CS212. Парадигмы и технологии программирования: часть 1, функциональное программирование

Лекция 4. Вычисления с побочными эффектами: ввод-вывод и случайные числа

В. Н. Брагилевский

28 февраля 2018 г.

Направление «Фундаментальная информатика и информационные технологии»
Институт математики, механики и компьютерных наук имени И. И. Воровича
Южный федеральный университет

Программа «Привет, мир»

Программа: функция main, компиляция, запуск

Файл helloworld.hs

```
main = putStrLn "Привет, мир"
```

```
$ ghc helloworld
[1 of 1] Compiling Main
Linking helloworld ... ( helloworld.hs, helloworld.o )
$ ./helloworld
Привет, мир
```

Как это вообще возможно?

- В языке Haskell «чистые» функции (функции без побочных эффектов, их результат однозначно определяется параметрами).
- Чтение с клавиатуры, вывод значения, генерация случайного числа — «грязные» действия.
- Имеется специальный механизм, позволяющий отделить «чистую» часть программы от той, в которой выполняется ввод-вывод и другие действия с побочными эффектами — механизм монадических вычислений.
- Ограничение побочных эффектов — это один из примеров применения монадических вычислений, есть и другие.

Смотрим на типы

```
ghci> :t putStrLn
putStrLn :: String -> IO ()
```

Основные понятия

- IO (конструктор типа) — монада, инкапсулирующая часть программы, ответственную за ввод-вывод.
- Значение типа IO <тип> называется действием ввода-вывода.
- Есть ли функция типа IO a -> a?
- Внутри монады IO можно вызывать чистые функции, а в чистых функциях пользоваться монадой IO нельзя.

Простейший ввод-вывод

Простейший ввод-вывод

Чтение пользовательского ввода

Чтение пользовательского ввода

```
main = do
    putStrLn "Привет, как тебя зовут?"
    name <- getLine
    putStrLn hello
```

```
ghci> :t getLine
getLine :: IO String
```

- Объединение действий ввода-вывода с помощью **do**.
- Извлечение результатов действий с помощью **<-**.
- Вызов чистых функций (например, ++).
- Действие выполняется, только если оно вызвано в функции `main`.

do-блоки и функциональный стиль

```haskell
main = do
    putStrLn "Привет, как тебя зовут?"
    name <- getLine
    putStrLn $ "Привет, " ++ name

main = putStrLn "Привет, как тебя зовут?"
       >> getLine >>= putStrLn . ("Привет, " ++)
```

- Полная эквивалентность.
- Операции >> и >>=.

Пример ошибочной программы

```
nameTag :: String
nameTag = "Привет, меня зовут " ++ getLine
```

- Попытка выхода из монады **IO**.
- Несоответствие типов.
- Результат чтения с клавиатуры должен быть извлечён с помощью **<-** (хотя он и остаётся внутри монады **IO**).

Ключевое слово let в блоке do

```haskell
import Data.Char

main = do
    putStrLn "Ваше имя?"
    firstName <- getLine
    putStrLn "Ваша фамилия?"
    lastName <- getLine
    let bigFirstName = map toUpper firstName
        bigLastName  = map toUpper lastName
    putStrLn $ "Привет, " ++ bigFirstName ++ " "
                 ++ bigLastName ++ ", как дела?"
```

Собственные действия и функция return

```haskell
getNumber :: IO Integer
getNumber = do
    line <- getLine
    return $ read line

main = do
   a <- getNumber
   b <- getNumber
   print $ a + b
```

Типы функций main и return

```
main = do
    return ()
    return "!!!!!"
    line <- getLine
    return "?????"
    return 4
    putStrLn line
```

```
ghci> :t return
return :: Monad m => a -> m a
ghci> :t main
main :: IO ()
```

- В современном коде вместо return может встретиться функция pure, подробности позднее.

Простейший ввод-вывод

Пример: обращение строк

Пример: обращение строк

Постановка задачи

Пользователь вводит строки одна за другой. Строка разбивается на слова, и порядок символов в каждом слове меняется на противоположный. Слова вновь соединяются в строку, которая выводится на консоль. Программа завершает работу при вводе пустой строки.

Компоненты решения

- Преобразование строки — чистая функция.
- Функция `main`: ввод строки, вызов функции преобразования строки и вывод результата либо завершение работы.
- Как обеспечить повторное выполнение действия?

Решение

```haskell
reverseWords :: String -> String
reverseWords = unwords . map reverse . words

main = do
    line <- getLine
    if null line
      then return ()
      else do
             putStrLn $ reverseWords line
             main
```

- Рекурсия для повторения действия.
- Блок **do** как «составной» оператор.

Простейший ввод-вывод

Некоторые функции ввода-вывода

Функции печати

- putStrLn
- putStr
- putChar
- print

```haskell
main = do
    print True
    print 2
    print "xa - xa"
    print 3.2
    print [3,4,3]

print :: Show a => a -> IO ()
```

Функция when

```haskell
import Control.Monad

main = do
  input <- getLine
  when (input /= "EXIT") (putStrLn input)
```

```haskell
when :: Monad m => Bool -> m () -> m ()
```

Функция when: пример с обращением строк

```haskell
import Control.Monad

reverseWords :: String -> String
reverseWords = unwords . map reverse . words

main = do
    line <- getLine
    when (not $ null line) $ do
                putStrLn $ reverseWords line
                main
```

Повторение действий: функция sequence

```
sequence :: (Monad m, Traversable t) =>
            t (m a) -> m (t a)

main = do
   rs <- sequence [getLine, getLine, getLine]
   print rs

ghci> :main
123
456
78
["123","456","78"]
```

Печать элементов списка

```
ghci> map print [1..3]

<interactive>:2:1:
 No instance for (Show (IO ()))
   arising from a use of print'
 Possible fix: add an instance declaration for (Show (IO ()))
 In a stmt of an interactive GHCi command: print it

ghci> sequence $ map print [1..3]
1
2
3
[(),(),()]
```

Функции mapM и mapM_

```
ghci> mapM print [1,2,3]
1
2
3
[(),(),()]
ghci> mapM_ print [1,2,3]
1
2
3
```

```
mapM  :: (Monad m, Traversable t) =>
         (a -> m b) -> t a -> m (t b)
mapM_ :: (Monad m, Foldable t) =>
         (a -> m b) -> t a -> m ()
```

Функция forever

```haskell
import Control.Monad
import Data.Char

main = forever $ do
    putStr "Введите строку: "
    l <- getLine
    putStrLn $ map toUpper l
```

Полезные соображения

- Монады позволяют структурировать вычисления.
- Монада `IO` инкапсулирует вычисления с побочными эффектами.
- Действия ввода-вывода — значения типа `IO <тип>`, они выполняются при вызове в `main`.

Обработка файлов

Чтение входного потока целиком

Файл up.hs

```
import Data.Char

main = do
    contents <- getContents
    putStr $ map toUpper contents
```

Файл haiku.txt

```
Я маленький чайник
Ох уж этот обед в самолёте
Он так мал и безвкусен
```

Использование перенаправления ввода-вывода

```
$ ./up < haiku.txt
Я МАЛЕНЬКИЙ ЧАЙНИК
ОХ УЖ ЭТОТ ОБЕД В САМОЛЁТЕ
ОН ТАК МАЛ И БЕЗВКУСЕН
```

Преобразование входного потока

```haskell
main = interact shortLinesOnly

shortLinesOnly :: String -> String
shortLinesOnly = unlines
                 . filter (\l -> length l < 15)
                 . lines
```

Функции readFile, writeFile и appendFile

```haskell
type FilePath = String

readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
```

Пример

```haskell
import System.IO
import Data.Char

main = do
    contents <- readFile "file1.txt"
    writeFile "file2.txt" (map toUpper contents)
```

- Содержимое файла целиком в память не записывается!

Дополнительные функции для работы с файлами

```
openTempFile :: FilePath -> String
                -> IO (FilePath, Handle)

import System.Directory
removeFile :: FilePath -> IO ()
renameFile :: FilePath -> FilePath -> IO ()
```

Чтение аргументов командной строки

Аргументы командной строки

```haskell
import System.Environment

main = do
    args <- getArgs
    putStrLn "Аргументы командной строки:"
    mapM_ putStrLn args



getArgs :: IO [String]
```

Запуск из командной строки

```
$ ./arg-test first second "multi word arg"
Аргументы командной строки:
first
second
multi word arg
```

Запуск из интерпретатора GHCi

```
ghci> :main first second "multi word arg"
Аргументы командной строки:
first
second
multi word arg
```

Пример: чтение и обработка данных Твиттера

Постановка задачи

Дан текстовый файл со следующей информацией: имя аккаунта, число читателей, число читаемых, число твитов (может отсутствовать):

```
KimKardashian  36066069  122  20152
blakeshelton  13926346  596  17753
selenagomez  33957639  1283
10Ronaldinho  12737249  13  2500
```

Разработать соответствующий тип данных, организовать загрузку данных из файла, имя которого задано аргументом командной строки, и вывести пользователя с наибольшим числом читателей.

Компоненты решения

```haskell
data TwitterAccount

str2ta :: String -> TwitterAccount
loadData :: FilePath -> IO [TwitterAccount]
report :: [TwitterAccount] -> TwitterAccount
```

Решение

Тип данных

```
data TwitterAccount =
       TA String Integer Integer (Maybe Integer)
   deriving Show
```

Преобразование строки

```haskell
str2ta :: String -> TwitterAccount
str2ta = buildTA . words
 where
  buildTA (name:flwrs:flwngs:xs) =
                        TA name (read flwrs)
                                (read flwngs)
                                (ntweets xs)
  buildTA _ = error "incorrect data format"

  ntweets [] = Nothing
  ntweets [twts] = Just (read twts)
  ntweets _ = error "incorrect data format"
```

Загрузка данных из файла

```
loadData :: FilePath -> IO [TwitterAccount]
loadData fname = do
    fc <- readFile fname
    return $ map str2ta $ lines fc
```

Обработка данных

```
report :: [TwitterAccount] -> TwitterAccount
report = maximumBy
          (comparing $ \(TA _ flwrs _ _) -> flwrs)
```

Основная программа

```
main = do
  [fname] <- getArgs
  acts <- loadData fname
  print $ report acts
```

Случайные числа

Случайные числа

- Псевдослучайные числа: начальное значение, алгоритм для вычисления следующего числа.
- Генератор псевдослучайных чисел.
- Случайные числа и чистые функции.

Модуль System.Random

```
random :: (RandomGen g, Random a) => g -> (a, g)
mkStdGen :: Int -> StdGen
```

```
ghci> random (mkStdGen 100) :: (Int, StdGen)
(-1352021624,651872571 1655838864)
ghci> random (mkStdGen 100) :: (Int, StdGen)
(-1352021624,651872571 1655838864)
ghci> random (mkStdGen 949488) :: (Float, StdGen)
(0.8938442,1597344447 1655838864)
ghci> random (mkStdGen 949488) :: (Bool, StdGen)
(False,1485632275 40692)
ghci> random (mkStdGen 949488) :: (Integer, StdGen)
(1691547873,1597344447 1655838864)
```

Подбрасывание монет

```haskell
threeCoins :: StdGen -> (Bool, Bool, Bool)
threeCoins gen =
    let (firstCoin, newGen) = random gen
        (secondCoin, newGen') = random newGen
        (thirdCoin, newGen'') = random newGen'
    in (firstCoin, secondCoin, thirdCoin)
```

```
ghci> threeCoins (mkStdGen 21)
(True,True,True)
ghci> threeCoins (mkStdGen 22)
(True,False,True)
ghci> threeCoins (mkStdGen 943)
(True,False,True)
```

Бесконечный список случайных чисел

```
randoms :: (RandomGen g, Random a) => g -> [a]

ghci> take 5 $ randoms (mkStdGen 11) :: [Integer]
[52586989052654 6791
,-10461301124107 7602
,36034014877393 0952,
-5956255232421 4439
,-24208876969841391]
ghci> take 5 $ randoms (mkStdGen 11) :: [Bool]
[True,True,True,True,False]
```

Случайные числа в заданном диапазоне

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

```
ghci> randomR (1,6) (mkStdGen 359353)
(6,1494289578 40692)
ghci> randomR (1,6) (mkStdGen 35935335)
(3,1250031057 40692)
ghci> rs = randomRs ('a','z') (mkStdGen 3) :: [Char]
ghci> take 10 rs
"ndkxbvmomg"
```

ГДЕ БРАТЬ ГЕНЕРАТОР СЛУЧАЙНЫХ ЧИСЕЛ?

Получение генератора случайных чисел

```
newStdGen :: IO StdGen
```

Пример программы

```
import System.Random

main = do
    gen <- newStdGen
    putStrLn $ take 10 $ randomRs ('a','z') gen
```

```
ghci> :main
yzgwojwjzf
ghci> :main
wvkywiwuot
ghci> :main
xvatecovpu
```

Пример: игра-угадайка

```
import System.Random
import Control.Monad (when)

main = do
    gen <- newStdGen
    askForNumber gen
```

```haskell
askForNumber :: StdGen -> IO ()
askForNumber gen = do
  let (randNumber, newGen) =
            randomR (1,10) gen :: (Int, StdGen)
  putStr "Я задумал число от 1 до 10. Какое? "
  numberString <- getLine
  when (not $ null numberString) $ do
     let number = read numberString
     putStrLn $ if randNumber == number
                then "Правильно!"
                else "Извините, но правильный ответ "
                     ++ show randNumber
     askForNumber newGen
```

УЖАСНАЯ МЕШАНИНА!

Взаимодействие с пользователем

```
game quest correct winMsg failMsg cont = do
  putStr quest
  answer <- getLine
  when (not $ null answer) $ do
    putStrLn $ if correct answer
               then winMsg
               else failMsg
    cont
```

Игровая логика

```
askForNumber gen =
  game "Я задумал число от 1 до 10. Какое? "
       correct
       "Правильно!"
       ("Извините, но правильный ответ "
         ++ show randNumber)
       (askForNumber newGen)
  where
    (randNumber, newGen) =
         randomR (1,10) gen :: (Int, StdGen)
    correct s = randNumber == read s
```