба вычисления.



Для большей универсальности вводят тип

```
newtype Reader r a = Reader { runReader :: (r -> a) }
```

На самом деле тип `Reader r a` определён по-другому, нам интересен публичный интерфейс его сборки и разборки:

```
reader :: (r -> a) -> Reader r a  
runReader :: Reader r a -> r -> a
```

```
instance Monad (Reader r) where  
  return x = reader $ \e -> x  
  m >>= k = reader $ \e -> let v = runReader m e  
                              in runReader (k v) e
```

Простейший Reader выполняет вычисления в окружении типа Int

```
simpleReader :: Reader Int String
simpleReader = reader (\e -> "Environment is " ++ show e)
```

```
*Fp10> runReader simpleReader 42
"Environment is 42"
```

Функция `ask :: Reader r r` возвращает окружение

```
type User = String
type Password = String
type UsersTable = [(User,Password)]

pwds :: UsersTable
pwds = [("Bill","123"),("Ann","qwerty"),("John","2sRq8P")]

firstUser :: Reader UsersTable User
firstUser = do
  e <- ask
  return $ fst (head e)
```

```
*Fp10> runReader firstUser pwds
"Bill"
```

Функция `asks :: (r -> a) -> Reader r a` возвращает результат выполнения функции над окружением

```
getPwdLen :: User -> Reader UsersTable Int
getPwdLen person = do
  mbPwd <- asks $ lookup person
  let mbLen = fmap length mbPwd
      len = fromMaybe (-1) mbLen
  return len
```

```
*Fp10> runReader (getPwdLen "Ann") pwds
6
*Fp10> runReader (getPwdLen "Ann") []
-1
```

Функция `local :: (r -> r) -> Reader r a -> Reader r a` позволяет локально модифицировать окружение

```
usersCount :: Reader UsersTable Int
usersCount = asks length

localTest :: Reader UsersTable (Int,Int)
localTest = do
  count1 <- usersCount
  count2 <- local (("Mike","1"): ) usersCount
  return (count1, count2)
```

```
*Fp10> runReader localTest pwds
(3,4)
*Fp10> runReader localTest []
(0,1)
```

- 1 Монада IO: ввод-вывод
- 2 Монада Reader: чтение из окружения
- 3 Монада Writer: запись в лог**
- 4 Монада State: изменяемое состояние

Вычисление, допускающее запись в лог.

```
newtype Writer w a = Writer {runWriter :: (a, w)}  
  
writer      :: (a, w) -> Writer w a  
runWriter   :: Writer w a -> (a, w)  
execWriter  :: Writer w a -> w
```

Контекст Monoid обеспечивает полноценное конструирование лога.

```
instance (Monoid w) => Monad (Writer w) where  
  return x = writer (x, mempty)  
  m >>= k  = let (x,u) = runWriter m  
                (y,v) = runWriter $ k x  
                in writer (y, u `mappend` v)
```

Простейшие примеры:

## Сессия GHCi

```
*Fp10> runWriter (return 3 :: Writer String Int)
(3, "")
*Fp10> runWriter (return 3 :: Writer (Sum Int) Int)
(3, Sum {getSum = 0})
*Fp10> execWriter (return 3 :: Writer (Product Int) Int)
Product {getProduct = 1}
```



База данных для интернет-магазина «Овощи-Фрукты».

```
type Vegetable = String

type Price = Double
type Qty = Double
type Cost = Double

type PriceList = [(Vegetable,Price)]

prices :: PriceList
prices = [("Potato",13),("Tomato",55),("Apple",48)]
```

Функция `tell :: Monoid w => w -> Writer w ()` позволяет задать вывод

```
addVegetable :: Vegetable -> Qty
              -> Writer (Sum Cost) (Vegetable, Price)
addVegetable veg qty = do
  let pr = fromMaybe 0 $ lookup veg prices
      cost = qty * pr
  tell $ Sum cost
  return (veg, pr)
```

```
*Fp10> runWriter $ addVegetable "Apple" 100
(("Apple",48.0),Sum {getSum = 4800.0})
*Fp10> runWriter $ addVegetable "Pear" 100
(("Pear",0.0),Sum {getSum = 0.0})
```

```
myCart0 = do
  x1 <- addVegetable "Potato" 3.5
  x2 <- addVegetable "Tomato" 1.0
  x3 <- addVegetable "AGRH!!" 1.6
  return [x1,x2,x3]
```

Суммарная стоимость копится «за кадром»:

## Сессия GHCi

```
*Fp10> runWriter myCart0
([("Potato",13.0),("Tomato",55.0),("AGRH!!",0.0)],
Sum {getSum = 100.5})
```

Если хотим знать промежуточные стоимости, используем `listen :: Monoid w => Writer w a -> Writer w (a, w)`

```
myCart1 = do
  x1 <- listen $ addVegetable "Potato" 3.5
  x2 <- listen $ addVegetable "Tomato" 1.0
  x3 <- listen $ addVegetable "AGRH!!!" 1.6
  return [x1,x2,x3]
```

## Сессия GHCi

```
*Fp10> runWriter myCart1
([(("Potato",13.0),Sum {getSum = 45.5}),(("Tomato",55.0),
Sum {getSum = 55.0}),(("AGRH!!!",0.0),Sum {getSum = 0.0})],
Sum {getSum = 100.5})
```

Есть более гибкая альтернатива

```
listens :: Monoid w => (w -> b) -> Writer w a -> Writer w (a, b)
```

```
myCart1' = do
  x1 <- listens getSum $ addVegetable "Potato" 3.5
  x2 <- listens getSum $ addVegetable "Tomato" 1.0
  x3 <- listens getSum $ addVegetable "AGRH!!" 1.6
  return [x1,x2,x3]
```

## Сессия GHCi

```
*Fp10> runWriter myCart1'
([(("Potato",13.0),45.5),(("Tomato",55.0),55.0),
(("AGRH!!",0.0),0.0)],Sum {getSum = 100.5})
```

Для модификации лога используем

```
sensor :: Monoid w => (w -> w) -> Writer w a -> Writer w a
```

```
myCart0' = sensor (discount 10) myCart0
```

```
discount proc (Sum x) = Sum $ if x < 100 then x  
                        else x * (100 - proc) / 100
```

## Сессия GHCi

```
*Fp10> execWriter myCart0  
Sum {getSum = 100.5}  
*Fp10> execWriter myCart0'  
Sum {getSum = 90.45}
```

- 1 Монада IO: ввод-вывод
- 2 Монада Reader: чтение из окружения
- 3 Монада Writer: запись в лог
- 4 Монада State: изменяемое состояние

Вычисление, позволяющее работать с изменяемым состоянием.

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

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

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

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

```
  return x = state $ \st -> (x,st)
```

```
  m >>= k = state $ \st -> let (x,st') = runState m st
                               m'       = k x
                               in runState m' st'
```

return упаковывает значение в функцию, не меняющую состояние. (>>=) передаёт «обновлённое» первым вычислением состояние во второе вычисление.



# Монада state: запуск вычислений

Помимо пары, в вычислении с состоянием можно получить либо только итоговое состояние, либо только итоговое значение вычисления:

```
runState  :: State s a -> s -> (a,s)
execState :: State s a -> s -> s
evalState :: State s a -> s -> a
```

## Сессия GHCi

```
*Fp10> runState (return 3 :: State String Int) "Hi, State!"
(3,"Hi, State!")
*Fp10> execState (return 3 :: State String Int) "Hi, State!"
"Hi, State!"
*Fp10> evalState (return 3 :: State String Int) "Hi, State!"
3
```

## Специальные функции для работы с состоянием

```
get      :: State s s
get      = state $ \s -> (s,s)

put      :: s -> State s ()
put s    = state $ \_ -> ((),s)

modify   :: (s -> s) -> State s ()
modify f = do s <- get
             put (f s)

gets    :: (s -> a) -> State s a
gets f  = do s <- get
             return (f s)
```

```
tick :: State Int Int
tick = do  n <- get
           put (n+1)
           return n
```

## Сессия GHCi

```
*Fp10> runState tick 3
(3,4)
```

```
succ' :: Int -> Int
succ' n = execState tick n

plus :: Int -> Int -> Int
plus n x = execState (sequence $ replicate n tick) x
```

```
plus :: Int -> Int -> Int
plus n x = execState (sequence $ replicate n tick) x
```

Конструкция `sequence . replicate n` встречается довольно часто, поэтому в `Control.Monad` определено

```
replicateM    :: (Monad m) => Int -> m a -> m [a]
replicateM n = sequence . replicate n
```

Тогда

```
plus' :: Int -> Int -> Int
plus' n x = execState (replicateM n tick) x
```