Душкин Роман Викторович
Автор нескольких книг по функциональному программированию.

Простой интерпретатор команд

В статье рассматривается одна из интереснейших практических задач, заключающаяся в организации цикла интерпретации команд, вводимых пользователем с клавиатуры. Данная задача повсеместно возникает в интерактивных приложениях, от простых вопрос-ответных вычислительных программ до операционных систем. На примере функционального языка программирования Haskell показывается один способ реализации простого интерпретатора команд.

Введение

Практически в любой прикладной задаче имеется необходимость организации интерактивного взаимодействия с пользователем. Другое дело, что в различных типах приложений такое взаимодействие осуществляется различными способами. Например, в графических приложениях, управляемых операционной системой типа Windows, интерактивное взаимодействие с пользователем осуществляется при помощи графических управляющих элементов.

Особым случаем являются так называемые *консольные приложения*, в которых взаимодействие осуществляется при помощи ввода команд с параметрами и получения реакции системы на них. Во времена до появления графических операционных систем это был естественный способ взаимодействия. Да и сегодня многие прикладные программы всё так же используют консольный ввод/вывод для общения с пользователем.

Вышеописанное показывает, что создание системы функций, которая позволяет организовывать цикл интерпретации и взаимодействовать с пользователем в интерактивном режиме, является типовой задачей, для решения которой можно подготовить так называемый *повторно используемый компонент* в виде некоторого шаблона. Шаблон можно организовать в виде нескольких модулей, часть из которых может быть заменяема в зависимости от назначения интерпретатора. Таким образом, ниже в статье предлагается один из вариантов решения этой задачи и создаётся простой интерпретатор команд.

1. Постановка задачи

Разработка любого программного обеспечения начинается с постановки задачи, написания *технического задания*. Предварительное написание краткого, но проработанного технического задания даже для небольшой задачи является хорошей практикой и способствует быстрому осознанию целей и задач будущей программы, а также не менее быстрой реализации. Поэтому в этом разделе будет кратко сформулирована и формализована задача, которую необходимо решить при разработке простого интерпретатора команд.

Итак, простой интерпретатор команд (далее – ПИК) должен выполнить следующее.

1. При запуске выводить на экран приветственное сообщение, которое может состоять из названия программы и её версии.

2. Постоянно находиться в ожидании команды пользователя, которая вводится при помощи клавиатуры и завершается нажатием клавиши «Ввод».

3. Режим ожидания ПИК характеризуется тем, что на экране присутствует надпись **«Введите команду:»**, а сам ПИК готов к вводу последовательности символов.

4. Любая команда должна состоять из мнемонического имени и произвольного числа её аргументов, отделённых от наименования команды пробелом. Все аргументы также отделяются друг от друга пробелами.

5. Команды могут иметь синонимы, которые полностью заменяют саму команду. Синоним неотличим от команды, нет никакой разницы – использовать команду или её синоним.

6. При вводе неизвестного набора символов ПИК должен сообщать пользователю, что введённая последовательность не является допустимой командой.

7. Не должно иметь значения, в каком регистре символов вводятся команды. Последовательности **«St0rE»** и «**store**» должны быть неразличимы для ПИК.

8. Распознавание команды должно производиться при помощи сравнения с началом последовательности символов, составляющих наименование команды. Если введённая пользователем последовательность является началом какой-либо команды, ПИК выполняет эту команду. Другими словами, последовательность **«sh»** выполнит команду **«show».**

9. Каждая команда должна поддерживать специальный информационный режим, при использовании которого команда не выполняет своё обычное действие, но выводит на экран справочную информацию о самой себе. Параметр, который переводит команду в описанный режим, – **«-?»**.

10. Команда **«exit»** должна позволять выйти из ПИК, при этом на экране должно быть выведено прощание. Возможный синоним команды – **«quit»**.

11. Команда **«help»** должна вы-

водить на экран справочную информацию. Возможные синонимы команды – **«?»**, **«manual»**.

12. Команда **«version»** должна выводить на экран наименование и версию программы.

13. Команда **«store»** должна записывать в таблицу строк заданную после неё строку символов. Пользователю должно разрешаться записывать одинаковые строки в таблицу.

14. Команда **«show»** должна выводить на экран таблицу хранимых строк.

Как видно, эти четырнадцать несложных правил неплохо описывают функциональность простого интерпретатора команд. Набор команд состоит из пяти несложных действий, на примере которых можно легко понять принципы работы подобных интерпретаторов. Особый интерес представляют две последние команды – **«store»** и **«show»**, поскольку они работают с некой таблицей символов, то есть по сути осуществляют некоторые действия, отличные от служебных – выхода из программы, вывода справочной информации и номера версии программы. Так что при дальнейшем рассмотрении этим двум командам необходимо будет уделить особенное внимание.

2. Основной набор функций

Что такое простой интерпретатор команд? Его алгоритм работы на верхнем уровне достаточно прост: вывод запроса пользователю на ввод команды, обработка введённого, выполнение команды, вывод результатов её работы (и выполнение иного эффекта) и возврат в начало этого алгоритма. Получается, что *в цикле интерпретации* необходимо получить команду от пользователя, выполнить её и вывести на экран результаты её выполнения; после чего всё начинается с начала.

Поэтому для реализации цикла интерпретации необходимо просто описать этот алгоритм на языке программирования. К тому же сам алгоритм весьма декларативен, поэтому его реализация на таком языке как Haskell будет простой. Однако перед определением функции, которая реализует описанный алгоритм, необходимо создать некоторые вспомогательные определения.

2.1. Вспомогательные типы данных

Сначала необходимо определить типы данных, с которыми будут работать функции, отвечающие за организацию цикла интерпретации. Прежде всего, это – тип *окружения* интерпретатора, в котором хранится информация о записываемых пользователем строках. По-

скольку кроме строк ничего в окружении храниться не будет, его тип можно определить в виде синонима:

```
type Environment = [String]
```

Одно значение этого типа будет создаваться в самом начале запуска программы и передаваться из функции в функцию для обеспечения того, что в любой момент времени произвольная функция знала бы состав окружения (даже в случае, если в самой функции это окружение не используется). Эта технология является обычной в функциональном программировании, поскольку глобальные переменные запрещены парадигмой (сторонние эффекты недопустимы, а каждая функция должна быть детерминированной). Само собой, что некоторые функции могут изменять окружение, некоторые могут им вообще не пользоваться, но в качестве параметра такое окружение будет передаваться везде, где оно было бы необходимо. Ниже при реализации суть этого будет ясна.

Второй тип данных, необходимый для работы интерпретатора, – код результата выполнения команд. Этот код необходим для того, чтобы в функции цикла интерпретации обрабатывать результат выполнения команды должным образом. Обычно в качестве множества значений такого кода используется два элемента – код выхода из программы и код успешного выполнения операции. Иногда добавляется код неуспешного выполнения операции, чтобы в цикле интерпретации выводить сообщение об этом в стандартный поток вывода сообщений об ошибках **`cerr`**. Также часто в это множество добавляют два кода – код необходимости записи окружения в файл и код необходимости чтения окружения из файла. Это делается для того, чтобы избавиться от необходимости использования системы ввода/вывода внутри функций, выполняющих команды, а сам ввод/вывод производился бы только на верхнем уровне в функции организации цикла интерпретации.

Итак, в случае простого интерпретатора команд такой тип может выглядеть следующим образом:

```
data ICode
  = IC_Exit
  | IC_Error
  | IC_HelpMessage
  | IC_Success
  deriving Eq
```

Дополнительный конструктор **`IC_HelpMessage`** добавлен для различения того, что в качестве результата команда вернула справочную информацию. Возможно, что это может пригодиться при развитии интерпретатора, хотя в создаваемой версии этот конструктор использоваться не будет.

Наконец, весьма важным типом является тип значения, который возвращается каждой функцией, которая обрабатывает одну команду. В целях единообразия это будет один тип (да и было бы весьма непросто сделать обработку различных типов, отдельных для каждой команды, хотя и по такому пути пойти, в принципе,

возможно). Этот тип должен объединять в себе код выполнения операции **ICode**, строку с сообщением о результате выполнения (если команда что-то рассчитывает, то результат расчётов можно привести в строковый вид), а также новое значение окружения, которое могло поменяться в результате действия команды. Итого:

```
data IResult
  = IResult
    {
      code        :: ICode,
      message     :: String,
      environment :: Environment
    }
```

Всё готово для реализации функции, организующей цикл интерпретации.

2.2. Цикл интерпретации

Итак, как указано в самом начале этого раздела, цикл интерпретации заключается в выполнении одной команды, которая, возможно, модифицирует окружение. Это значит, что функция, реализующая цикл интерпретации, должна получать на вход значение типа **Environment**. Вроде бы ничего дополнительного ей для работы не требуется. Возвращает же эта функция пустое значение **IO ()**. Собственно, реализация самой функции достаточно проста:

```
runICycle :: Environment -> IO ()
runICycle env
  = do putStr "Введите команду: "
       cmd <- getLine
       result <- interprete cmd env
       case (code result) of
         IC_Exit -> putStrLn "Всего доброго."
         _       -> do putStrLn  (message result ++ "\n")
                       runICycle $ environment result
```

Как обычно, перевод декларативного алгоритма на некоторый функциональный язык программирования производится чуть ли не слово в слово. В списке операций ввода/вывода выражения **do** (первого уровня) имеется ровно четыре операции, ровно как в описанном в начале алгоритме. Первая строка с функцией **putStr** выводит на экран приглашение к вводу команды. Вторая строка с функцией **getLine** считывает с клавиатуры строку символов. Четвёртая строка с оператором **case** проверяет код результата выполнения команды, и если этот код сигнализирует о необходимости выхода из цикла интерпретации и программы (первая альтернатива), то происходит вывод на экран прощания. В противном случае, если код результата какой-то иной, происходит вывод на экран сообщения, которое вернула выполненная команда, а также запуск той же самой функции **run ICycle** с новым значением окружения.

Наиболее интересной строкой в этом списке операций ввода/вывода является третья строка, в которой получается результат выполнения команды **cmd** при текущем значении окружения **env**. Функция **interprete**, которую ещё предстоит реализовать, возвращает в виде образца **result**, который имеет тип **IResult**, как раз результат выполнения команды. Как видно из её применения, она получает на вход два параметра типов **String** и **Environment**, а возвращает результат типа **IO IResult**. Тело функции опять же не представляет ничего сложного:

```
interprete :: String -> Environment -> IO  IResult
interprete cmd env
  = if (cmd == "")
      then return $ IResult IC_Success "Пожалуйста, введите
      команду."  env
      else let (c:args) = words cmd
                command = findCommand c commands
           in case command of
                 Nothing  -> return $ IResult IC_Error "Ошибка.
                 Неизвестная команда."  env
                 Just fnc -> return $ fnc (unwords args) env
```

Прежде всего, необходимо проверить, не является ли введённая пользователем команда пустой строкой. Если это так, то пользователю необходимо сообщить, что он должен ввести команду. Это делается при помощи операции в части **then** условия. Метод **return** «оборачивает» переданное ему значение в тип-контейнер **IC**. Во второй альтернативе, если введённая команда не является пустой строкой, она разбивается на «слова» (по пробельным символам) при помощи стандартной функции **words**. Эта функция получает на вход строку, а возвращает список строк. Таким образом, первым элементом этого списка является наименование команды, а все последующие – её аргументы. *Замыкание* **command** создаётся при помощи функции **findCommand**, которая ещё будет реализована. Эта функция возвращает значение, «обёрнутое» в тип **Maybe** – этот тип всегда используется в тех случаях, когда после выполнения функции может не оказаться какого-либо результата. В данном случае получение значения **Nothing** свидетельствует о том, что функция для выполнения введённой команды не была найдена, то есть команда является неизвестной простому интерпретатору.

Соответственно, если функция **findCommand** находит требуемую команду и возвращает её в конструкторе **Just**, то эту команду необходимо выполнить, передав ей на вход список аргументов (в виде строки) и текущее значение окружения (ведь команда может работать с окружением). Но среди чего функция **findCommand** ищет функции, которые реализуют действия команд? Как видно из её применения, вторым аргументом ей на вход передаётся некоторая функция **commands** (то, что это внешняя функция, указывает то, что она нигде не определена в функции **interprete** в виде замыкания или образца). Эта константная функция будет написана ниже – она просто возвращает список соответствий имён команд (строк) функциям. Её

реализация будет показана в следующем разделе. Ну а функция **findCommand** выглядит не сложнее других, определённых ранее:

```
findCommand :: String -> [(String, Command)] -> Maybe Command
findCommand _   []           = Nothing
findCommand ñmd ((n, c):cs) = if ((map toLower cmd) 'isPrefix0f' n)
                                 then Just c
                                 else findCommand cmd cs
```

При определении этой функции используются две функции, входящие в модули стандартной поставки библиотек языка Haskell. Функция **toLower** переводит заданный символ в нижний регистр, а функция **isPrefix0f** возвращает значение **True**, если первый заданный список является началом второго (или они совпадают). Поэтому применение условия **((map toLower cmd) 'isPrefix0f' n)** как раз и удовлетворяет требованиям к простому интерпретатору команд – команда должна распознаваться в любом регистре, а само распознавание должно вестись по начальным символам. Для того чтобы использовать эти функции, необходимо произвести подключение модулей:

```
import Char (toLower)
import List (isPrefix0f)
```

Теперь всё готово для реализации функции **main**. Она состоит из двух строк:

```
main :: IO ()
main = do putStrLn greetings
          runICycle []
```

Функция **greetings** является константной и содержит строку приветствия. Эта строка вынесена в константную функцию в целях повторного использования (она ещё пригодится при реализации функций, обеспечивающих выполнение команд). Её определение тривиально:

```
greetings :: String
greetings = "Простой интерпретатор команд. Версия 2007."
```

3. Функции для исполнения команд

Теперь пришло время реализовать все функции, которые выполняют заявленные в требованиях команды. Но перед этим необходимо записать список таких функций и их отображение на различные команды в специальной константной функции **commands**, которая вызывается из функции **findCommand**. Из сигнатуры этой функции также ясно, что типом константной функции **commands** должен быть список пар [(**String**, **Command**)]. Остаётся неясным, что такое за тип **Command**. Это просто синоним функционального типа, созданный для удобства:

```
type Command = String -> Environment -> IResult
```

Этот тип должна иметь каждая функция, которая исполняет команду, введённую пользователем. Как видно, каждая такая функция принимает на вход строку аргументов и окружение, а возвращает обычный результат выполнения команды, описываемый типом **IResult**. Так что теперь можно создавать список команд:

```
commands :: [(String, Command)]
commands = [("?",       doHelp),
            ("exit",    doExit),
            ("help",    doHelp),
            ("manual",  doHelp),
            ("quit",    doExit),
            ("show",    doShow),
            ("store",   doStore),
            ("version", doVersion)]
```

Как видно, этот список позволяет одновременно задавать мнемонические имена для команд и их синонимы. В таком списке можно держать произвольное количество синонимов команд, равно как и задавать новые команды. Поскольку функциональные языки программирования трактуют функции как обыкновенные значения, нет ничего удивительного в том, что сами функции могут выступать в качестве элементов структур данных – кортежей и списков. Так что теперь остаётся реализовать пять функций: **doExit, doHelp, doShow, doStore** и **doVersion.**

Имеет смысл начать рассмотрение функции **doHelp.** Она распечатывает общую справочную информацию, а также информацию о заданной команде. Так что внутри неё имеет смысл вызвать все остальные функции со специальным аргументом **«-?»**, как это заявлено требованиями к простому интерпретатору команд.

Итак, функция **doHelp** выглядит примерно следующим образом:

```
doHelp :: Command
doHelp args env
  = case args of
      ""   -> IResult IC_HelpMessage \
                (greetings ++ "\nВведите \"help <cmd>\" для помощи
                по команде cmd.") env
      "-?" -> IResult IC_HelpMessage "Выводит справочную
      информацию о заданной команде." env
      _    -> let cmd = findCommand args commands
              in case cmd of
                   Nothing  -> IResult IC_Error \
                     ("Ошибка. Нет информации о команде
                     \"" ++ args ++ "\".") env"
                   Just fnc -> fnc "-?" env
```

Как видно, если эта команда введена без аргументов, она выводит общую информацию о программе, в том числе и первоначальное приветствие с версией программы. Если передать этой функции в качестве аргумента заявленную строку **«-?»**, то произойдёт вывод справочной информации о самой команде **help**. Если же ввести какой-либо иной аргумент, то произойдёт поиск команды опять всё в том же списке команд **commands**, и в случае, если команда будет найдена, для получения справки она будет вызвана с аргументом **«-?»**. Весьма изящно. Что будет, если в строке ввода простого интерпретатора команд ввести команду **help help**?

Наиболее простыми являются команды **doExit** и **doVersion**. Их реализация тривиальна:

```
doExit :: Command
doExit "-?" env = IResult IC_HelpMessage "Выходит из простого
интерпретатора команд". env
doExit args env = IResult IC_Exit "" env

doVersion :: Command
doVersion "-?" env = IResult IC_HelpMessage "Выводит информацию
о текущей версии программы". env
doVersion args env = IResult IC_Success greetings env
```

А вот функции **doShow** и **doStore** являются наиболее интересными, поскольку работают с окружением **env**, которое передаётся через все функции простого интерпретатора. Первая команда не модифицирует окружение, но использует его для вывода сообщения на экран. Она немногим более сложная, чем остальные, рассмотренные до этого:

```
doShow :: Command
doShow "-?" env = IResult IC_HelpMessage "Выводит текущее
содержимое таблицы строк." env
doShow args env = IResult IC_Success (showTable env) env
  where
    showTable []     = ""
    showTable (s:[]) = s
    showTable (s:ss) = s ++ "\n" ++ showTable ss
```

Наконец, функция **doStore** модифицирует окружение. Но и она сама по себе не так сложна:

```
doStore :: Command
doStore "-?" env = IResult IC_HelpMessage "Записывает заданную
строку в таблицу строк." env
doStore args env = IResult IC_Success ("Строка \"" ++ args ++
"\" успешно сохранена.") (args:env)
```

Как видно, новая строка просто добавляется в голову имеющегося списка строк, представленного окружением. Это новое окружение возвращается в стандартном результате выполнения команды. Нет ничего проще.

Таким образом, описанным несложным способом можно реализовать произвольные наборы команд. Единственное, на что необходимо обращать внимание, – операции ввода/вывода недопустимы в командах при такой реализации, поскольку тип функций **Command** не имеет внутри себя типа **IO**. В случае, если необходимо сохранять окружение в файл и считывать его из файла, а также совершать иные операции ввода/вывода, необходимо или каким-либо образом указывать на этот факт более высоким функциям (например, функции **runICircle**, в которой обрабатываются результаты команд), либо менять тип **Command**, включая в него возможность работы с системой ввода/вывода. Последний способ целесообразен, когда большая часть команд работает с вводом/выводом.

Заключение

В данной статье, кроме непосредственного изучения системы ввода/вывода языка Haskell и реализации простого интерпретатора команд, также рассматривается интереснейший вопрос передачи некоторого окружения по всему набору функций, использующихся в цикле интерпретации. В рассмотренном примере эта передача повсюду осуществлялась вручную. Как сказано, это сделано нарочито, чтобы не загружать читателя более серьёзными и сложными технологиями.

Дело в том, что такая «сквозная» передача параметра из одной функции в другую является типовой задачей, которая часто встречается в различных программах. Само собой, что для её решения уже имеется готовая технология, основанная на *монаде*, так же, как и сама система ввода/вывода языка Haskell. Данная монада называется **State** – «Состояние». Читатели, заинтересовавшиеся вопросом, могут самостоятельно изучить принципы применения монады **State**.

Все функции и другие программные сущности, определённые при рассмотрении темы этой статьи, сведены в отдельные модули, которые автор может предоставить каждому, кто пришлёт свой запрос по адресу электронной почты darkus.14@gmail.com (при запросе необходимо указывать наименование статьи, для которой необходимо выслать файлы с исходными кодами примеров). Автор также будет признателен любому отклику, который можно присылать на указанный адрес.