monads.lhs

```
> module Monads where

> import Data.Semigroup
```

Класс Monad

Вот знакомые нам методы из `Functor` и `Applicative`:

```
(<$>) ::   (a -> b) -> f a -> f b
(<*>) :: f (a -> b) -> f a -> f b
```

А ещё мы умеем композировать чистые функции:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

Часто функции, имеющие какой-то дополнительный эффект, кроме возвращаемого значения, часто принимают на вход обычное значение. Хочется уметь композировать такие функции.

Для этого используется класс `Monad`, а конкретно — метод

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

(читается “bind”, “байнд”).

Используется этот метод так:

```
move :: Direction -> State -> Maybe State
move = ...

moves :: State -> Maybe State
moves s = pure s >>= move Up >>= move Down >>= move Left
-- вот и композирование действий с эффектом!
```

Сам класс выглядит так:

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

  return :: a -> m a
  return = pure
  -- этот метод "сложился" исторически, сейчас его не реализуют

  (>>)  :: m a -> m b -> m b
  x >> y = x >>= \_ -> y
  -- этот метод отбрасывает значение и используется, когда нужно
  -- соединить только действия с эффектом.

  fail :: String -> m a
  -- этот метод останавливает вычисление с "ошибкой". Реализация
  -- ошибочности будет разной для разных типов.

> -- Пример использования (>>) с IO
> p123 :: IO ()
> p123 = putStr "Hello, ">> putStr "World" >> putStrLn "!"

> -- Пример использования (>>=) с IO
> -- getLine :: IO String
> -- putStrLn :: String -> IO ()
> echoLine :: IO ()
> echoLine = getLine >>= putStrLn
```

do-нотация

Вот так выглядит “процедурный” код, когда мы часть действий выполняем ради эффекта, а другая часть возвращает какие-то значения:

```
> greet :: IO ()
> greet =
>   putStr "Name: "         >>  (
>     getLine               >>= (\name    ->
>       putStr "Surname: " >>  (
>         getLine           >>= (\surname ->
>           let msg = "Hello, " ++ name ++ " " ++ surname ++ "!"
>           in putStrLn msg
>         )
>       )
>     )
>   )
```

Да, выглядит громоздко, но это всё ещё обычно выражение, составленное за счёт композирования действий. Для упрощения написания подобного кода существует специальный синтаксис — “do-нотация”:

```
> greet' :: IO ()
> greet' = do
>   putStr "Name: "
>   name <- getLine
>   -- Выглядит, как присваивание, но помните, это неявная лямбда!
>   -- Это тоже байнд.
>   putStr "Surname: "
>   surname <- getLine
>   let msg = "Hello, " ++ name ++ " " ++ surname ++ "!"
>   -- "in" нет, есть только дача определений
>   putStrLn msg
```

Выше код был мономорфный, но монадический код может быть и полиморфным. Полиморфны и всяческие “искоробочные” комбинаторы, помогающие монадический код. Примеры:

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

Когда же использовать do-нотацию, а когда операторы? Логика такая. Если код у вас линейный и похож на конвеер, то стоит писать цепочки:

```
game = pure s >>= move Up >>= move Down >>= move Left
```

Если же у вас предполагаются ветвления, то стоит подумать о do-нотации:

```
game s = do
  s1 <- move Up s
  s2 <- move Down s1
  s3 <- if s1 /= s2
     then fail "oops!"
     else pure s1
  pure (s1, s2, s3)
```

Каноничные монады: Identity

Эта монада “не делает ничего”. Как функция `id`. Нужна для композирования с другими эффектами.

```
> newtype Identity a = Identity { runIdentity :: a }

> instance Functor Identity where
>   fmap f = Identity . f . runIdentity

> instance Applicative Identity where
>   pure = Identity
>   f <*> x = Identity $ runIdentity f (runIdentity x)

> instance Monad Identity where
>   x >>= f = f (runIdentity x)
```

Примеры использования:

```
> i1, i2 :: Int
> i1 = runIdentity $ (+) <$> pure 1 <*> pure 2
> i2 = runIdentity $ do
>   x <- pure 1
>   y <- pure 2
>   pure (x + y)
```

Каноничные монады: Reader

`Reader` обобщает вычисления, зависящие от некоего “окружения”. Окружение доступно в любом месте вычисления, при этом явно передавать его из функции в функцию его не нужно.

```
> newtype Reader r a = Reader
>   { runReader :: r -> a
>   -- ^ та самая функция из окружения в результат!
>   }

> instance Functor (Reader r) where
>   fmap f rx = Reader $ \env ->
>     let x = runReader rx env
>     in f x

> instance Applicative (Reader r) where
>   pure x = Reader $ \_ -> x
>   rf <*> rx = Reader $ \env ->
>     let f = runReader rf env
>         x = runReader rx env
>     in f x

> instance Monad (Reader r) where
>   rx >>= f = Reader $ \env ->
>     let x = runReader rx env
>     in runReader (f x) env
```

“Рабочая лошадка” — функция, которая запрашивает окружение:

```
> ask :: Reader env env
> ask = Reader $ \env -> env
```

Пример использования:

```
> greetR :: Reader String String
> greetR = do
>    let greeting = "Hello, "
>    name <- doubleName
>    pure $ greeting ++ name
>  where
>    doubleName = do
>      name <- ask
>      name2 <- ask
>      pure $ name ++ name2

> runReader greetR "Bob"
"Hello, BobBob"
```

Каноничные монады. Writer

`Writer` обобщает накопление некоего результата в процессе выполнения комплексного вычисления.

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

> instance Functor (Writer w) where
>   fmap f wx = Writer $
>     let (x, w) = runWriter wx
>     in (f x, w)

> instance Monoid w => Applicative (Writer w) where
>   pure x = Writer (x, mempty)
>   wf <*> wx = Writer $
>     let
>       (f, w1) = runWriter wf
>       (x, w2) = runWriter wx
>     in (f x, w1 <> w2)

> instance Monoid w => Monad (Writer w) where
>   wx >>= f = Writer $
>     let
>       (x, w1)  = runWriter wx
>       (x', w2) = runWriter (f x)
>     in (x', w1 <> w2)
```

“Рабочая лошадка” `Writer` — функция `tell`:

```
> tell :: Monoid w => w -> Writer w ()
> tell x = Writer ((), x)
```

Так как `Writer` “пишет” в `Monoid`, то и накапливать можно разные результаты:

```
> writeSometing :: Writer (Max Int, Min Int, [Int]) Int
> writeSometing = do
>     _ <- makeFrom 7
>     x <- makeFrom (2 :: Int)
>     y <- makeFrom 5
>     _ <- makeFrom 15
>     pure $ x + y
>   where
>     makeFrom n = do
>       tell (Max n, Min n, [n])
>       pure n

> runWriter writeSometing
(7,(Max {getMax = 15},Min {getMin = 2},[7,2,5,15]))
```

Задания

local

Реализуйте функцию `local`:

```
local :: (r -> r) -> Reader r a -> Reader r a
```

Эта функция позволяет изменить окружение для подвычисления:

```
foo = do
  -- тут окружение -- оригинальное
  -- ...

  local (+ 1) $ do
    -- ...
    env <- ask
    -- здесь значение окружения больше на единицу
  -- ...
  -- здесь окружение опять старое
```

censor

Реализуйте функцию `censor`:

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

Эта функция позволяет изменить значение, накопленное при выполнении подвычисления:

```
mute = censor (const mempty)

bar = do
  tell "A"
  mute $ do
    tell "B"
    tell "C"
  tell "D"

-- snd (runWriter bar) == "AD"
-- всё, что писалось "под" mute, было отброшено
```