25 декабря 2021 г.

Общую информацию о монадах можно найти в билете 19, здесь ее дублировать, кажется, мало смысла.

Монада Reader

Нужен, если мы хотим передать что-нибудь свыше в множество функций. Понять поможет пример с хабра, в лекции есть похожий, тоже +- простой.

```
greeter :: Reader String String
greeter = do
    name <- ask
    return ("hello, " ++ name ++ "!")
```

То есть есть функция, в ней мы хотим уметь принять какой-то параметр из внешнего мира (ну консоли например), и засунуть в возвращаемое значение. Это делаем с помощью монады.

Теперь немного кода объявлений для монады:

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

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
```

Какие у монады есть функции?

1. ask: возвращает окружение. Собственно, делает запрос к внешнему миру и забирает оттуда значение. Была использована в самом первом примере. Само окружение это не обязательно консоль, мы можем в качестве окружения передать функции список или что-то аналогичное.

2. asks: возвращает результат выполнения функции над окружением. Что происходит в примере с лекции. Возвращается результат - второй элемент пары при поиске по первому элементу, принимается asks, обрабатывается.

3. local: позволяет менять передаваемое окружение. Принимает функцию и окружение, производит манипуляции с окружением, возаращает результат.

Монада Writer

Монада Writer хранит в себе два параметра – записываемый лог и вычисляемое в ходе программы значения. Дает возможность, собственно, записывать вычисления в лог, потом его выводить при необходимости, при этом не делая код слишком громоздким.

```
-- объявление

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

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

-- реализуем Monoid для удобства

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)
```

Соответственно, из реализации можно понять, что при операции »= будет происходить последовательное вычисление и дозапись в лог. Запустить процесс вычисления результата функции можно так:

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

execWriter :: Writer w a -> w

-- пример, после которого становится немного понятнее

-- завели аналог бд

type Vegetable = String
type Price = Double
type Qty = Double
```

```haskell
type Cost = Double
type PriceList = [(Vegetable,Price)]

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

-- завели функцию, которая работает с монадой
-- fromMaybe - вернет 0, если к ней передать Nothing, иначе вернет a из Maybe a
-- tell -- запись в лог
-- вернем пару добавленного и лога

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

-- развернем это добавление в функцию, которая сделает их много и увидим
-- полноценный лог

myCart0 :: Writer (Sum Cost) [(Vegetable, Price)]
myCart0 = do
    x1 <- addVegetable "Potato" 3.5
    x2 <- addVegetable "Tomato" 1.0
    x3 <- addVegetable "AGRH!!" 1.6
    return [x1,x2,x3]

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

Кроме того, есть ещё ряд функций у монады. Копипастить определения и примеры из слайдов я уже не буду, так что просто поясню, что они делают.

1. listen: позволяет хранить не только лог, но еще и промежуточные результаты вычислений, запихнет все в список
2. listens: улучшенная версия listen. Делает абсолютно тоже самое, но более читаемо.

3. censor: позволяет модифицировать лог у какой-то функции. Например, функция уже есть в устоявшемся виде, мы ее не хотим менять. И хотим написать функцию, которая ко всем промежуточным результатам исходной функции будет прибавлять 5, или отнимать 10 процентов, или делать еще что-нибудь похожее. Используем censor для удобной модификации.

Монада State

Позволяет работать с изменяемым состоянием. Идейно: как будто комбинация Reader и Writer. И читаем, и пишем. На этом этапе интернет перестал давать полезную информацию...

Объявления:

```
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'

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

-- ура, примеры. Просто три разные функции, которые возвращают ответ и лог/состояние, либо т

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

-- еще вводятся функции для работы со State

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

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

```haskell
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)
```

Что делают функции?

1. get: получает состояние (как ask в Reader)
2. put: меняет состояние (как tell, но без дозаписи)
3. modify: новое состояние является результатом применения функции к старому состоянию. Старое состояние при этом пропадает.
4. gets: вынимает из состояния какую-либо информацию с фотощью дополнительно передаваемой функции

И еще простой пример от Москвина:

```haskell
-- вытаскиваем значение снаружи, вкладываем n+1 в состояние
tick :: State Int Int
tick = do n <- get
          put (n + 1)
          return n

GHCi> runState tick 3
(3,4)
```

Монада Except

Монада хранит в себе вычисляемое значение и умеет бросать исключение, либо возвращать значение.

Объявления:

```haskell
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 $ Right a
    m >>= k = case runExcept m of
```

```
        Left e -> Except $ Left e
        Right x -> k x

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
```

Пример у Москвина какой-то сложный, вот пример попроще из дз

```
{-# LANGUAGE FlexibleContexts #-}
import Control.Monad.Except

data ListIndexError =
  ErrTooLargeIndex Int
  | ErrNegativeIndex
  | OtherErr String
  deriving (Eq, Show)

infixl 9 !!!

(!!!) :: MonadError ListIndexError m => [a] -> Int -> m a
xs !!! n  | n < 0 = throwError ErrNegativeIndex
          | length (take (n+1) xs) == (n+1) = return $ xs!!n
          | otherwise  =  throwError (ErrTooLargeIndex n)
```

тут мы реализовали оператор !!!, который кидает ошибку, если выходим за границы массива. Собственно, можно увидеть, как мы выбрасываем ошибку. Важно! Снаружи в коде эту ошибку надо ловить через catchError.

Пример от Москвина:

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

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

example0 :: Double -> Double -> Except DivByError String
example0 x y = action `catchE` handler where
    action = do q <- x /? y
                return $ show q
    handler = return . show

GHCi> runExcept $ example0 5 2
```

```
Right "2.5"
GHCi> runExcept $ example0 5 0
Right "ErrZero"
```