Курс: Функциональное программирование
Практика 9. Использование монад

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

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

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

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

Монада State

Вспомним пример с лекции

```
tick :: State Int Int
tick = do  n <- get
           put (n+1)
           return n

succ' :: Int -> Int
succ' n = execState tick n

plus :: Int -> Int -> Int
plus n x = execState (sequence $ replicate n tick) x
```

Функция

```
sequence  :: Monad m => [m a] -> m [a]
sequence ms = foldr k (return []) ms
            where
              k m m' = do { x <- m; xs <- m'; return (x:xs) }
```

выполняет каждое действие списка слева направо и собирает результаты в список.

► Проследите за работой `sequence`, заменив в коде `plus` вызов `execState` на `runState`.

► Напишите функцию вычисляющую числа Фибоначчи с использованием монады State.

Монада `Writer`

► Используя монаду `Writer`, напишите версию библиотечной функции `sum` — функцию `sumLogged :: Num a => [a] -> Writer String a`, в которой бы рекурсивные вызовы сопровождались бы записью в лог, так чтобы в результате получалось такое поведение:

```
> runWriter $ sumLogged [1..10]
(55,"(1+(2+(3+(4+(5+(6+(7+(8+(9+(10+0)))))))))")
```

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)

```
> :t getStdGen
getStdGen :: IO StdGen
> getStdGen
701460132 1
```

2. если есть требование воспроизводимости — создать свой

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

Для получения случайных чисел используют соответственно

```
randomIO :: IO a

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

> randomIO :: IO Int
-1347547884
> randomIO :: IO Double
0.14185627922415733
> randomIO :: IO Double
0.26660025701858103

> (sequence $ replicate 5 randomIO) :: IO [Int]
[76719735,-514201760,-1452869230,1224644498,-853026828]
> take 5 $ randoms myGen :: [Int]
[-1673289139,1483475230,-825569446,1208552612,104188140]
```

Часто удобны версии с ограниченным диапазоном

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

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

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

▶ Напишите программу эмулирующую 1000 серий подбрасываний монетки по 1000 раз. Вычислите усреднённый по сериям модуль отклонения.

Файловый IO

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 ()
```

▶ Результат отклонений подбрасований монетки от среднего (из предыдущего задания) запишите в файл в виде гистограммы (ascii-art):

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