CS314. Функциональное программирование

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

В. Н. Брагилевский

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

22 сентября 2017 г.

Содержание

Содержание

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.

Содержание

Содержание

Чтение пользовательского ввода

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

```
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

```
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

```
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

```
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.

Содержание

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

```
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.

Содержание

Функции печати

- putStrLn
- putStr
- putChar
- print

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

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

Функция when

```
import Control.Monad

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

```
ghci> :t when
when :: Monad m => Bool -> m () -> m ()
```

Функция when: пример с обращением строк

```
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

```
ghci> :t sequence
sequence :: Monad m => [m a] -> m [a]
```

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

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

```
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.

Содержание

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

```
main = interact shortLinesOnly

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

Чтение файла

```
import System.IO

main = do
    handle <- openFile "file.txt" ReadMode
    contents <- hGetContents handle
    putStr contents
    hClose handle
```

Чтение файла: пример с ошибкой

```
import System.IO

main = do
    handle <- openFile "file.txt" ReadMode
    contents <- hGetContents handle
    hClose handle
    putStr contents
```

- Ленивость.
- Чтение из закрытого дескриптора.
- Жестокий грязный мир IO: явная последовательность действий, дескрипторы как сущности операционной системы и пр.
- Вывод: следует пользоваться высокоуровневыми обёртками, инкапсулирующими работу с файловыми дескрипторами.

Функция withFile

```
import System.IO

main =  withFile "file.txt" ReadMode (\handle -> do
            contents <- hGetContents handle
            putStr contents)
```

```
import System.IO

processFile handle = do
    contents <- hGetContents handle
    putStr contents

main =  withFile "file.txt" ReadMode processFile
```

Функции readFile, writeFile и appendFile

```
type FilePath = String

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

Пример

```
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!
- Этот способ наиболее предпочтителен!

Дополнительные функции для работы с файлами

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

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

Содержание

Аргументы командной строки

```
import System.Environment
import Data.List

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

```
ghci> :t getArgs
getArgs :: IO [String]
```

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

Содержание

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.

Требуемые типы и функции

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

Решение

АТД

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

Преобразование строки

```
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"
```

Решение

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
```

Содержание

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

```
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]
[5258698905265467 91,-10461301124107760 2,3603401487393095 2,
-5956255232421443 9,-2420887696984139 1]
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)
```

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

Получение генератора случайных чисел

```
ghci> :t getStdGen
getStdGen :: IO StdGen
ghci> :t newStdGen
newStdGen :: IO StdGen
```

```
import System.Random

main = do
   gen <- getStdGen
   putStrLn $ take 20 $ randomRs ('a','z') gen
```

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

Содержание