Курс: Функциональное программирование
Практика 9 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 -> ['a','b'] >>= \c -> return (n,c)
```

▶ Запишите приведенные выше примеры в `do`-нотации.

▶ 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 `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"
```

▶ Покажем, что каждая монада — это функтор. Для этого выразите `fmap` через `(>>=)` и `return`:

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

(Отметим, что для полноценного доказательства следует еще проверить выполнение законов класса `Functor`.)

▶ Запишите эту 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"
"abababab"
```

▶ Покажите, что `liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c` 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 `(<*>) :: Applicative f => f (a -> b) -> f a -> f b` на языке монад:

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

(Отметим, что для полноценного доказательства следует еще проверить выполнение законов класса `Applicative`.)

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

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.

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:

```
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]

zipWithM :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m [c]

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

foldM :: (Foldable t, Monad m) => (b -> a -> m b) -> b -> t a -> m b
```

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

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

Получение всех конфигурации для некоторой игры на некотором поле:

```
type Board = Int
```

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

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

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?
► Напишите функцию, которая возвращает всевозможные конфигурации доски через n ходов.

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

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

Законы класса `Monad`

В терминах композиций стрелок Клейсли законы класса `Monad` имеют особенно простой вид

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

«Монада в категории — это моноид в категории её эндофункторов, где умножение — композиция эндофункторов, а единица — тождественный эндофунктор.»

▶ Переведите законы класса `Monad` на язык `(>>=)`, используя формулу из предыдущего задания (`(>=>)` через `(>>=)`). В результате должны получится классические законы:

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

▶ Выразите `(>=>)` через `join` (и `fmap`).

▶ Переведите законы класса `Monad` на язык `join` и `fmap`, используя представление `(>=>)` через `join` (и `fmap`). В результате должны получится законы:

```
join . return           ==  id
join . fmap return      ==  id
join . join             ==  join . fmap join
```

Совет: в преобразованиях можно использовать 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
```

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]
```

► (4 балла суммарно) Задача: проанализировать всевозможные партии в крестики-нолики на произвольном квадратном поле $\mathfrak{n} \times \mathfrak{n}$. Начальная конфигурация: все клетки пусты. Крестики начинают, следующие ходы выполняются по очереди. Выигрыш — это $\mathfrak{n}$ крестиков или ноликов подряд по какой-то горизонтали, вертикали или большой диагонали (одной из двух). После выигрыша партия завершается, дальнейшие ходы не допускаются.

```
data Cell = E  -- empty, пустаяклетка
          | X  -- крестик
          | O  -- нолик
          deriving (Eq,Show)

type Row a = [a]
type Board = Row (Row Cell)
```

Начальная конфигурация для поля размера n:

```
iniBoard :: Int -> Board
iniBoard n = let row = replicate n E in replicate n row
```

▶ (1 балл) Напишите функцию, которая проверяет, является ли конфигурация на доске `brd` выигрышной для «игрока» `x`:

```
win :: Cell -> Board -> Bool
win E _   = False
win x brd = undefined
```

Проверка:

```
GHCi> win X [[X,O,X],[O,X,O],[X,O,X]]
True
```

▶ (2 балла) Напишите функцию, которая возвращает всевозможные конфигурации доски, которые могут быть получены за 1 ход «игрока» `x` из заданной конфигурации `brd`.

```
nxt :: Cell -> Board -> [Board]
nxt x brd = do undefined
```

Не забудьте, что выигрышные конфигурации завершают партию и не имеют продолжений:

```
GHCi> nxt O [[X,E],[O,X]]
[]
```

Проверка:

```
GHCi> nxt X (iniBoard 2)
[[[X,E],[E,E]],[[E,X],[E,E]],[[E,E],[X,E]],[[E,E],[E,X]]]
```

▶ (1 балл) Напишите функцию, которая возвращает всевозможные конфигурации доски через `n` последовательных ходов (крестик-нолик-крестик и т.д.) из заданной конфигурации `ini`.

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

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

Проверка:

```
GHCi> length $ doNTurns 3 (iniBoard 2)
24
```