Reader, Writer, State, List

Reader

Вычисление, допускающее чтение значений из разделяемого окружения.

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

reader :: (r -> a) -> Reader r a
runReader :: Reader r a -> r -> a
```

ask :: Reader r r возвращает окружение

asks :: (r -> a) -> Reader r a возвращает результат выполнения функции над окружением

local :: (r -> r) -> Reader r a -> Reader r a позволяет локально модифицировать окружение

Выражения (Reader)

Перепишите eval еще раз (вызывайте ошибку с помощью error, если это необходимо):

```
import Control.Monad.Trans.Reader

evalR :: Expr -> Reader (M.Map Name Integer) Integer
evalR = undefined
```

Например:

```
ghci> env = M.fromList [("x", 0)]
ghci> runReader (evalR (Let "x" (Num 7) (Var "x"))) env
7
ghci> runReader (evalR (Let "y" (Num 7) (Var "x"))) env
0
```

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)
execWriter :: Writer w a -> w

tell :: Monoid w => w -> Writer w () позволяет задать
      вывод
```

Упражнение (Writer)

Используя монаду Writer, напишите функцию правой свертки списка при помощи операции вычитания

```
minusLoggedR :: (Show a, Num a) => a -> [a] -> Writer String a
minusLoggedR = undefined
```

в которой рекурсивные вызовы сопровождались бы записью в лог, так чтобы в результате получалось такое поведение:

```
ghci> runWriter $ minusLoggedR 0 [1..3]
(2,"(1-(2-(3-0)))")
```

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)

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

Упражнение (State)

Реализуйте функцию f2 таким образом, чтобы sumNotTwice суммировала только первые вхождения значений в списке

```
sumNotTwice :: [Int] -> Int
sumNotTwice xs = fst $ runState (foldM f2 0 xs) []

f2 :: Int -> Int -> State [Int] Int
f2 acc x = todo
```

```
ghci>sumNotTwice [1,2,3]
6
ghci>sumNotTwice [1,1,2,3,2,2,3]
6
ghci>sumNotTwice [3,-2,3]
1
```

Упражнение (List)

Напишите реализацию функции filter, используя монаду списка и do-нотацию

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

Упражнение (List)

Напишите функцию, которая возвращает все пары чисел из xs, сумма которых равна k.

Реализуйте три варианта:

1. с помощью генераторов списков (list comprehension)
2. с помощью do-нотации
3. с помощью оператора (>>=)

```
findSum :: [Int] -> Int -> [(Int,Int)]
findSum xs k = undefined
```

Упражнение

Напишите реализацию функции:

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

```
ghci> filterM' (Just . odd) [1,2,3]
Just [1,3]
```

Объясните поведение:

```
ghci> filterM' (\_ -> [True,False]) [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```