# Функциональное программирование
# Лекция 11. Стандартные монады

Денис Николаевич Москвин

СПбГУ, факультет МКН,
бакалавриат «Современное программирование», 2 курс

20.11.2025

# План лекции

1. Монада `Reader`: чтение из окружения
2. Монада `Writer`: запись в лог
3. Монада `State`: изменяемое состояние
4. Монада `IO`: ввод-вывод
5. Классы `Alternative` и `MonadPlus`
6. Монада `Except`: возбуждение и перехват исключений

# План лекции

# Монада Reader/Environment

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.

```
GHCi> do {a <- (^2); b <- (*5); return (a + b)} $ 3
24
```

# Тип и монада Reader

Для большей универсальности в пакетах `transformers` / `mtl` 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 $ \_ -> x
  m >>= k   =  reader $ \e -> let v = runReader m e
                              in runReader (k v) e
```

# Пример монады Reader

Простейший Reader, отчитывающийся о значении окружения

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

```
GHCi> runReader simpleReader 42
"Environment is 42"
GHCi> runReader simpleReader True
"Environment is True"
```

# Стандартный интерфейс: функция ask

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","2sR")]

firstUser = do
  e <- ask
  let name = fst (head e)
  return name
```

```
GHCi> runReader firstUser pwds
"Bill"
```

# Стандартный интерфейс: функция asks

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
  let len = fromMaybe (-1) mbLen
  return len
```

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

# Стандартный интерфейс: функция local

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)
```

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

# План лекции

# Монада Writer

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)
```

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)
```

# Запуск вычислений в монаде Writer

Есть два способа запустить вычисление

```
runWriter  ::  Writer w a -> (a, w)

execWriter ::  Writer w a -> w
```

Простейшие примеры:

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

# Монада Writer: расширенный пример

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",55),("Tomato",120),("Apple",90)]
```

# Стандартный интерфейс: функция tell

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
  let cost = qty * pr
  tell $ Sum cost
  return (veg, pr)
```

```
GHCi> runWriter $ addVegetable "Apple" 10
(("Apple",90.0),Sum {getSum = 900.0})
GHCi> runWriter $ addVegetable "Pear" 10
(("Pear",0.0),Sum {getSum = 0.0})
```

# Использование addVegetable

```
myCart0 :: Writer (Sum Cost) [(Vegetable, Price)]
myCart0 = do
  x1 <- addVegetable "Potato" 5
  x2 <- addVegetable "Tomato" 2
  x3 <- addVegetable "Pear" 1.5
  return [x1,x2,x3]
```

Суммарная стоимость копится «за кадром»:

```
GHCi> runWriter myCart0
([("Potato",55.0),("Tomato",120.0),("Pear",0.0)],Sum {getSu
m = 515.0})
GHCi> execWriter myCart0
Sum {getSum = 515.0}
```

# Стандартный интерфейс: функция listen

Если хотим знать промежуточные стоимости, используем

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

```
myCart1 :: Writer (Sum Cost)
                  [((Vegetable, Price), Sum Cost)]
myCart1 = do
  x1 <- listen $ addVegetable "Potato" 5
  x2 <- listen $ addVegetable "Tomato" 2
  x3 <- listen $ addVegetable "Pear" 1.5
  return [x1,x2,x3]
```

```
GHCi> runWriter myCart1
([(("Potato",55.0),Sum {getSum = 275.0}),(("Tomato",120.0),
Sum {getSum = 240.0}),(("Pear",0.0),Sum {getSum = 0.0})],Su
m {getSum = 515.0})
```

# Стандартный интерфейс: функция listens

Есть более гибкая альтернатива

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

```
myCart1' :: Writer (Sum Cost)
                   [((Vegetable, Price), Cost)]
myCart1' = do
  x1 <- listens getSum $ addVegetable "Potato" 5
  x2 <- listens getSum $ addVegetable "Tomato" 2
  x3 <- listens getSum $ addVegetable "Pear" 1.5
  return [x1,x2,x3]
```

```
GHCi> runWriter myCart1'
([(("Potato",55.0),275.0),(("Tomato",120.0),240.0),(("Pear"
,0.0),0.0)],Sum {getSum = 515.0})
```

# Стандартный интерфейс: функция censor

Для модификации лога используем

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

```
myCart0' :: Writer (Sum Cost) [(Vegetable, Price)]
myCart0' = censor (discount 10) myCart0

discount :: Double -> Sum Cost -> Sum Cost
discount proc s@(Sum x)
  | x < 500    = s
  | otherwise = Sum $ x * (100 - proc) / 100
```

```
GHCi> execWriter myCart0
Sum {getSum = 515.0}
GHCi> execWriter myCart0'
Sum {getSum = 463.5}
```

# План лекции

# Монада State

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 значенние вычисления:

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

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

# Стандартный интерфейс для монады State

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)
```

# Монада State: примеры

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

```
GHCi> 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
```

# Полезные функции

```
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 :: Applicative m => Int -> m a -> m [a]
replicateM n  =  sequenceA . replicate n
```

(Раньше контекст был `Monad m`).
Тогда

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

# Другие инструменты для работы с состоянием

- `State` это чистая функциональная конструкция.
- Монада `ST` позволяет локально работать с настоящим изменяемым состоянием. Имеется удобный вспомогательный ссылочный тип `STRef`. Локальность обеспечивается типом второго ранга

  ```
  runST :: (forall s. ST s a) -> a
  ```
- `IORef` это `STRef` без локальности и соответствующих гарантий безопасности.
- `MVar` это `IORef` с блокировками, поддерживающими конкурентный доступ.
- `TVar` это изменяемые ячейки памяти в рамках STM (Software transactional memory).
- Подробнее см. у Саймона Марлоу [Mar13] (русский перевод [С.14]).

# План лекции

# 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.

# Тип IO

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

```
newtype IO a =
  IO (State# RealWorld -> (# State# 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.

# Монада IO: недоступные внутренности

Работаем с упрощенной версией

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

unIO :: IO a -> SRW -> (SRW, a)
unIO (IO f) = f
```

```
instance  Monad IO  where
  return x = IO $ \w -> (w, x)

  (>>=) (IO m) k = IO $
    \w -> case m w of (new_w, a) -> unIO (k a) new_w
```

Гарантии, которые должны выполняться:

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

# Основные функции консольного ввода-вывода

- 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

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

Какой тип имеет `main`?

# Пример ввода-вывода

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

Какой тип имеет `main`?

Подсказка:

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

# Устройство getLine

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).

# Устройство putStr

Как, имея putChar, сделать putStr?

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

Можно выделить общий шаблон свёртки

```
sequence_ :: (Foldable t, Monad m) => t (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_ :: (Foldable t, Monad m)
          => (a -> m b) -> t a -> m ()
mapM_ f  =  sequence_ . fmap f
```

Используя её, получим

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

# Полезные функции

Есть более «полновесные» аналоги sequence_ и mapM_

```
sequence :: (Traversable t, Monad m)
                 => t (m a) -> m (t a)
sequence  =  sequenceA
```

```
mapM :: (Traversable t, Monad m)
           => (a -> m b) -> t a -> m (t b)
mapM f  =  sequence . fmap f    -- traverse
```

```
GHCi> mapM_ putChar "Hello"
HelloGHCi> mapM putChar "Hello"
Hello[(),(),(),(),()]
```

# План лекции

# Класс Alternative

`Alternative` — это `Applicative` с дополнительной моноидальной операцией.

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

```
instance Alternative [] where
  empty = []
  (<|>) = (++)
instance Alternative Maybe where
  empty  =  Nothing
  Nothing    <|> m  =  m
  m@(Just _) <|> _  =  m
```

```
GHCi> Nothing <|> Just 3 <|> Just 5 <|> Nothing
Just 3
```

# Представитель Alternative для IO

Метод empty возбуждает исключение ввода-вывода.
Метод <|> при ошибке в левом аргументе отбрасывает ее и запускает правый.

```
instance Alternative IO where
  empty :: IO a
  empty = failIO "mzero"

  (<|>) :: IO a -> IO a -> IO a
  m <|> n = m `catchException` \(_ :: IOError) -> n
```

Если первый файл присутствует, а второго нет:

```
GHCi> readFile "1.txt" <|> return "File doesn't exist."
"Contents.\nAlso contents."
GHCi> readFile "2.txt" <|> return "File doesn't exist."
"File doesn't exist."
```

# Класс MonadPlus

```
class (Alternative m, Monad m) => MonadPlus m where
   mzero :: m a
   mzero = empty
   mplus :: m a -> m a -> m a
   mplus = (<|>)
```

Минимальное полное определение: ничего не делать.

```
instance MonadPlus []

instance MonadPlus Maybe

instance MonadPlus IO
```

# Законы MonadPlus

Помимо унаследованных «моноидальных» законов требуют выполнения

## Left and Right Zero

```
mzero >>= k   ≡   mzero
v >> mzero    ≡   mzero
```

и по крайней мере одного из двух

## Left Distribution

```
(a `mplus` b) >>= k   ≡   (a >>= k) `mplus` (b >>= k)
```

## Left Catch law

```
return a `mplus` b   ≡   return a
```

# Использование MonadPlus

```haskell
-- Haskell 2010 :: MonadPlus m   => Bool -> m ()
guard           :: Alternative f => Bool -> f ()
guard True      =  pure ()
guard False     =  empty
```

```haskell
pythags = do  z <- [1..]
              x <- [1..z]
              y <- [x..z]
              guard (x^2 + y^2 == z^2)
              return (x, y, z)
```

```
GHCi> take 5 pythags
[(3,4,5),(6,8,10),(5,12,13),(9,12,15),(8,15,17)]
```

# Использование MonadPlus (2)

```haskell
asum :: (Foldable t, Alternative f) => t (f a) -> f a
asum = foldr (<|>) empty

msum :: (Foldable t, MonadPlus m) => t (m a) -> m a
msum = asum      -- foldr mplus mzero
```

```haskell
mfilter :: MonadPlus m => (a -> Bool) -> m a -> m a
mfilter p ma = do
  a <- ma
  if p a
  then return a
  else mzero
```

# План лекции

# Монада Except

Вычисление, допускающее возбуждение и перехват исключений.

```
newtype Except e a = Except { runExcept :: Either e a }

except :: Either e a -> Except e a
except = Except
```

```
instance Monad (Except e) where
  return :: a -> Except e a
  return a = except $ Right a

  (>>=) :: Except e a -> (a -> Except e b) -> Except e b
  m >>= k  = case runExcept m of
    Left e  -> except $ Left e
    Right x -> k x
```

# Стандартный интерфейс

```
throwE :: e -> Except e a
throwE = except . Left

catchE :: Except e a -> (e -> Except e' a)
                                -> Except e' a
m `catchE` h = case runExcept m of
  Left  l -> h l
  Right r -> except $ Right r
```

Использование

```
do { action1; action2; action3 } `catchE` handler
```

В библиотеке mtl по историческим причинам `throwError` и `catchError`.
Разумного представителя `MonadFail` написать невозможно.

# Пример использования

```haskell
data DivByError = ErrZero | Oth String deriving (Eq,Show)

(/?) :: Double -> Double -> Except DivByError Double
_ /? 0 = throwE ErrZero
x /? y = return $ x / y

example1 :: Double -> Double -> String
example1 x y = fromRight "" . runExcept $
  action `catchE` handler where
    action = do
      q <- x /? y
      return $ show q
    handler = return . show
```

```
GHCi> example1 5 2
"2.5"
GHCi> example1 5 0
"ErrZero"
```

# Расширение функциональности

Если хотим использовать функциональность `MonadPlus`, то есть `guard`, `msum`, `mfilter`, нужно сделать `Except e` представителем:

```
instance Monoid e => MonadPlus (Except e) where
  mzero = except $ Left mempty
  x `mplus` y = except $ let alt = runExcept y in
    case runExcept x of
      Left e  -> either (Left . mappend e) Right alt
      r       -> r
```

Семантика:

- `mzero` — ошибка по умолчанию для `guard`, задается `mempty`;
- `mplus` — накапливает ошибки слева направо, но если происходит удачная попытка, то возвращает удачу.

# Расширение функциональности, пример

```
instance Monoid DivByError where
  mempty = Oth ""
  Oth s1 `mappend` Oth s2     = Oth $ s1 ++ s2
  Oth s1 `mappend` ErrZero    = Oth $ s1 ++ "zero;"
  ErrZero  `mappend` Oth s2   = Oth $ "zero;" ++ s2
  ErrZero  `mappend` ErrZero  = Oth $ "zero;zero"
```

```
example2 :: Double -> Double -> String
example2 x y = fromRight "" . runExcept $
  action `catchE` handler where
    action = do
      q <- x /? y
      guard $ y >= 0
      return $ show q
    handler = return . show
```

# Расширение функциональности, пример (2)

```
example2 :: Double -> Double -> String
example2 x y = fromRight "" . runExcept $
  action `catchE` handler where
    action = do
      q <- x /? y
      guard $ y >= 0
      return $ show q
    handler = return . show
```

```
GHCi> example2 5 0
"ErrZero"
GHCi> example2 5 (-2)
"Oth \"\""
GHCi> runExcept $ msum [5/?0, 7/?0, 2/?0]
Left (Oth "zero;zero;zero;")
GHCi> runExcept $ msum [5/?0, 7/?0, 2/?4]
Right 0.5
```

# Литература


Марлоу С.

*Параллельное и конкурентное программирование на языке Haskell*.

М.:ДМК Пресс, 2014.

S. Marlow.

*Parallel and Concurrent Programming in Haskell*.

O'Reilly, 2013.