Get Programming with Haskell

*Командная строка и ленивый ввод-вывод

Для получения аргументов командной строки вы можете использовать функцию getArgs, которую можно найти в System.Environment. Её типовая аннотация выглядит следующим образом:

getArgs :: IO [String]

То есть вы получаете список строк в контексте IO.

*Пример

```
import System.Environment
main :: IO ()
main = do
  args <- getArgs
  …
```

*Доступ к аргументам командной строки (getArgs)

Чтобы понять, как работает getArgs, можно вывести полученные нами аргументы args. Как вам известно, args — список, а потому вы можете использовать map для прохода по всем значениям. Но есть проблема: вы работаете в контексте do-нотации с типом IO. Вам хотелось бы написать следующий код:

main putStrLn args

Но args — не просто список, а putStrLn — не просто функция. Выполнить map над списком в контексте IO можно с помощью специальной версии функции map, работающей со списками в этом контексте (вообще говоря, со списком в любом контексте, являющемся представителем класса типов Monad). Для этого существует специальная функция под названием mapM (M означает Monad).

*Попытка исправления: функция mapM (не работает!)

```
main :: IO ()
main = do
  args <- getArgs
  mapM putStrLn args
```

При попытке скомпилировать программу вы снова получите ошибку:
Couldn’t match type ’[()]’ with ’()’

*Попытка исправления: функция mapM (не работает!)

```
main :: IO ()
main = do
  args <- getArgs
  mapM putStrLn args
```

При попытке скомпилировать программу вы снова получите ошибку:
Couldn’t match type ’[()]’ with ’()’

GHC выдаёт ошибку из-за того, что тип main должен быть IO (), но map, как вы помните, возвращает список. Вам нужно просто пройти по элементам списка args и выполнить действия ввода-вывода; результаты этих действий не имеют значения, и вам не нужен список возвращаемых значений.

*Попытка исправления: функция mapM_ (работает!)

Справиться с этой проблемой поможет функция под названием mapM_ (обратите внимание на нижнее подчёркивание).
Она работает как mapM, но отбрасывает результаты. Обычно, если имя функции в Haskell заканчивается нижним подчёркиванием, это означает, что вы отбрасываете результаты.

Внеся небольшие изменения в код, вы получите:

```
main :: IO ()
main = do
  args <- getArgs
  mapM_ putStrLn args
```

*Упражнение

Напишите функцию main, использующую mapM, чтобы трижды вызвать getLine, а затем используйте mapM_, чтобы напечатать введённые значения (подсказка: вам нужно отбросить аргумент при использовании mapM с getLine, для достижения этого используйте (_ -> ...)).

*Упражнение (ответ)

```
exampleMain :: IO ()
exampleMain = do
  vals <- mapM (\_ -> getLine) [1..3]
  mapM_ putStrLn vals
```

*Функция print

Реализация функции print выглядит как (putStrLn. show) и облегчает вывод значения любого типа.

Количество строк как аргумент командной строки

```
main :: IO ()
main = do
  args <- getArgs
  let linesToRead = if length args > 0
                    then read (head args)
                    else 0 :: Int
  print linesToRead
```

*replicateM

Теперь, когда вы знаете, сколько строк вам нужно считать, нужно столько раз вызвать getLine. В Haskell есть подходящая для этого функция под названием replicateM. Она принимает число, обозначающее количество раз, которое вы хотите выполнить действие ввода-вывода, и повторяет соответствующее действие указанное количество раз. Для её использования вам нужно импортировать модуль Control.Monad.

*Считывание строк в заданном количестве

```
import Control.Monad
main :: IO ()
main = do
args <- getArgs
let linesToRead = if length args > 0
                  then read (head args)
                  else 0 :: Int
numbers <- replicateM linesToRead getLine
print "summa"
```

У вас почти получилось! Как вы помните, функция getLine возвращает String в контексте IO. Перед тем как вы сможете вычислить сумму всех аргументов, вам нужно преобразовать их к типу Int, а затем вернуть сумму списка целых чисел

```
import System.Environment
import Control.Monad
```

*Полная реализация программы

```
main :: IO ()
main = do
args <- getArgs
let linesToRead = if length args > 0
                  then read (head args)
                  else 0 :: Int
numbers <- replicateM linesToRead getLine
let ints = map read numbers :: [Int]
print (sum ints)
```

Что делает программа?

*Ответ

Программа, позволяющая пользователям вводить произвольное количество целых чисел и складывать их:

```
>main 2
4
59
63
>main 4
1
2
3
410
416
```

*Функции для повторения действий в контексте IO

Функция	Поведение
mapM	Принимает на вход действие ввода-вывода и обычный список, выполняет действие на каждом элементе списка и возвращает список в контексте IO
mapM_	Работает как mapM, но отбрасывает результат (обратите внимание на _)
replicateM	Принимает на вход действие ввода-вывода и целое число n, повторяет действие n раз и возвращает результаты в виде списка в контексте IO
replicateM_	Работает как replicateM, но отбрасывает результат

*Упражнение

Напишите собственную версию replicateM под названием myVersionM, использующую mapM (можете не задумываться о типовой аннотации).

*Упражнение (ответ)

```
myVersionM :: Monad m => Int -> m a -> m [a]
myVersionM n func = mapM (\_ -> func) [1 .. n]
```