Душкин Роман Викторович
Специалист по функциональному программированию. Автор книги «Функциональное программирование на языке Haskell».

Язык Haskell: ввод и вывод

В настоящей статье применительно к функциональному языку Haskell рассматривается такой немаловажный аспект любого языка программирования, как система ввода/вывода. Приводятся основные определения, способы использования и примеры построения функций, реализующих ввод/вывод и взаимодействие с внешним миром (окружением программы).

Введение

Язык Haskell является *чистым* функциональным языком программирования, а это значит, что любая функция, которая определена при разработке программ, должна быть *детерминированной*, а также не должна использовать в своей работе так называемые *побочные эффекты*.

Под детерминированностью функций понимают такое свойство, что результат, вычисляемый функцией, зависит только от значений входных параметров. Это значит, что для любого заданного набора значений входных параметров результат детерминирован (определён) функцией. Два вызова функции с одним и тем же набором значений входных параметров всегда возвратят один и тот же результат.

Отсутствие побочных эффектов означает, в свою очередь, что функция в процессе своей работы обращается и изменяет только ту область памяти, которая выделена для её работы. В функциональном программировании в этой области памяти расположены значения выходных параметров, которые доступны только для чтения, а также результат выполнения функции, который доступен для чтения и записи. Кроме того, в области памяти, выделенной для работы функции, могут быть созданы так называемые *замыкания*, т.е. локальные переменные, область видимости (контекст) которых ограничен функцией.

Всё вышеперечисленное налагает на чистый язык программирования определённые ограничения. Например, невозможно программировать в терминах изменения некоторых глобальных переменных, поскольку их использование сделает некоторые функции недетерминированными (функция, к примеру, может возвращать значения некоторой глобальной

переменной, которая изменяется в другой функции, и потому в зависимости от времени вызова первой функции её результаты могут быть различными) и использующими сторонние эффекты (изменение глобальных переменных и есть сторонний эффект).

Другая проблема – система ввода/вывода языка, т.е. общение с внешним миром (или, как говорят, с *окружением программы*). В общем виде ввод/вывод можно понимать как чтение информации с какого-либо внешнего устройства (например, клавиатуры), а также запись данных опять-таки на внешнее устройство (например, в видеозапись терминала). Для понимания сущности проблемы организации системы ввода/вывода в чистом языке программирования необходимо детально рассмотреть оба случая.

Ввод – это чтение данных с внешнего устройства – клавиатуры, мыши, из файла, из сети и т. д. Например, пусть имеется некоторая функция **read**, которая считывает с клавиатуры один символ и возвращает его. Каков должен быть тип этой функции? Совершенно ясно, что принимать на вход ей ничего не нужно. А возвращает такая функция символ, т.е. значение типа **Char**. Таким образом, её *сигнатура* (описание типа) выглядит следующим образом:

```
read :: Char
```

Однако ясно, что эта функция не детерминирована, поскольку её результат будет зависеть от того, какой символ на клавиатуре нажал пользователь. Если пользователь нажмёт кнопку «A», функция вернёт символ с кодом 65. Нажмёт другую кнопку – функция возвратит другой код. Полная недетерминированность в поведении.

Вывод, т.е. запись данных на некоторое внешнее устройство – на экран, в файл, в сеть и т.д. Пусть также имеется некоторая функция **put**, которая выводит заданный символ на экран. Какой тип у этой функции? Ясно, что она должна принимать на вход один параметр – код символа, который должен быть выведен. Так что тип первого (и единственного) параметра – **Char**. А что должна возвращать эта функция? Какое значение? Совершенно неясно. Скорее всего, она не должна возвращать ничего. В языках программирования типа C для этих целей используется тип **void**. В языке Haskell используется тип «пустой кортеж», т.е. последовательность значений, состоящая из нуля элементов ( ). Так что сигнатура функции **put** выглядит так:

```
put :: Char -> ()
```

Эта функция использует сторонние эффекты, поскольку модифицирует внешнюю память – видеопамять экрана, на который производится вывод. Если бы функция записывала информацию в файл, она модифицировала бы память на носителе информации, где записан файл.

Видно, что и ввод, и вывод являются недопустимыми с точки зрения чистого языка программирования. Однако является совершеннейшим абсурдом отсутствие системы ввода/вывода в языке программирова-

ния общего назначения. Для чего нужен такой язык? Как организовывать взаимодействие с пользователем? Как работать с файлами? Как организовывать многопотоковые программы и распределённые вычисления с передачей информации между потоками? Как взаимодействовать с внешними программами по сети? Ведь без перечисленных технологий в современном мире делать нечего. И если в языке программирования не существует средств для работы с ними, этому языку просто нет места в арсенале современного программирования.

Естественно, что в языке Haskell имеются все необходимые средства для работы с перечисленными технологиями (и даже больше), поскольку этот язык является языком общего назначения. Но возникает резонный вопрос: как может ужиться чистота языка с заведомо «нечистыми» способами вычислений? Этот вопрос на примере системы ввода/вывода и будет детально исследован в этой статье далее.

1. Основы функционального вывода

Все опубликованные ранее статьи в журнале «Потенциал», посвящённые вопросам использования языка Haskell, обходились без ввода или вывода информации. Рассматривались расчётные задачи, создавались системы взаимосвязанных функций, а результаты их работы исследовались с помощью интерпретатора (например, HUGS 98). Интерпретатор всегда выводит на экран результат, возвращаемый функцией, а потому использование ввода/вывода при выполнении расчётных задач смысла особого не имело.

Однако разработка современного программного обеспечения, как уже указано во введении, не может обойтись без взаимодействия с окружением программы, в первую очередь, без взаимодействия с пользователем. Интерактивные программы уже настолько плотно вошли в нашу жизнь, что работа с интерпретаторами для получения результатов выглядит глубоко архаичной. Так что для умения создавать активно взаимодействующие программы необходимо изучить основы системы ввода/вывода в языке Haskell.

Но как быть с недерминированностью и наличием побочных эффектов? Если принять их как есть, то можно распрощаться с чистотой языка. Необходимо отметить, что некоторые функциональные языки пошли в своей эволюции именно по этому пути – их разработчики пожертвовали чистотой, но внедрили в язык обычную систему ввода/вывода. Но разработчики языка Haskell пошли иным путём.

В языке Haskell создан некоторый достаточный «мирок», где разрешены недетерминированность и побочные эффекты. Этот «мирок» ограничен в случае системы ввода/вывода единственным типом данных, с которым можно работать, – **`IO`**. Любая функция, которая осуществляет ввод или вывод, должна возвращать (а иногда и принимать на вход) значения типа **`IO`**. И более того, нет возможности выхода из этого «маленького мирка» – как только некоторый вычислительный процесс «попал в лапы» к системе ввода/вывода, он обречён оставаться в ней до конца.

Итак, рассмотренные во введении гипотетические функции **`read`** и **`put`** в этом случае должны иметь следующие сигнатуры:

```
read :: IO Char

put :: Char -> IO ()
```

Как видно из этих описаний, тип **IO** представляет собой *контейнерный* тип – внутри себя он может содержать значения произвольного типа. Другим контейнерным типом, к примеру, является список **[]**. Да и более того, любой *алгебраический тип данных*, если только это не *перечисление*, является контейнерным типом. Так что в этом нет ничего особенного.

Тип данных **IO** как бы оборачивает собой недетерминированные операции и операции, связанные с побочными эффектами. Поэтому в рамках ввода/вывода такие операции и могут происходить только внутри типа **IO**. Это сделано для того, чтобы оставить сам язык Haskell чистым.

Но это ещё не всё. Для выстраивания последовательности действий ввода/вывода в синтаксисе языка Haskell имеется специальное ключевое слово **do**, которое позволяет последовательно выполнить несколько операций ввода/вывода одну за другой. Например, при помощи исследованных ранее гипотетических функций **read** и **put** можно написать несложную программу – считать символ с клавиатуры и вывести его на экран. На языке Haskell это будет выглядеть следующим образом:

```
do c <- read
   put c
```

Ключевое слово **do** поддерживает так называемый *двумерный синтаксис*, когда последовательность операций ввода/вывода можно не отделять друг от друга точкой с запятой (;), как того требует строгий синтаксис языка, а записывать в столбик друг под другом. Главное, чтобы начало каждой операции находилось на том же знакоместе на новой строке, что и весь столбец.

В синтаксисе ключевого слова **do** имеются два момента. Первый – использование символа (**<-**). Этот символ вводит *образец*, который сопоставляется со значением, которое возвращается функцией после символа (**<-**). Если сопоставление прошло успешно, то ниже этот образец можно использовать с фактическим значением. Кроме того символ (**<-**) «разворачивает» тип **IO**, поэтому тип значений в образце не имеет контейнерной оболочки. Так что выполнение первой операции ввода/вывода в примере выше запишет в образец **c** код символа, который пользователь ввёл при помощи клавиатуры.

Второй момент – операции ввода/вывода без символа (**<-**). Эти операции просто выполняются, так что в примере выше на второй строке на экран будет выведен символ, код которого записан в образце **c**. Как видно, функция **put** принимает аргумент типа **Char**, и именно такой тип имеет значение в образце **c**, несмотря на то, что функция **read** возвращает значение типа **IO Char**.

Кроме указанных двух типов операций ввода/вывода, в конструкции **do** могут присутствовать определения замыканий посредством ключевого слова **let**, при этом использование ключевого слова **in** не нужно. Это – синтаксическое послабление,

которое ведёт к тому, что следующие два примера тождественны с точки зрения транслятора языка Haskell:

```
let p = someFunction a1 a2 a3
in  do outputProcess p

do let p = someFunction a1 a2 a3
   outputProcess p
```

Ещё один аспект – вложение выражений **do** друг в друга. Можно неограниченно вкладывать списки операций ввода/вывода друг в друга, только необходимо помнить, что выравнивание операций в строках соответствует тому или иному уровню выражения **do**. Нарушение принципов двумерного синтаксиса может привести не только к синтаксическим ошибкам, но и к логическим, которые не проявляются на этапе компиляции, но самым беспощадным образом нарушают работоспособность программы во время её исполнения. Кроме того, такие ошибки очень сложно отлавливать.

Наконец, необходимо упомянуть, что любое выражение **do** имеет тип (впрочем, как и любое выражение в функциональной парадигме). Тип выражения **do** равен типу последней операции ввода/вывода в списке. Поэтому в списке операций ввода/вывода на последнем месте обязательно должна стоять операция, которая имеет некоторый тип, а не возвращает его в образец (то есть в последней строке выражения **do** не должны использоваться символ (**<-**) и символ локального определения **let**).

Поскольку выражение **do** является обычным выражением с точки зрения транслятора языка Haskell, оно может использоваться во всех тех местах, где имеет смысл использование выражения соответствующего типа. Поэтому, к примеру, можно создавать следующие объекты (значения):

```
ioList = [do {c1 <- read; c2 <- read; put '>'},
          put 'a' ,
          do {put 'b'; put 'c'}]
```

Что интересно, определённая таким образом функция не приведёт к выполнению каких-либо операций ввода/вывода. При её вызове произойдёт простой возврат списка, состоящего из трёх операций ввода/вывода, каждая из которых имеет тип **IO** (). Так что тип функции **ioList** равен [**IO** ()]. Это – замечательное свойство системы ввода/вывода в языке Haskell, которое делает саму систему полностью функциональной.

2. Стандартные функции ввода/вывода

В стандартном модуле **Prelude** определён базовый набор функций, работающих в системе ввода/вывода в языке Haskell. Этих базовых функций вполне достаточно для решения большинства прикладных задач. В стандартной поставке любого транслятора языка имеется также дополнительный модуль **IO**, в котором определены расширенные возможности, позволяющие работать с каналами, исключениями (хотя и в моду-

ле **Prelude** имеются средства для этого, но весьма ограниченные) и представляющие множество вспомогательных функций для работы с файлами, консолью и клавиатурой. Модуль **IO** является достаточно большим, его описание выходит за рамки этой статьи. Поэтому ниже описываются только некоторые функции из стандартного модуля **Prelude**, которые позволят в разделе 3 создать несколько интересных прикладных программ.

Самым первым объектом, который определён в системе ввода/вывода Haskell, является синоним типа для представления путей к файлам. Этот синоним определён исключительно для удобочитаемости сигнатур функций, работающих с файлами. Его определение выглядит следующим образом:

```
type FilePath = String
```

Второй важный тип, который используется при работе с файлами, представляет собой перечисление, которое описывает режимы открытия файлов – открытие для чтения, записи, дозаписывания, а также для чтения и записи одновременно. Это стандартные константы, которые поддерживаются большинством современных операционных систем:

```
data IOMode
  = ReadMode
  | WriteMode
  | AppenMode
  | ReadWriteMode
  deriving (Eq, Ord, Ix, Bounded, Enum, Read, Show)
```

Очень важными функциями являются функции для работы с дескрипторами файлов. Все они определены в виде *примитивов*, т.е. объектов, запрятанных глубоко в недрах трансляторов и имеющих только отображение в стандартном модуле **Prelude**. Это отображение представляет собой сигнатуру, которая разрешает использование соответствующего идентификатора. Ниже перечисляются такие примитивы:

1. `openFile :: FilePath -> IOMode -> IO Handle` – открывает файл по заданному имени и режиму открытия, возвращает дескриптор файла, который впоследствии может использоваться для работы с открытым файлом;
2. `hClose :: Handle -> IO ()` – закрывает указанный при помощи дескриптора файл;
3. `hGetContents :: Handle -> IO String` – читает всё содержимое заданного файла в одну строку;
4. `hGetChar :: Handle -> IO Char` – читает один символ с текущей позиции в заданном файле;
5. `hGetLine :: Handle -> IO String` – читает одну строку с текущей позиции в заданном файле, при этом строка заканчивается символом перевода строки;

6. `hPutChar :: Handle -> Char -> IO ()` – записывает один символ на текущую позицию в заданном файле;
7. `hPutStr :: Handle -> String -> IO ()` – записывает строку на текущую позицию в заданном файле.

Тип **Handle** является синонимом типа **Int** для текущей реализации интерпретатора (HUGS 98), но может иметь иную реализацию в других трансляторах типа Haskell, поэтому при программировании необходимо использовать только этот синоним.

Кроме всего прочего, в виде примитивов определены *три константные функции*, которые возвращают стандартные потоки – **stdin** (стандартный поток ввода с клавиатуры), **stdout** (стандартный поток вывода в обычную консоль) и **stderr** (стандартный поток вывода в консоль ошибок). Все эти константные функции возвращают значения типа **Handle**, а сами возвращаемые значения отличаются от файлов тем, что их не надо ни открывать, ни закрывать – они всегда доступны для работы.

В принципе, перечисленных функций достаточно для выполнения произвольных операций ввода/вывода. Однако для удобной работы с клавиатурой и консолью определены дополнительные функции (все они выражены через перечисленные выше семь функций):

1. `putChar :: Char -> IO ()` – выводит заданный символ в стандартную консоль вывода **stdout**;
2. `putStr :: String -> IO ()` – выводит заданную строку в стандартную консоль вывода **stdout**;
3. `putStrLn :: String -> IO ()` – вариант функции **putStr**, добавляющий после выведенной в стандартную консоль вывода **stdout** строки символ перевода строки;
4. `print :: Show a => a -> IO ()` – выводит в стандартную консоль вывода **stdout** некоторое значение, которое может быть преобразовано в строку (тип этого значения должен иметь экземпляр класса **Show**);
5. `getChar :: IO Char` – читает со стандартного потока ввода **stdin** один символ;
6. `GetContents :: IO String` – считывает всё содержимое стандартного потока ввода **stdin** в одну строку;
7. `GetLine :: IO String` – считывает из стандартного потока ввода **stdin** одну строку, заканчивающуюся символом перевода строки;
8. `readLn :: Read a => IO a` – считывает из стандартного потока ввода **stdin** некоторое значение, которое может быть получено при помощи синтаксического разбора строки (тип этого значения должен иметь экземпляр класса **Read**), само значение должно занимать всю строку , заканчивающуюся символом перевода строки.

Как уже сказано, все эти функции выражены через примитивы. Вдумчивый читатель может самостоятельно открыть стандартный мо-

дуль **Prelude** для того, чтобы изучить способ такого выражения. Это позволит заодно изучить неплохие примеры разработки функциональных операций ввода/вывода.

Следующий набор функций предоставляет удобные инструменты для работы с файлами. Все они также выражены через перечисленные ранее семь примитивных функций:

1. `writeFile :: FilePath -> String -> IO ()` – создаёт файл с заданным именем, содержимым которого является заданная строка, сам файл после записи закрывается;
2. `appendFile :: FilePath -> String -> IO ()` – дозаписывает в заданный по имени файл заданную строку, сам файл после записи закрывается;
3. `readFile :: FilePath -> IO String` – открывает файл по заданному имени и считывает всё его содержимое в одну строку, которая возвращается в качестве результата, файл остаётся открытым;
4. `interact :: (String -> String) -> IO ()` – интересная функция, которая считывает всё содержимое стандартного потока ввода **stdin** в строку, применяет к этой строке заданную функцию, а результат работы этой функции выводит в стандартный поток вывода **stdout**.

Наконец, последней функцией для работы с системой ввода/вывода, которая описана в стандартном модуле **Prelude**, является функция **catch**. Эта функция предназначена для отлова *исключений*. Любая операция ввода/вывода в процессе этой работы может сгенерировать исключение некоторого типа. Это может произойти по многим причинам – заданный по имени файл отсутствует, нет прав доступа для чтения файла и т.д. Для того, чтобы программа не произвела аварийный останов с выходом в операционную систему, такие исключения необходимо ловить и обрабатывать. Для этого как раз и используется функция **catch**, которая имеет следующую сигнатуру:

```
catch :: IO a -> (IOError -> IO a) -> IO a
```

Эта функция принимает на вход два параметра – операцию ввода/вывода, а также обработчик некоторого исключения (типы исключений описываются перечислением **IO Error**). Если в процессе выполнения операции ввода/вывода исключений не произошло, функция **catch** возвращает результат операции. Если же произошло исключение, то оно передаётся на вход функции-обработчику, которая задана вторым аргументом функции **catch**. Эта функция может обработать исключение, тогда результат её работы будет результатом вызова функции **catch**, а может и не обработать. В последнем случае исключение будет сгенерировано повторно и передано в обработчик более высокого уровня. Этим достигается иерархичность обработки исключений. Самым верхним обработчиком исключений является системная функция, которая останавливает

программу и выводит стандартное сообщение об ошибке. Этот обработчик вызывается всегда, если исключение не обработано ни одним из обработчиков, определённых программистом.

Таким образом, в стандартном модуле **Prelude** определены достаточные средства для решения произвольных прикладных задач, связанных с использованием ввода/вывода. В следующем разделе будут приведены некоторые примеры использования этих функций.

3. Примеры программ

В языке Haskell имеется одно соглашение, сходное с соглашением об имени главной функции в таких языках программирования, как C или C++. Если программа готовится к компиляции, то в ней должна быть определена функция **main**, при этом она должна быть определена в модуле **Main** (или в главном модуле без наименования – по умолчанию наименование модуля берётся как раз **Main**), а её тип должен быть:

```
main :: IO ()
```

Если подобные программы использовать в интерпретаторах (например, в интерпретаторе HUGS 98), то функция **main** будет запущена на использование при нажатии на кнопку «Запуск» на панели инструментов интерпретатора.

Название главной функции должно учитываться при программировании прикладных задач.

3.1. Вывод результатов исполнения функции на экран

До этого во всех примерах, приводимых в статьях журнала, которые были посвящены языку Haskell, использовались функции, результаты которых выводились автоматически в интерпретаторе. Для этого было необходимо просто написать в строке приглашения интерпретатора наименование функции и список её фактических параметров, чтобы интерпретатор проинтерпретировал этот вызов функции и вывел результат на экран.

В этом моменте имеется одна очень важная вещь. Интерпретатор может вывести на экран только такое значение, тип которого имеет экземпляр **Show**. Например, функции с типом a **->** b не могут быть выведены на экран в качестве результата вычисления, поскольку для этого типа нет экземпляра класса **Show**. Обычно такая ошибка происходит, если вызвать какую-либо функцию, передав ей на одно фактическое значение меньше, чем того требует сигнатура. В этом случае произойдёт частичное применение, результатом которого будет функция одного аргумента, и интерпретатор выведет на экран примерно следующее:

```
ERROR - Cannot find "show" function for:
*** Expression : flip (+) 1
*** Of type : Integer -> Integer
```

Это происходит потому, что для типа любого значения, которое должен вывести на экран интерпретатор, он ищет экземпляр класса **Show**, чтобы преобразовать это значение в строку при помощи метода **show**, после чего строка выводится на экран. Весь этот процесс делается при помощи стандартной функции **print**.

То же самое необходимо делать при выводе результатов работы функции в откомпилированной программе. Однако весь этот процесс необходимо делать уже вручную. В качестве примера можно рассмотреть получение списка из пяти первых совершенных чисел. Быстро записав функции для получения списка совершенных чисел:

```
divisors :: Integer -> [Integer]
divisors x = [y | y <- [1..(div x 2)] ,
                  mod x y == 0]

perfects :: [Integer]
perfects = [x | x <- [2, 4..],
             sum (divisors x) == x],
```

необходимо задуматься, а как вывести первые пять чисел на экран? Для этого необходимо определить функцию **main**, в которой произойдёт вычисление первых пяти совершенных чисел, после чего результат будет выведен на экран при помощи операции ввода/вывода. Достаточно просто:

```
main :: IO ()
main = do let p = take 5 perfects
          print p
```

В принципе, для таких простых примеров можно обойтись и без конструкции **do**, поскольку одна операция ввода/ вывода может быть вызвана без этого ключевого слова, а само слово **do** необходимо только для связывания последовательности операций. Так что функция **main** может выглядеть так:

```
main :: IO ()
main = print $ take 5 perfects
```

3.2. Альтернатива: экран или файл

А что если для каких-то целей необходимо иметь возможность вывести результаты не только на экран, но и в некоторый файл, при этом предварительно имя файла необходимо запросить у пользователя. Для решения этой задачи такой простой функцией **main**, какая была в предыдущем подразделе, уже не обойтись. Пусть теперь необходимо вычислять список простых чисел, а для некоторого усложнения задачи количество чисел, которое необходимо вычислять, также будет запрашиваться у пользователя.

Функция для вычисления списка простых чисел несложная:

```
primes :: [Integer]
primes = sieve [2..]
  where
    sieve (x:xs) = x:sieve (filter ((/= 0).('mod' x))xs)
```

А вот функция **main** будет уже посложнее:

```
main :: IO ()
main = do putStr "Привет. Сколько простых чисел вычислить?"
          npr<- readLn :: IO Int
          let prs = take npr primes
          putStr "Куда вывести результаты (f – файл):"
          dst <- getLine
          if (dst /= "f")
            then print prs
            else do putStr "Пожалуйста, введите имя файла:"
                    fn <- getLine
                    writeFile fn (show prs)
                    put Str ("Файл \"" ++ fn ++ "\" записан.")
```

Эту функцию необходимо рассмотреть более подробно. Ниже подробно исследуется каждая операция ввода/вывода, записанная в списке выражения **do** этой функции.

Первая операция выводит на экран при помощи стандартной функции **putStr** приветствие и вопрос о том, сколько простых чисел необходимо подсчитать. Вторая строка как раз предназначена для считывания с клавиатуры целого числа, которое введёт пользователь. Это число будет положено в образец **npr**. Поскольку количество простых чисел, которое необходимо подсчитать, уже известно, это можно сделать – в третьей строке создаётся замыкание **prs**, которое содержит список простых чисел, состоящий из введённого количества элементов.

Четвёртая операция опять является запросом относительно того, куда вывести результаты. Соответственно, пятая операция считывает символ с клавиатуры. В шестой строке имеется условное выражение, оформленное инструкцией **if-then-else**. В принципе, любое выражение, которое имеет тип, обрамлённый контейнером **IO**, может присутствовать в списке **do**. Поэтому в списке **do** могут присутствовать выражения **if** или **case** – главное, чтобы они возвращали соответствующее значение. Таким образом, введённый символ проверяется на неравенство символу **"f"** (на самом деле сравниваются строки, поскольку с клавиатуры всегда вводятся строки символов, даже если символ введён один). Если введённый символ не является символом **"f"**, то стандартным образом на экран выводится подсчитанный список простых чисел.

В части **else**, однако, имеется новый список операций ввода/вывода, оформленный ключевым словом **do**. Это вполне возможно по причинам, описанным выше. Так что здесь опять происходит запрос у пользователя имени файла, запись в файл результата при помощи стандартной функции **writeFile**, а также уведомления о том, что файл записан. Результат ра-

боты функции **main** может выглядеть, к примеру, следующим образом:

```
Привет. Сколько простых чисел вычислить: 1000
Куда вывести результаты (f – файл): f

Пожалуйста, введите имя файла: primes.txt
Файл "primes.txt" записан.
```

К сожалению, эта функция **main** имеет ряд существенных недостатков. Во-первых, она аварийно завершается, если пользователь введёт на запрос количества простых чисел какую-нибудь строку, которую невозможно преобразовать в целое число. Во-вторых, она также аварийно завершается, если файл, имя которого введено пользователем, невозможно создать или переписать (например, нет прав доступа или у файла имеется атрибут «только чтение»). Ну и, наконец, в-третьих, функция выводит результат на экран в любом случае, если пользователь ввёл символ **"f"**. Хорошо бы, если на экран результаты выводились бы только по нажатию какого-нибудь определённого символа.

Было бы намного интересней, если бы на некорректный ввод во всех трёх случаях программа сообщала об этом и повторно запрашивала информацию. Вдумчивому читателю предлагается в целях тренировки самостоятельно реализовать такую функцию **main**. Это не так сложно, как может показаться на первый взгляд.

3.3. Копирование файлов

Последний пример связан с копированием файлов. Небольшая утилита, которая запрашивает у пользователя имя существующего файла, а также имя нового файла, в который необходимо скопировать содержимое старого файла. Если оба имени корректны, утилита производит копирование. Если что-либо неправильно, происходит какое-либо исключение, то утилита просит пользователя заново ввести все данные.

Для начала необходимо определить функцию, которая скопирует содержимое файла в другой файл, а также сообщит об этом. Применяя знания, полученные в разделе 2, определить такую функцию не составит труда:

```
copyFile :: String -> String -> IO ()
copyFile src dst = do body <- readFile src
                      writeFile dst body
                      putStrLn ("Файл \"" ++ src ++ "\" скопиро-
                      ван в \"" ++ dst ++ "\".")
```

Однако эта функция также не лишена недостатка – она совершит аварийный останов в случае, если с файлами существует какой-то непорядок: нет прав доступа, невозможно прочитать или записать файл. Обработчик исключений должен охватывать эту функцию и «сторожить» её поведение. Поэтому вызов функции **catch** должен происходить в функции **main**:

```
main :: IO ()
main = do putStr "Пожалуйста, введите новое имя файла для копирования:"
          src <- getline
          putStr "Введите новое имя файла:"
          dst <- getLine
          catch (copyFile dst) (\e -> do putStrLn "Ошибка. Пожа-
          луйста, повторите." main)
```

Первые четыре строки в выражении **do** – это запрос имён файлов. Последняя строка наиболее интересна. Здесь и происходит обработка исключений посредством вызова стандартной функции **catch**. Первый операнд этой функции – вызов функции **copyFile** с заданными именами файлов. А второй операнд – обработчик ошибки. В данном случае обработчик написан достаточно просто. Он игнорирует тип ошибки, просто выводит на экран сообщение о том, что произошла какая-то ошибка, и запускает процесс сначала. Рекурсивный вызов функции **main** не должен настораживать – это обычное дело.

И опять же, имеется много возможностей для улучшения этой функции. Можно распознавать тип произошедшей ошибки и выводить на экран соответствующее сообщение. Можно не заставлять вводить пользователя оба имени снова, а просить вводить только то, с которым произошла ошибка. Можно придумать много чего ещё. Все эти новые возможности опять остаются для самостоятельной проработки. Кроме того, вдумчивый читатель увидит здесь одну логическую ошибку, которая нарочито оставлена в целях обучения. Эта ошибка не приводит к генерации исключения, но может быть причиной непредсказуемого поведения программы. Кто первый из читателей сообщит о ней и о возможном способе её преодоления, получит интересный подарок.

Заключение

Данная статья кратко описывает систему ввода/вывода в чистом функциональном языке Haskell. В ней не затрагивается важнейший аспект этой системы – её реализации при помощи *монады,* поскольку эта теоретическая тема достаточно сложна и выходит за рамки простой научно-популярной статьи. Вдумчивый читатель, желающий двигаться дальше, может самостоятельно изучать материалы, посвящённые монадам, поскольку монады являются одной из важнейших систем языка Haskell. В следующих научно-популярных статьях в журнале «Потенциал» тема системы ввода/вывода будет раскрыта дополнительно.

Все приведённые в разделе 3 примеры программ сведены в отдельные модули, которые автор может предоставить каждому, кто пришлёт свой запрос по адресу электронной почты darkus14@gmail. com (при запросе необходимо указывать наименование статьи, для которой необходимо выслать файлы с исходными кодами примеров). Также автор будет признателен любому отклику, который можно также присылать на указный адрес.