Курс: Функциональное программирование
Практика 11. Стандартные монады

Разминка

▶ Устно вычислите значения выражений и проверьте результат в GHCi:

```
sequence [Just 1,Just 2,Just 3]

sequence [Just 1,Just 2,Nothing,Just 4]

sequence [[11,20],[21,40],[31,60]]

mapM (\x -> [x+1,x*2]) [10,20,30]

sequence_ [[11,20],[21,40],[31,60]]

mapM_ (\x -> [x+1,x*2]) [10,20,30]

forM_ [10,20,30] (\x -> [x+1,x*2])

do {x <- Just 5; guard $ x > 10}

do {x <- Just 5; guard $ x < 10}

msum [Just 1,Just 2,Just 3]

msum [Nothing,Nothing,Just 1,Just 2]

msum [[1,2,3],[10,20]]

mfilter (< 10) $ Just 12

mfilter (< 10) [1,3..]
```

▶ Устно вычислите значения выражений и определите их побочные эффекты. Проверьте результат в GHCi:

```
let x = print "first" in print "second"

let x = print "first" in x >> print "second"

(\x -> print "first") (print "second")

print "first" `seq` print "second"
```

Класс типов `MonadPlus`

▶ Какие из законов класса типов `MonadPlus` выполняются для списка? типа `Maybe`? Приведите доказательство или опровергающий пример.

Монада `Reader`

На лекции мы работали с монадой `Reader`, определенной так

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

instance Monad (Reader r) where
  return :: a -> Reader r a
  return x = reader $ \e -> x
  (>>=) :: Reader r a -> (a -> Reader r b) -> Reader r b
  m >>= k  = reader $ \e -> let v = runReader m e
                            in runReader (k v) e
```

▶ Сделайте этот `Reader` «работоспособным», реализовав

```
reader :: (r -> a) -> Reader r a
reader = undefined
instance Functor (Reader r) where
  fmap :: (a -> b) -> Reader r a -> Reader r b
  fmap = undefined
instance Applicative (Reader r) where
  pure :: a -> Reader r a
  pure = undefined
  (<*>) :: Reader r (a -> b) -> Reader r a -> Reader r b
  (<*>) = undefined
```

▶ Обеспечьте монаде `Reader` стандартный интерфейс:

```
ask :: Reader r r
ask = undefined
asks :: (r -> a) -> Reader r a
asks = undefined
local :: (r -> r) -> Reader r a -> Reader r a
local = undefined
```

▶ Интерфейс функции `local` можно расширить, допустив локальное изменение не только значения окружения, но и его типа. Напишите сигнатуру и реализацию соответствующей функции:

```
local' :: (r -> r') -> _
local' = undefined
```

Монада `Writer`

▶ (Stepik, 1 балл) Следующий тип данных

```
data Logged a = Logged String a deriving (Eq,Show)
```

удобно использовать для записи в строковой лог в процессе вычислений. Это упрощенная версия монады `Writer`, однако с инвертированным порядком записи в лог, см. пример ниже. Сделайте тип `Logged` представителем класса типов `Monad`. Реализуйте также функцию

```
write2log :: String -> Logged ()
write2log = undefined
```

позволяющую пользователю осуществлять запись в лог в процессе вычисления в монаде `Logged`.
Для проверки используйте следующий код

```
logIt v = do
  write2log $ "var = " ++ show v ++ "; "
  return v

test = do
  x <- logIt 3
  y <- logIt 5
  let res = x + y
  write2log $ "sum = " ++ show res ++ "; "
  return res
```

который при запуске должен дать

```
GHCi> test
Logged "sum = 8; var = 5; var = 3; " 8
```

Обратите внимание на обратный порядок записи в лог по сравнению с библиотечным `Writer`'ом.

Монада `State`

▶ Напишите функцию, вычисляющую факториал с использованием монады `State`.

```
fac n = fst $ execState (replicateM n facStep) (1,0)

facStep :: State (Integer,Integer) ()
facStep = do undefined
```

▶ Можно задать внешнее управление «переменной цикла»:

```
fac' n = execState (forM_ [1..n] facStep') 1

facStep' :: Integer -> State Integer ()
facStep' i = do undefined
```

Напишите «тело цикла» для этой реализации.

Тип `IORef`

Тип `IORef` позволяет определять ссылки на изменяемые переменные (ячейки памяти) внутри монады `IO`. Интерфейс работы с ними таков:

```
-- создание
newIORef :: a -> IO (IORef a)
-- чтение
readIORef :: IORef a -> IO a
-- запись
writeIORef :: IORef a -> a -> IO ()
-- изменение
modifyIORef  :: IORef a -> (a -> a) -> IO ()
-- строгая версия
modifyIORef' :: IORef a -> (a -> a) -> IO ()
```

Пример использования

```
testIORef = do
  ref  <- newIORef 1
  val1 <- readIORef ref
  writeIORef ref 41
  val2 <- readIORef ref
  modifyIORef' ref succ
  val3 <- readIORef ref
  return [val1,val2,val3]
```

```
GHCi> testIORef
[1,41,42]
```

▶ Напишите функцию в «императивном» стиле, вычисляющую факториал с использованием `IORef`.

```
fac'' :: Integer -> IO Integer
fac'' n = do undefined
```

Случайные числа (System.Random)

Два способа получить генератор псевдо-случайных чисел:
1. Использовать глобальный, инициализированный системным временем (при каждом запуске программы — новая уникальная псевдо-случайная последовательность):

```
GHCi> :t getStdGen
getStdGen :: IO StdGen
GHCi> getStdGen
1033221633 1
```

2. Если есть требование воспроизводимости — создать свой:

```
GHCi> :t mkStdGen
mkStdGen :: Int -> StdGen
GHCi> myGen = mkStdGen 42
GHCi> myGen
43 1
```

Для получения случайных чисел используют соответственно

```
randomIO :: IO a

random :: RandomGen g => g -> (a, g)

randoms :: RandomGen g => g -> [a]
```

```
GHCi> randomIO :: IO Int
307975744274596427
GHCi> randomIO :: IO Double
0.3315014473825033
GHCi> randomIO :: IO Double
0.759124955234098

GHCi> (replicateM 3 randomIO) :: IO [Int]
[7649266450277920169,-2426475120119592122,-7374390656014310684]

GHCi> take 3 $ randoms myGen :: [Int]
[-3900021226967401631,6115732954341747105,-7802898033696815382]
```

Часто удобны версии с ограниченным диапазоном

```
randomRIO :: (a, a) -> IO a

randomR  :: RandomGen g => (a, a) -> g -> (a, g)
randomRs :: RandomGen g => (a, a) -> g -> [a]
```

```
GHCi> (replicateM 5 $ randomRIO (1,6)) :: IO [Int]
[1,2,4,6,2]
GHCi> (replicateM 5 $ randomRIO (1,6)) :: IO [Int]
[4,3,1,5,2]
GHCi> take 5 $ randomRs (1,6) myGen :: [Int]
[6,4,2,5,3]
GHCi> take 5 $ randomRs (1,6) myGen :: [Int]
[6,4,2,5,3]
```

► Иногда неудобно пользоваться одним бесконечным списком, возвращаемым `randoms` или `randomRs`. В этих случаях приходится явно передавать генератор между вычислениями. Реализуйте функцию, прячущую эту передачу в монаде `State`:

```
randomRState :: (Random a, RandomGen g) => (a, a) -> State g a
randomRState (x,y) = do undefined
```

Используйте для проверки

```
testWork :: ([Int],[Int])
testWork = evalState doWork (mkStdGen 42)

doWork :: State StdGen ([Int],[Int])
doWork = do
  xs <- replicateM 5 $ randomRState (1,6)
  ys <- replicateM 5 $ randomRState (1,6)
  return (xs, ys)
```

В результате (с высокой степенью вероятности) должны получиться **разные** списки:

```
GHCi> testWork
([6,4,2,5,3],[2,1,6,1,4])
```

Файловый ввод-вывод

Типы для работы с файлами (экспортируются из `System.IO`):

```
data IOMode  =  ReadMode | WriteMode | AppendMode | ReadWriteMode
                deriving (Eq, Ord, Ix, Enum, Read, Show)

type FilePath =  String

data Handle  =  ...
```

Основные функции для работы с файлами:

```
openFile :: FilePath -> IOMode -> IO Handle

hPutChar :: Handle -> Char -> IO ()
hPutStr :: Handle -> String -> IO ()
hPutStrLn :: Handle -> String -> IO ()
hPrint :: Show a => Handle -> a -> IO ()

hGetContents :: Handle -> IO String

hClose :: Handle -> IO ()

withFile :: FilePath -> IOMode -> (Handle -> IO r) -> IO r
```

Пример файлового ввода-вывода:

```
main = do
  let txt = "Some text"
  handle <- openFile "Text.txt" WriteMode
  hPutStrLn handle txt
  hClose handle

  putStrLn "Hit any key to continue..."
  ignore <- getChar

  withFile "Text.txt" ReadMode $
    \h -> hGetContents h
    >>= putStrLn

  putStrLn "Hit any key to continue..."
  ignore <- getChar
  return ()
```

Домашнее задание

▶ (1 балл) Используя монаду `Writer`, напишите функцию правой свертки вычитанием

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

▶ (2 балла) Используя монаду `Writer`, напишите функцию **левой** свертки вычитанием

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

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

```
GHCi> runWriter $ minusLoggedL 0 [1..3]
(-6,"(((0-1)-2)-3)")
```

▶ (1 балл) Напишите функцию, вычисляющую числа Фибоначчи с использованием монады `State`.

```
fib :: Int -> Integer
fib n = fst $ execState (replicateM n fibStep) (0,1)

fibStep :: State (Integer,Integer) ()
fibStep = do undefined
```

▶ (2 балла) Напишите функцию

```
while :: IORef a -> (a -> Bool) -> IO () -> IO ()
while ref p action = do undefined
```

позволяющую кодировать «императивные циклы» следующего вида:

```haskell
factorial :: Integer -> IO Integer
factorial n = do
  r <- newIORef 1
  i <- newIORef 1
  while i (<= n) ( do
    ival <- readIORef i
    modifyIORef' r (* ival)
    modifyIORef' i (+ 1)
   )
  readIORef r
```

► (4 балла суммарно) Задача — эмуляция случайного эксперимента с использованием генератора случайных чисел. Эксперимент: наблюдаются `k` серий подбрасываний монетки (каждая серия состоит из `n` подбрасываний).

► (2 балла) Вычислите усреднённый по сериям модуль отклонения количества орлов от своего теоретического среднего значения в серии (предполагается, что эта величина известна и равна половине длины серии `n`). Используйте глобальный системный генератор случайных чисел.

```haskell
avgdev :: Int -> Int -> IO Double
avgdev k n = undefined
```

► (1 балл) Вычислите усреднённый по сериям модуль отклонения количества орлов от своего теоретического среднего значения в серии (предполагается, что эта величина известна и равна половине длины серии `n`). Генератор случайных чисел подразумевается созданным с помощью `mkStdGen` и передается в процессе вычислений через монаду `State` (используйте реализованную на практике функцию `randomRState`).

```haskell
randomRState :: (Random a, RandomGen g) => (a, a) -> State g a
randomRState (x,y) = do undefined

avgdev' :: Int -> Int -> State StdGen Double
avgdev' k n = undefined
```

► (1 балл) Вычислите усреднённый по сериям модуль отклонения количества орлов от своего теоретического среднего значения в серии (предполагается, что эта величина известна и равна половине длины серии `n`). Генератор случайных чисел создайте с помощью `mkStdGen`. Решите задачу, не используя монад: явно передавая генератор между вычислениями или используя как источник случайности бесконечный список, возвращаемый `randomRs`.

```haskell
avgdev'' :: Int -> Int -> Double
avgdev'' k n = undefined
```

▶ (0 баллов) Для $\mathtt{n} = 1000$ и $\mathtt{k} = 1000$ запишите в файл в виде гистограммы (ascii-art):

```
...
 -5 xxxxxxxxxxxxxxx
 -4 xxxxxxxxxxxxxxxxxx
 -3 xxxxxxxxxxxxxxxxxxxxx
 -2 xxxxxxxxxxxxxxxxxxxxx
 -1 xxxxxxxxxxxxxxxxxxxxxxxxx
  0 xxxxxxxxxxxxxxxx
  1 xxxxxxxxxxxx
  2 xxxxxxxxx
...
```

абсолютные частоты округленных до целых отклонений количества орлов от среднего значения. Постарайтесь не включать пустые «хвосты» распределения и не допускать неровностей основания гистограммы из-за изменения знака и/или разрядности чисел.

Домашнее задание 2

▶ (1 балл) Введём тип данных для представления ошибки обращения к списку по недопустимому индексу.

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

Реализуйте оператор `(!!!)` доступа к элементам массива по индексу, отличающийся от стандартного `(!!)` поведением в исключительных ситуациях. В этих ситуациях он должен выбрасывать подходящее исключение типа `ListIndexError`.

```
infixl 9 !!!
(!!!) :: (MonadError ListIndexError m) => [a] -> Int -> m a
xs !!! n = undefined
```

Ожидаемое поведение:

```
GHCi> let Right x = [1,2,3] !!! 0 in x
1
GHCi> let Left e = [1,2,3] !!! 42 in e
ErrTooLargeIndex 42
GHCi> let Left e = [1,2,3] !!! (-10) in e
ErrNegativeIndex
```

▶ (2 балла) Реализуйте собственную монаду обработки ошибок (взамен `Either e`) со строковым типом информации об ошибке на основе типа данных

```
data Excep a =  Err String | Ok a
  deriving (Eq, Show)
```

Сделайте этот тип представителем классов типов `Monad`, `MonadFail`, `Alternative`, `MonadPlus` и `MonadError String` (в последнем случае потребуются прагмы `FlexibleInstances`, `FlexibleContexts` и `MultiParamTypeClasses`). Протестируйте работу на примере оператора деления:

```
(?/) :: (MonadError String m)
            => Double -> Double -> m Double
x ?/ 0 = throwError "Division by 0."
x ?/ y = return $ x / y
```

Представители классов типов `Monad` и `MonadPlus` должны обеспечивать следующее поведение: при вызовах функции

```
example :: Double -> Double -> Excep String
example x y = action  `catchError` return where
  action = do
    q <- x ?/ y
    guard (q >= 0)
    if q  > 100 then do
      100 <- return q
      undefined
    else
      return $ show q
```

должны возвращаться такие результаты:

```
GHCi> example 5 2
Ok "2.5"
GHCi> example 5 0
Ok "Division by 0."
GHCi> example 5 (-2)
Ok "MonadPlus.empty error."
GHCi> example 5 0.002
Ok "Monad.fail error."
```

▶ (2 балла) Введём тип данных для представления ошибки синтаксического разбора и зададим синоним типа для монады-обработчицы ошибок

```
data ParseError = ParseError {location::Int, reason::String}

type ParseMonad = Either ParseError
```

Разработайте следующие функции

```
parseHex :: String -> ParseMonad Integer
parseHex = undefined

printError :: ParseError -> ParseMonad String
printError = undefined
```

Функция `parseHex` пытается разобрать переданную ей строку как шестнадцатеричное число. При удачном исходе она возвращает это число, а при неудачном — генерирует исключение. Функция `printError` выводит информацию об этом исключении в удобном текстовом виде. Для тестирования используйте

```
test s = str where
  (Right str) = do
      n <- parseHex s
      return $ show n
    `catchError` printError
```

Ожидаемое поведение:

```
GHCi> test "DEADBEEF"
"3735928559"
GHCi> test "DEADMEAT"
"At pos 5: M: invalid digit"
```

Совет: воспользуйтесь вспомогательными функциями из `Data.Char`.

▶ (2 балла) Разберитесь в работе следующего кода

```
import Control.Monad.Trans.Maybe
import Data.Char (isNumber, isPunctuation)

askPassword :: MaybeT IO ()
askPassword = do
  liftIO $ putStrLn "Enter your new password:"
  value <- msum $ repeat getValidPassword
  liftIO $ putStrLn "Storing in database..."

getValidPassword :: MaybeT IO String
getValidPassword = do
  s <- liftIO getLine
  guard (isValid s)
  return s

isValid :: String -> Bool
isValid s = length s >= 8
            && any isNumber s
            && any isPunctuation s
```

вызывая его в интерпретаторе:

```
GHCi> runMaybeT askPassword
```

Используя пользовательский тип ошибки и трансформер `ExceptT`, модифицируйте приведенный выше код так, чтобы он выдавал пользователю сообщение о причине, по которой пароль отвергнут.

```haskell
data PwdError = PwdError String

type PwdErrorMonad = ExceptT PwdError IO

askPassword' :: PwdErrorMonad ()
askPassword' = do
  liftIO $ putStrLn "Enter your new password:"
  value <- msum $ repeat getValidPassword'
  liftIO $ putStrLn "Storing in database..."

getValidPassword' :: PwdErrorMonad String
getValidPassword' = undefined
```

Ожидаемое поведение:

```
GHCi> runErrorT askPassword'
Enter your new password:
qwerty
Incorrect input: password is too short!
qwertyuiop
Incorrect input: password must contain some digits!
qwertyuiop123
Incorrect input: password must contain some punctuations!
qwertyuiop123!!!
Storing in database...
GHCi>
```