МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
Факультет прикладной математики и информатики

ОТЧЁТ

по лабораторной работе № 6

«Ввод-вывод и монада IO»

по дисциплине «Функциональное программирование»

Выполнил: студент группы ПМИМ-61

Резанко А. А.

Проверил: _____

Новосибирск, 2026

Лабораторная работа № 6

Ввод-вывод и монада IO

Цель работы

Освоить систему ввода-вывода Haskell, работу с монадой IO, нотацию do, файловые операции и взаимодействие чистого кода с побочными эффектами.

Теоретические сведения

Haskell разделяет чистые вычисления и побочные эффекты на уровне типов. Значение типа ввода-вывода – это описание действия, которое при выполнении даст результат указанного типа. Само по себе оно ничего не выполняет: действие запускается рантаймом, когда входит в состав точки входа программы.

Нотация do позволяет записывать последовательность действий построчно, а стрелка влево связывает результат действия с именем. Это лишь сокращённая запись цепочки связываний. Чистую функцию можно вызвать из действия, обратное невозможно: из типа ввода-вывода нельзя извлечь значение в чистом коде, и именно этот запрет гарантирует, что чистые функции остаются чистыми.

Средства и порядок выполнения

Компилятор GHC 9.4.7, операционная система Ubuntu 24.04. Исходный текст программы размещён в файле Lab6.hs и приведён полностью в приложении А. Сборка и запуск:

```
ghc -O0 -Wall Lab6.hs -o lab6
./lab6 < input_lab6.txt
```

Ключ -Wall включает полный набор предупреждений компилятора; сборка проходит без замечаний. В программу встроено 14 проверок: каждая сверяет результат функции с ожидаемым значением и печатает отметку в протоколе. Вместо экранных снимков в отчёте приведён текстовый протокол

консольного вывода – он воспроизводится запуском программы и содержит те же данные.

Ход работы

6.1 Базовый ввод-вывод

Программа печатает приглашение, читает строку и выводит приветствие. Стрелка влево связывает результат чтения с именем; дальше это обычная строка, с которой работают чистые функции – в данном случае конкатенация.

Чтение оформлено вспомогательной функцией: она печатает приглашение, сбрасывает буфер вывода и, если ввод подаётся не с терминала, печатает прочитанную строку. Без этого в протоколе остались бы одни приглашения без ответов – при перенаправлении ввода из файла эхо не появляется. Различие определяется проверкой того, связан ли поток ввода с терминалом.

Листинг:

```
task61 :: IO ()
task61 = do
  name <- ask "Как вас зовут? "
  putStrLn ("Привет, " ++ name ++ "!")
```

Результат работы:

```
Как вас зовут? Александр
Привет, Александр!
```

6.2 Вычисление выражений с консоли

Введённые строки разбираются в числа. В методичке для этого используется функция read, которая на нечисловом вводе завершает программу исключением. Здесь взят её безопасный вариант, возвращающий Maybe: разбор обоих операндов проверяется одним сопоставлением с образцом по паре результатов, и при неудаче печатается сообщение.

Частное вычисляется по правилам плавающей арифметики, где деление на ноль даёт бесконечность, а не исключение. Такой вывод бесполезен, поэтому нулевой делитель обрабатывается отдельно.

Листинг:

```
task62 :: IO ()
task62 = do
  a <- ask "Введите первое число: "
  b <- ask "Введите второе число: "
  case (readMaybe a, readMaybe b) of
    (Just x, Just y) -> do
      putStrLn ("Сумма: " ++ show (x + y))
      putStrLn ("Разность: " ++ show (x - y))
      putStrLn ("Произведение: " ++ show (x * y))
      putStrLn ("Частное: " ++ if y == 0
                               then "не определено"
                               else show (x / y :: Double))
    _ -> putStrLn "Ожидались два числа"
```

Результат работы:

```
Введите первое число: 12.5
Введите второе число: 4
Сумма: 16.5
Разность: 8.5
Произведение: 50.0
Частное: 3.125
```

6.3 Работа с файлами

Программа читает файл, считает строки, слова и символы и записывает результат в другой файл. Подсчёты выполняют чистые функции, действие лишь доставляет им содержимое.

Порядок действий важен из-за ленивого чтения: содержимое читается по мере обращения к нему, и файл остаётся открытым, пока не прочитан до конца. Все три величины вычисляются до запроса имени выходного файла, поэтому к моменту записи входной файл уже закрыт. Если указать одно и то же имя для входа и выхода, программа завершится ошибкой блокировки файла.

Для проверки результата выходной файл читается обратно и печатается. Значения из него сверяются со статистикой исходного текста встроенными проверками – это подтверждает, что записанные числа соответствуют содержимому файла.

Листинг:

```
task63 :: IO ()
task63 = do
  inputPath <- ask "Введите путь к входному файлу: "
  content <- readFile inputPath
  let lineCount = length (lines content)
      wordCount = length (words content)
      charCount = length content
  outputPath <- ask "Введите путь к выходному файлу: "
  writeFile outputPath $
    "Строк: "    ++ show lineCount ++ "\n" ++
```

```
    "Слов: "     ++ show wordCount ++ "\n" ++
    "Символов: " ++ show charCount ++ "\n"
  putStrLn "Готово!"
  putStrLn ("Содержимое " ++ outputPath ++ ":")
  readFile outputPath >>= putStr
```

Результат работы:

```
Введите путь к входному файлу: sample.txt
Введите путь к выходному файлу: stats.txt
Готово!
Содержимое stats.txt:
Строк: 4
Слов: 28
Символов: 207
```

6.4 Интерактивный калькулятор

Цикл выражен рекурсией: после обработки строки функция вызывает себя, а выход происходит по команде завершения. Дополнительно перед чтением проверяется конец ввода – иначе при исчерпании входного файла программа завершилась бы исключением.

Разбор строки идёт в три уровня вложенных сопоставлений: сначала проверяется, что слов ровно три, затем что первое и третье – числа, затем вычисляется операция.

Вычисление вынесено в чистую функцию и возвращает Either вместо Maybe. В методичке обе ошибочные ситуации – неизвестная операция и деление на ноль – дают одно и то же отсутствующее значение, и пользователь получает сообщение о некорректной операции даже при делении на ноль. Either переносит причину отказа, и сообщения различаются, что видно в протоколе.

Листинг:

```
-- Either вместо Maybe: неизвестная операция и деление на ноль –
-- разные ошибки, и сообщение должно их различать
apply :: String -> Double -> Double -> Either String Double
apply "+" x y = Right (x + y)
apply "-" x y = Right (x - y)
apply "*" x y = Right (x * y)
apply "/" _ 0 = Left "деление на ноль"
apply "/" x y = Right (x / y)
apply op  _ _ = Left ("неизвестная операция: " ++ op)

task64 :: IO ()
task64 = do
  putStrLn "=== Калькулятор ==="
  putStrLn "Доступные операции: + - * / (или quit для выхода)"
  calculatorLoop

-- Цикл выражен рекурсией: хвостовой вызов после обработки строки
calculatorLoop :: IO ()
calculatorLoop = do
```

```
    eof <- isEOF
    if eof
      then putStrLn "Ввод закончился"
      else do
        input <- ask "> "
        if input == "quit"
          then putStrLn "До свидания!"
          else do
            case words input of
              [a, op, b] ->
                case (readMaybe a, readMaybe b) of
                  (Just x, Just y) ->
                    case apply op x y of
                      Right r  -> putStrLn ("= " ++ show r)
                      Left err -> putStrLn ("Ошибка: " ++ err)
                  _ -> putStrLn "Ошибка: операнды должны быть числами"
              _ -> putStrLn "Формат: число операция число"
            calculatorLoop
```

Результат работы:

```
=== Калькулятор ===
Доступные операции: + - * / (или quit для выхода)
> 2 + 3
= 5.0
> 10 / 4
= 2.5
> 7 * 6
= 42.0
> 10 / 0
Ошибка: деление на ноль
> 2 ^ 3
Ошибка: неизвестная операция: ^
> 1 2
Формат: число операция число
> quit
До свидания!
```

6.5 Взаимодействие чистого кода и ввода-вывода

Задание требует использовать функции из лабораторной работы № 2, поэтому та программа объявлена модулем с именем Lab2 и собирается с указанием точки входа. Здесь из неё импортированы суммирование и длина списка, написанные рекурсивно без стандартной библиотеки.

Разбор содержимого файла вынесен в чистую функцию: она разбивает текст на строки и оставляет только те, что разобрались в число. Такая функция проверяется без файлов – в протоколе есть проверка на строке с пустой и нечисловой строками. Прямое использование read, как в методичке, завершилось бы исключением на завершающем переводе строки.

Пустой список обрабатывается отдельно: среднее по нему не определено, а минимум и максимум на пустом списке дают ошибку времени выполнения.

Листинг:

```
-- Разбор строк вынесен в чистую функцию: её можно проверить без файлов.
-- mapMaybe отбрасывает пустые и нечисловые строки, тогда как read на
-- них завершился бы исключением.
parseNumbers :: String -> [Double]
parseNumbers = mapMaybe readMaybe . lines

task65 :: IO ()
task65 = do
  content <- readFile "numbers.txt"
  let nums = parseNumbers content
  if null nums
    then putStrLn "Файл numbers.txt не содержит чисел"
    else do
      let total = mySum nums
          avg   = total / fromIntegral (myLength nums)
      putStrLn ("Количество: " ++ show (myLength nums))
      putStrLn ("Сумма: "      ++ show total)
      putStrLn ("Среднее: "    ++ show avg)
      putStrLn ("Минимум: "    ++ show (minimum nums))
      putStrLn ("Максимум: "   ++ show (maximum nums))
```

Результат работы:

```
Количество: 7
Сумма: 65.25
Среднее: 9.321428571428571
Минимум: -8.5
Максимум: 41.0
```

6.6 Монада ввода-вывода без нотации do

Первое задание переписано операторами связывания. Действие без результата соединяется со следующим оператором, отбрасывающим значение, а чтение строки – оператором, передающим результат лямбде.

Именно в такую цепочку разворачивается нотация do. Видно, что тело лямбды охватывает весь остаток цепочки: связанное имя доступно во всех последующих действиях, и никакого изменяемого состояния при этом не возникает.

Эхо ввода здесь не печатается – вспомогательная функция чтения не используется, чтобы запись осталась дословно такой, как в задании. Введённое имя видно в самом приветствии.

Листинг:

```
-- То же, что task61, но записанное операторами связывания:
-- именно в такую цепочку разворачивается do-нотация
task66 :: IO ()
task66 =
  putStrLn "Как вас зовут?" >>
  getLine >>= \name ->
  putStrLn ("Привет, " ++ name ++ "!")
```

Результат работы:

```
Как вас зовут?
Привет, Александр!
```

Тестирование

Полный вывод программы за один запуск, включая встроенные проверки:

```
=== 6.1. Базовый ввод-вывод ===
Как вас зовут? Александр
Привет, Александр!

=== 6.2. Вычисление выражений с консоли ===
Введите первое число: 12.5
Введите второе число: 4
Сумма: 16.5
Разность: 8.5
Произведение: 50.0
Частное: 3.125

=== 6.3. Работа с файлами ===
Введите путь к входному файлу: sample.txt
Введите путь к выходному файлу: stats.txt
Готово!
Содержимое stats.txt:
Строк: 4
Слов: 28
Символов: 207

=== 6.4. Интерактивный калькулятор ===
=== Калькулятор ===
Доступные операции: + - * / (или quit для выхода)
> 2 + 3
= 5.0
> 10 / 4
= 2.5
> 7 * 6
= 42.0
> 10 / 0
Ошибка: деление на ноль
> 2 ^ 3
Ошибка: неизвестная операция: ^
> 1 2
Формат: число операция число
> quit
До свидания!

=== 6.5. Чистый код и ввод-вывод ===
Количество: 7
Сумма: 65.25
Среднее: 9.321428571428571
Минимум: -8.5
Максимум: 41.0

=== 6.6. Монада IO без do-нотации ===
Как вас зовут?
Привет, Александр!

=== Самопроверка ===
  [OK  ] apply "+" 2 3 == Right 5
  [OK  ] apply "-" 2 3 == Right (-1)
  [OK  ] apply "*" 6 7 == Right 42
  [OK  ] apply "/" 10 4 == Right 2.5
  [OK  ] деление на ноль даёт Left
  [OK  ] неизвестная операция даёт Left
  [OK  ] readMaybe отсеивает нечисловой ввод
  [OK  ] parseNumbers пропускает пустые строки
  [OK  ] stats.txt содержит число строк sample.txt
  [OK  ] stats.txt содержит число слов sample.txt
  [OK  ] stats.txt содержит число символов sample.txt
  [OK  ] numbers.txt разобран полностью
  [OK  ] mySum из лабораторной № 2 совпадает с sum
  [OK  ] среднее лежит между минимумом и максимумом
```

```
Пройдено 14 из 14 проверок.
```

Все 14 проверок пройдены (14 из 14), расхождений с ожидаемыми значениями нет.

Ответы на контрольные вопросы

1. Почему нельзя вызвать действие ввода-вывода из чистой функции? Как тогда чистый код получает данные из внешнего мира?

Потому что из типа ввода-вывода не предусмотрено извлечения значения: в стандартной библиотеке нет функции, превращающей действие в его результат. Запрет умышленный. Если бы чтение файла можно было вызвать изнутри чистой функции, та перестала бы давать один и тот же результат на одних и тех же аргументах, а вместе с этим потерялись бы ленивость, переупорядочивание вычислений и возможность рассуждать о коде подстановкой равного за равное.

Данные передаются в обратном направлении: действие получает их из внешнего мира и передаёт чистой функции обычным аргументом. В задании 6.5 чтение файла выполняет действие, а разбор текста, суммирование и вычисление среднего – чистые функции. Граница проходит по типу: всё, что связано с внешним миром, видно в сигнатуре.

2. В чём разница между двумя операторами связывания?

Оба соединяют действия в последовательность, но по-разному поступают с результатом первого. Оператор с передачей результата принимает справа функцию, которая этот результат получит аргументом и вернёт следующее действие. Оператор без передачи результат отбрасывает и просто выполняет второе действие.

Второй выражается через первый: связывание с функцией, игнорирующей аргумент. Применяют его там, где результат неинформативен, – после печати строки, которая возвращает пустой кортеж. В нотации do первому отвечает строка со стрелкой влево, второму – строка без неё; это видно в задании 6.6, где обе формы выписаны явно.

3. Что делает чтение файла? Загружает ли оно файл в память целиком?

Возвращает действие, результат которого – содержимое файла строкой. Чтение ленивое: файл открывается сразу, но содержимое подтягивается порциями по мере обращения к строке, а дескриптор закрывается при достижении конца файла.

Поэтому сразу в память файл не загружается, и посчитать строки в большом файле можно без его полного размещения в памяти. Но если ссылку на строку удерживать и проходить её несколько раз, прочитанное сохранится целиком – как в задании 6.3, где по содержимому считаются три величины.

Практическое следствие ленивости – файл остаётся открытым до конца чтения. Попытка записать в тот же файл до того, как он дочитан, приводит к ошибке блокировки. Когда такое поведение мешает, используют строгое чтение из библиотек работы с текстом.

4. Как нотация do связана с монадой? Во что компилируется строка со стрелкой влево?

Нотация do – синтаксический сахар над операциями монады, она не добавляет языку выразительности. Строка со стрелкой влево компилируется в связывание: действие слева от стрелки соединяется оператором передачи результата с лямбдой, параметр которой – имя слева, а тело – весь остаток блока.

Строка без стрелки превращается в связывание с отбрасыванием результата, а локальные привязки через let остаются обычными привязками. Разворачивание идёт сверху вниз, поэтому вложенность растёт вправо – это и показывает задание 6.6.

Отсюда же следует, что нотация доступна любому типу с экземпляром монады, а не только вводу-выводу: в лабораторной работе № 5 тот же синтаксис использован для Maybe, списка и собственного контейнера.

Выводы

Написаны шесть программ ввода-вывода: диалог, разбор чисел с консоли, подсчёт статистики файла с записью результата, рекурсивный цикл калькулятора, объединение чистого кода с чтением файла и та же программа без нотации do.

Граница между чистым кодом и эффектами прошла по типам: разбор текста, арифметика калькулятора и статистика оформлены чистыми функциями, а действия лишь доставляют им данные. Благодаря этому логика проверяется без файлов и консоли – встроенные проверки вызывают чистые функции напрямую.

Разобранные в методичке решения дополнены в трёх местах: безопасный разбор чисел вместо аварийного, различимые сообщения об ошибках калькулятора через Either и проверка конца ввода в цикле. Программа собирается без предупреждений при ключе -Wall и проходит все 14 встроенных проверок, сверяющих в том числе содержимое записанного файла с исходным текстом.

ПРИЛОЖЕНИЕ А. Полный листинг программы (Lab6.hs)

```
-- Лабораторная работа № 6. Ввод-вывод и монада IO
-- Резанко А. А., ПМИМ-61
--
-- Задание 6.5 использует чистые функции из лабораторной работы № 2,
-- поэтому Lab2.hs должен лежать рядом.
--
-- Сборка:  ghc -O0 -Wall Lab6.hs -o lab6
-- Запуск:  ./lab6 < input_lab6.txt   (или ./lab6 в диалоговом режиме)

module Main where

import Data.Maybe (mapMaybe)
import GHC.IO.Encoding (setLocaleEncoding)
import Lab2 (myLength, mySum)
import System.IO
import Text.Read (readMaybe)

-- Запрос строки с приглашением. Когда ввод подаётся из файла, а не с
-- терминала, введённая строка печатается обратно – иначе в протоколе
-- остались бы одни приглашения без ответов.
ask :: String -> IO String
ask prompt = do
  putStr prompt
  hFlush stdout
  line <- getLine
  tty <- hIsTerminalDevice stdin
  if tty then return line else putStrLn line >> return line

-- ====================== 6.1. Базовый ввод-вывод ======================

task61 :: IO ()
task61 = do
  name <- ask "Как вас зовут? "
  putStrLn ("Привет, " ++ name ++ "!")

-- ====================== 6.2. Вычисление выражений с консоли ======================

task62 :: IO ()
task62 = do
  a <- ask "Введите первое число: "
  b <- ask "Введите второе число: "
  case (readMaybe a, readMaybe b) of
    (Just x, Just y) -> do
      putStrLn ("Сумма: " ++ show (x + y))
      putStrLn ("Разность: " ++ show (x - y))
      putStrLn ("Произведение: " ++ show (x * y))
      putStrLn ("Частное: " ++ if y == 0
                               then "не определено"
                               else show (x / y :: Double))
    _ -> putStrLn "Ожидались два числа"

-- ====================== 6.3. Работа с файлами ======================

task63 :: IO ()
task63 = do
  inputPath <- ask "Введите путь к входному файлу: "
  content <- readFile inputPath
  let lineCount = length (lines content)
      wordCount = length (words content)
      charCount = length content
  outputPath <- ask "Введите путь к выходному файлу: "
  writeFile outputPath $
    "Строк: "    ++ show lineCount ++ "\n" ++
    "Слов: "     ++ show wordCount ++ "\n" ++
    "Символов: " ++ show charCount ++ "\n"
  putStrLn "Готово!"
  putStrLn ("Содержимое " ++ outputPath ++ ":")
  readFile outputPath >>= putStr
```

```haskell
-- ==================== 6.4. Интерактивный калькулятор ====================

-- Either вместо Maybe: неизвестная операция и деление на ноль –
-- разные ошибки, и сообщение должно их различать
apply :: String -> Double -> Double -> Either String Double
apply "+" x y = Right (x + y)
apply "-" x y = Right (x - y)
apply "*" x y = Right (x * y)
apply "/" _ 0 = Left "деление на ноль"
apply "/" x y = Right (x / y)
apply op  _ _ = Left ("неизвестная операция: " ++ op)

task64 :: IO ()
task64 = do
  putStrLn "=== Калькулятор ==="
  putStrLn "Доступные операции: + - * / (или quit для выхода)"
  calculatorLoop

-- Цикл выражен рекурсией: хвостовой вызов после обработки строки
calculatorLoop :: IO ()
calculatorLoop = do
  eof <- isEOF
  if eof
    then putStrLn "Ввод закончился"
    else do
      input <- ask "> "
      if input == "quit"
        then putStrLn "До свидания!"
        else do
          case words input of
            [a, op, b] ->
              case (readMaybe a, readMaybe b) of
                (Just x, Just y) ->
                  case apply op x y of
                    Right r  -> putStrLn ("= " ++ show r)
                    Left err -> putStrLn ("Ошибка: " ++ err)
                _ -> putStrLn "Ошибка: операнды должны быть числами"
            _ -> putStrLn "Формат: число операция число"
          calculatorLoop

-- ==================== 6.5. Чистый код и ввод-вывод ====================

-- Разбор строк вынесен в чистую функцию: её можно проверить без файлов.
-- mapMaybe отбрасывает пустые и нечисловые строки, тогда как read на
-- них завершился бы исключением.
parseNumbers :: String -> [Double]
parseNumbers = mapMaybe readMaybe . lines

task65 :: IO ()
task65 = do
  content <- readFile "numbers.txt"
  let nums = parseNumbers content
  if null nums
    then putStrLn "Файл numbers.txt не содержит чисел"
    else do
      let total = mySum nums
          avg   = total / fromIntegral (myLength nums)
      putStrLn ("Количество: " ++ show (myLength nums))
      putStrLn ("Сумма: "      ++ show total)
      putStrLn ("Среднее: "    ++ show avg)
      putStrLn ("Минимум: "    ++ show (minimum nums))
      putStrLn ("Максимум: "   ++ show (maximum nums))

-- ==================== 6.6. Монада IO без do-нотации ====================

-- То же, что task61, но записанное операторами связывания:
-- именно в такую цепочку разворачивается do-нотация
task66 :: IO ()
task66 =
  putStrLn "Как вас зовут?" >>
```

```
  getLine >>= \name ->
  putStrLn ("Привет, " ++ name ++ "!")

-- ==================== Протокол выполнения ====================

section :: String -> IO ()
section t = putStrLn ("\n=== " ++ t ++ " ===")

check :: String -> Bool -> IO Bool
check label ok = do
  putStrLn ("  [" ++ (if ok then "OK  " else "FAIL") ++ "] " ++ label)
  return ok

main :: IO ()
main = do
  setLocaleEncoding utf8
  hSetEncoding stdin utf8
  hSetEncoding stdout utf8
  hSetBuffering stdout LineBuffering

  section "6.1. Базовый ввод-вывод"
  task61

  section "6.2. Вычисление выражений с консоли"
  task62

  section "6.3. Работа с файлами"
  task63

  section "6.4. Интерактивный калькулятор"
  task64

  section "6.5. Чистый код и ввод-вывод"
  task65

  section "6.6. Монада IO без do-нотации"
  task66

  section "Самопроверка"
  sample <- readFile "sample.txt"
  stats  <- readFile "stats.txt"
  numbers <- readFile "numbers.txt"
  let nums = parseNumbers numbers
  results <- sequence
    [ check "apply \"+\" 2 3 == Right 5" (apply "+" 2 3 == Right 5)
    , check "apply \"-\" 2 3 == Right (-1)" (apply "-" 2 3 == Right (-1))
    , check "apply \"*\" 6 7 == Right 42" (apply "*" 6 7 == Right 42)
    , check "apply \"/\" 10 4 == Right 2.5" (apply "/" 10 4 == Right 2.5)
    , check "деление на ноль даёт Left"
        (apply "/" 1 0 == Left "деление на ноль")
    , check "неизвестная операция даёт Left"
        (apply "^" 2 3 == Left "неизвестная операция: ^")
    , check "readMaybe отсеивает нечисловой ввод"
        ((readMaybe "два" :: Maybe Double) == Nothing)
    , check "parseNumbers пропускает пустые строки"
        (parseNumbers "1\n\n2\nтри\n3" == [1, 2, 3])
    , check "stats.txt содержит число строк sample.txt"
        (("Строк: " ++ show (length (lines sample))) `elem` lines stats)
    , check "stats.txt содержит число слов sample.txt"
        (("Слов: " ++ show (length (words sample))) `elem` lines stats)
    , check "stats.txt содержит число символов sample.txt"
        (("Символов: " ++ show (length sample)) `elem` lines stats)
    , check "numbers.txt разобран полностью"
        (myLength nums == length (filter (not . null) (lines numbers)))
    , check "mySum из лабораторной № 2 совпадает с sum"
        (abs (mySum nums - sum nums) < 1e-9)
    , check "среднее лежит между минимумом и максимумом"
        (let avg = mySum nums / fromIntegral (myLength nums)
         in minimum nums <= avg && avg <= maximum nums)
    ]
  putStrLn ("\nПройдено " ++ show (length (filter id results))
```

```
++ " из " ++ show (length results) ++ " проверок.")
```

ПРИЛОЖЕНИЕ Б. Сценарий ввода для воспроизводимого запуска (input_lab6.txt)

```
Александр
12.5
4
sample.txt
stats.txt
2 + 3
10 / 4
7 * 6
10 / 0
2 ^ 3
1 2
quit
Александр
```

ПРИЛОЖЕНИЕ В. Входной текстовый файл для задания 6.3 (sample.txt)

```
Haskell — чистый функциональный язык программирования.
Вычисления в нём ленивые, а типы выводятся автоматически.
Побочные эффекты отделены от чистого кода типом IO.
Эта строка нужна для подсчёта статистики.
```

ПРИЛОЖЕНИЕ Г. Файл с числами для задания 6.5 (numbers.txt)

```
12.5
-3
7.25
0
41
-8.5
16
```