

## Курс: Функциональное программирование

### Практика 10. Монады

#### Разминка

- Устно вычислите значения выражений и проверьте результат в GHCi:

```
Just 17 >=> \x -> Just (x < 21)

[1,2,3] >=> \x -> [x, 10 * x]

[1,2] >=> \n -> "ab" >=> \c -> return (n,c)
```

- Запишите приведенные выше примеры в `do`-нотации.
- Что вернет вызов и почему

```
GHCi> (\bs -> do {b<-bs; when b []; return 42}) [True,False,False]
```

#### Монада списка

- Напишите реализацию функций `filter` и `replicate`, используя монаду списка и `do`-нотацию.

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' p xs = do undefined

replicate' :: Int -> a -> [a]
replicate' n x = do undefined
```

## Связь `Monad` и `Functor`

► Покажите, что оператор `(>.) :: Functor f => f a -> b -> f b` из модуля `Data.Functor` может быть выражен через монады:

```
(>.) :: Monad m => m a -> b -> m b
xs > . y = undefined
```

Для проверки

```
GHCI> [1,2,3] > . 'a'
"aaa"
```

► Покажите, что каждая монада — это функтор. Для этого выразите `fmap` через `>=>` и `return`:

```
fmap' :: Monad m => (a -> b) -> m a -> m b
fmap' f xs = undefined
```

► Отметим, что для полноценного доказательства того факта, что каждая монада — это функтор следует еще проверить выполнение законов класса `Functor`. Факультативно, но весьма желательно.

► Запишите вашу реализацию `fmap`, используя `do`-нотацию.

## Связь `Monad` и `Applicative`

► Покажите, что оператор `(>*) :: Applicative f => f a -> f b -> f b` может быть выражен через монады:

```
(>*) :: Monad m => m a -> m b -> m b
xs > * . ys = do undefined
```

Для проверки

```
GHCI> [1,2,3] > * . "ab"
"ababab"
```

- Покажите, что `liftA2` тоже может быть выражена через монады:

```
liftA2'  :: Monad m => (a -> b -> c) -> m a -> m b -> m c
liftA2' f xs ys = do undefined
```

Для проверки

```
GHCi> liftA2' (+) [10,20] [1,2]
[11,12,21,22]
```

- Покажите, что каждая монада — это аппликативный функтор. Для этого выразите `<*>` на языке монад:

```
(<*>.) :: Monad m => m (a -> b) -> m a -> m b
fs <*>. xs = do undefined
```

- Запишите эту реализацию `<*>.`, через `>>=` и `return`, не используя `do`-нотацию.

- Отметим, что для полноценного доказательства, что каждая монада — это аппликативный функтор, следует еще проверить выполнение законов класса `Applicative` для вашей реализации. Факультативно.

## Монадические комбинаторы

В модуле `Control.Monad` определены полезные комбинаторы:

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

(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> (a -> m c)
(<=<) = flip (>=>)

join  :: Monad m => m (m a) -> m a
```

«Рыбки» определяют композицию стрелок Клейсли, а `join` заменяет `>>=` в альтернативном (теоретико-категориальном) определении монады.

- Вычислите значения выражений в GHCi:

```

replicate 2 >=> replicate 3 $ 'x'

(\x -> [x,x+10]) >=> (\x -> [x,2*x]) $ 1

join ["aaa","bb"]

```

- Выразите `>=>` через `>>=`.
- Выразите `join` через `>>=`.
- Запишите `join` в `do`-нотации.

## Прочие функции для работы с монадами

Полное определение класса `Traversable` содержит монадические эквиваленты основных функций:

```

class (Functor t, Foldable t) => Traversable t where
    traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
    traverse f = sequenceA . fmap f

    sequenceA :: Applicative f => t (f a) -> f (t a)
    sequenceA = traverse id

    mapM :: Monad m => (a -> m b) -> t a -> m (t b)
    mapM = traverse

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

```

В модуле `Control.Monad` имеются обобщения стандартных функций над списками. В современном Haskell контекст первых трех был обобщен с `Monad` до `Applicative`, а в последней список обобщился до `Foldable`

```
replicateM :: Applicative m => Int -> m a -> m [a]
replicateM n xs = sequenceA (replicate n xs)
```

```
GHCi> replicateM 3 (Just 42)
Just [42,42,42]
GHCi> replicateM 2 [1,2,3]
[[1,1],[1,2],[1,3],[2,1],[2,2],[2,3],[3,1],[3,2],[3,3]]
GHCi> replicateM 2 (ZipList [1,2,3])
ZipList {getZipList = [[1,1],[2,2],[3,3]]}
```

```
zipWithM :: Applicative m => (a -> b -> m c) -> [a] -> [b] -> m [c]
zipWithM f xs ys = sequenceA (zipWith f xs ys)
```

```
GHCi> zipWithM (\x y -> Just (x+y)) [1,2,3] [10,20,30]
Just [11,22,33]
GHCi> zipWithM (\x y -> [x,y]) [1,2,3] [10,20,30]
[[1,2,3],[1,2,30],[1,20,3],[1,20,30],[10,2,3],[10,2,30],[10,20,3],[10,20,30]]
GHCi> zipWith (\x y -> [x,y]) [1,2,3] [10,20,30]
[[1,10],[2,20],[3,30]]
GHCi> zipWithM (\x y -> (show x++"++"++show y++";",x+y)) [1,2,3] [10,20,30]
("1+10;2+20;3+30;",[11,22,33])
```

```
-- old version
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
filterM _ [] = return []
filterM p (x:xs) = do
    flg <- p x
    ys <- filterM p xs
    return (if flg then x:ys else ys)

-- new version
filterM :: Applicative m => (a -> m Bool) -> [a] -> m [a]
filterM p [] = pure []
filterM p (x:xs) = (\b -> if b then (x:) else id)
                    <$> p x
                    <*> filterM p xs
```

```

GHCi> filterM (Just . odd) [1,2,3]
Just [1,3]
GHCi> traverse (Just . odd) [1,2,3]
Just [True,False,True]
> traverse (\x -> if odd x then Just x else Nothing) [1,2,3]
Nothing
GHCi> filterM (\_ -> [True,False]) [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
> filterM (\_ -> ZipList [True,False]) [1,2,3]
ZipList {getZipList = [[1,2,3],[]]}
GHCi> filterM (\x -> (show x,odd x)) [1,2,3]
("123",[1,3])

```

```

foldM :: (Foldable t, Monad m) =>
        (b -> a -> m b) -> b -> t a -> m b
foldM f ini xs = foldr (\x k z -> f z x >>= k) return xs ini

```

Пример использования `foldM`: суммирование с контролем выхода за диапазон (вычисления в монаде `Maybe`).

```

GHCi> isTiny x = x >= (-128) && x < 128
GHCi> (?+?) x y = if isTiny (x+y) then Just (x+y) else Nothing
GHCi> 3 ?+? 100
Just 103
GHCi> 90 ?+? 100
Nothing
GHCi> sumTiny = foldM (?+?) 0
GHCi> sumTiny [1..15]
Just 120
GHCi> sumTiny [1..16]
Nothing

```

### Пример: задача перечисления конфигураций

Рассмотрим задачу получения всех допустимых конфигурации для некоторой игры на некотором поле. Для определенности

```
type Board = Int
```

Для данной конфигурации на доске функция `next` возвращает список всех достижимых за один ход конфигураций

```
next :: Board -> [Board]
next ini = filter (>= 0) . filter (<= 9) $ [ini+2, ini-1]
```

Через `next` легко выразить позиции после двух и трех ходов

```
twoTurns :: Board -> [Board]
twoTurns ini = do
  bd1 <- next ini
  next bd1

threeTurns :: Board -> [Board]
threeTurns ini = do
  bd1 <- next ini
  bd2 <- next bd1
  next bd2
```

Как это обобщить?

► (Stepik) Напишите функцию, которая возвращает всевозможные конфигурации доски через `n` ходов.

```
doNTurns :: Int -> Board -> [Board]
doNTurns n ini = foldM undefined ini undefined
```

Конфигурация должна входить в результирующий список столько раз, сколько раз разными способами она может быть получена.

(Совет: используйте или `foldM` или обычную свертку над композицией рыбок.)

```
GHCi> doNTurns 1 5 == next 5
True
GHCi> doNTurns 2 5 == twoTurns 5
True
GHCi> doNTurns 3 5 == threeTurns 5
True
```

## Законы класса `Monad`

► Покажите, что из законов класса типов `Monad`

```
return a >>= k    ≡ k a
m >>= return      ≡ m
(m >>= v) >>= w    ≡ m >>= (\x -> v x >>= w)
```

следует, что стрелки Клейсли образуют моноид относительно операции их композиции `>=>` с `return` в качестве нейтрального элемента. Иными словами докажите, что верны равенства

```
return >=> k       ≡ k
k >=> return       ≡ k
(u >=> v) >=> w     ≡ u >=> (v >=> w)
```

Используйте найденное вами выше определение `>=>`, через `>>=` (или подглядите библиотечное).

► Покажите, что законы

```
join . return      ≡ id
join . fmap return  ≡ id
join . fmap join    ≡ join . join
```

следуют из законов класса типов `Monad`, используя полученные ранее реализации `join` через `>>=` и `fmap` (aka `liftM`) через `>>=` и `return`.

► Выразите `>=>`, `>>=` и `<*>` через `join` (и `fmap`).

```
(>=>..) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
g >=>.. h = undefined

(>>=..) :: Monad m => m a -> (a -> m b) -> m b
m >>=.. k = undefined

(<*>...) :: Monad m => m (a -> b) -> m a -> m b
fs <*>... xs = undefined
```

► Покажите, что законы класса типов `Monad`

```

return a >>= k    ≡ k a                -- (m1)
m >>= return     ≡ m                  -- (m2)
(m >>= v) >>= w   ≡ m >>= (\x -> v x >>= w) -- (m3)

```

следуют из законов

```

join . return     ≡ id                -- (mjfr1)
join . fmap return ≡ id                -- (mjfr2)
join . fmap join   ≡ join . join      -- (mjfr3)

```

используя полученную ранее реализацию `>>=` через `fmap` и `join`.

В преобразованиях могут пригодиться второй закон функторов

```
fmap (f . g) ≡ fmap f . fmap g
```

и «свободные теоремы» для типов `return` и `join`

```

return . f          ≡ fmap f . return
join . fmap (fmap f) ≡ fmap f . join

```

► Выразите `>>=`, `fmap` и `join` через `>=>` (и `return`).

```

infixl 1 >>==
(>>==) :: Monad m => m a -> (a -> m b) -> m b
m >>== k = undefined

fmap''' :: Monad m => (a -> b) -> m a -> m b
fmap''' f = undefined

join''' :: Monad m => m (m a) -> m a
join''' = undefined

```

► Покажите, что законы класса типов `Monad`

```

return a >>== k    ≡ k a                -- (m1')
m >>== return     ≡ m                  -- (m2')
(m >>== k1) >>== k2 ≡ m >>== (\x -> k1 x >>== k2) -- (m3')

```

следуют из «рыбьих» законов

```
return ==> k      ≡ k                -- (fLN)
k ==> return      ≡ k                -- (fRN)
(u ==> v) ==> w    ≡ u ==> (v ==> w)  -- (fAss)
```

используя полученную ранее реализацию `>>=` через `>=>`.

В преобразованиях понадобится следующая «свободная теорема» для типа `>=>`

```
k1 ==> k2 ≡ (id ==> k2) . k1
```

► Покажите, что типы всех трех способов записи композиции монадических эффектов изоморфны. Для этого предъявите функции-преобразования между ними и докажете тождественность обеих их композиций.

Например, для доказательства изоморфизма

```
forall a b c. (a -> m b) -> (b -> m c) -> a -> m c
≅
forall b c. m b -> (b -> m c) -> m c
```

надо задать две функции между этими типами

```
c2b :: (forall a b c. (a -> m b) -> (b -> m c) -> a -> m c) ->
      m b -> (b -> m c) -> m c
c2b c = undefined

b2c :: (forall b c. m b -> (b -> m c) -> m c) ->
      (a -> m b) -> (b -> m c) -> a -> m c
b2c b = undefined
```

и показать тождественность обеих композиций

```
isoCB :: (forall b c. m b -> (b -> m c) -> m c) ->
        m b -> (b -> m c) -> m c
isoCB b =
  (\m k -> c2b (b2c b) m k)
  == undefined
  == b
```

```

isoBC :: (forall a b c. (a -> m b) -> (b -> m c) -> a -> m c) ->
        (a -> m b) -> (b -> m c) -> a -> m c
isoBC c =
    (\k1 k2 a -> b2c (c2b c) k1 k2 a)
    == undefined
    == c

```

В преобразованиях понадобятся следующие «свободные теоремы» для типов `>>=` и `>=>` (2 представителя)

```

m >>= k    ≡    fmap k m >>= id

k1 >=> k2    ≡    (id >=> k2) . k1
k1 >=> k2    ≡    (id >=> id) . fmap k2 . k1

```

## Домашнее задание

- (1 балл) «Окружите» каждый элемент списка заданными «скобками», используя монаду списка и `do`-нотацию:

```
surround :: a -> a -> [a] -> [a]
surround x y zs = do undefined
```

Проверка:

```
GHCi> surround '{' '}' "abcd"
"{a}{b}{c}{d}"
```

- (1 балл) Ассоциативным списком называют список пар (ключ, значение). Реализуйте функцию поиска в таком списке, возвращающую список всех значений с заданным ключом. Используйте монаду списка и `do`-нотацию.

```
lookups :: Eq a => a -> [(a,b)] -> [b]
lookups x ys = do undefined
```

Проверка:

```
GHCi> lookups 2 [(1,"one"),(2,"two"),(3,"three"),(2,"two'")]
["two","two'"]
```

- (2 балла) Разложите положительное целое число на два сомножителя всевозможными способами, используя монаду списка и `do`-нотацию.

```
factor2 :: Integer -> [(Integer, Integer)]
factor2 n = do undefined
```

Пары должны быть уникальными, первый элемент пары не должен превышать второй, результат следует упорядочить лексикографически, в возрастающем порядке.

Проверка:

```
GHCi> factor2 45
[(1,45),(3,15),(5,9)]
```

► (2 балла) Вычислите модули разностей между соседними элементами списка, используя монаду списка и `do`-нотацию.

```
absDiff :: Num a => [a] -> [a]
absDiff xs = do undefined
```

Проверка:

```
GHCi> absDiff [2,7,22,9]
[5,15,13]
```

► (2 балла) Для типа данных

```
data OddC a = Un a | Bi a a (OddC a) deriving (Eq,Show)
```

(контейнер-последовательность, который по построению может содержать только нечетное число элементов) реализуйте функцию

```
concat3OC :: OddC a -> OddC a -> OddC a -> OddC a
```

конкатенирующую три таких контейнера в один:

```
GHCi> tst1 = Bi 'a' 'b' (Un 'c')
GHCi> tst2 = Bi 'd' 'e' (Bi 'f' 'g' (Un 'h'))
GHCi> tst3 = Bi 'i' 'j' (Un 'k')
GHCi> concat3OC tst1 tst2 tst3
Bi 'a' 'b' (Bi 'c' 'd' (Bi 'e' 'f' (Bi 'g' 'h' (Bi 'i' 'j' (Un 'k'))))))
```

Обратите внимание, что соображения четности запрещают конкатенацию двух контейнеров `OddC`.

► (2 балла) Для типа данных

```
data OddC a = Un a | Bi a a (OddC a) deriving (Eq,Show)
```

реализуйте функцию

```
concatOC :: OddC (OddC a) -> OddC a
```

Она должна обеспечивать для типа `OddC` поведение, аналогичное поведению функции `concat` для списков:

```
GHCi> concatOC $ Un (Un 42)
Un 42
GHCi> tst1 = Bi 'a' 'b' (Un 'c')
GHCi> tst2 = Bi 'd' 'e' (Bi 'f' 'g' (Un 'h'))
GHCi> tst3 = Bi 'i' 'j' (Un 'k')
GHCi> concatOC $ Bi tst1 tst2 (Un tst3)
Bi 'a' 'b' (Bi 'c' 'd' (Bi 'e' 'f' (Bi 'g' 'h' (Bi 'i' 'j' (Un 'k'))))))
```

► (2 балла) Сделайте тип данных

```
data OddC a = Un a | Bi a a (OddC a) deriving (Eq, Show)
```

представителем классов типов `Functor`, `Applicative` и `Monad`. Семантика должна быть подобной семантике представителей этих классов типов для списков: монада `OddC` должна иметь эффект вычисления с произвольным нечетным числом результатов:

```
GHCi> tst1 = Bi 10 20 (Un 30)
GHCi> tst2 = Bi 1 2 (Bi 3 4 (Un 5))
GHCi> do {x <- tst1; y <- tst2; return (x + y)}
Bi 11 12 (Bi 13 14 (Bi 15 21 (Bi 22 23 (Bi 24 25 (Bi 31 32 (Bi 33 34 (Un 35)))))))
GHCi> do {x <- tst2; y <- tst1; return (x + y)}
Bi 11 21 (Bi 31 12 (Bi 22 32 (Bi 13 23 (Bi 33 14 (Bi 24 34 (Bi 15 25 (Un 35)))))))
```