**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 3](#_Toc484649512)

[1 ПОСТАНОВКА ЗАДАЧИ 4](#_Toc484649513)

[2 ОБЗОР СУЩЕСТВУЮЩИХ АНАЛОГОВ 5](#_Toc484649514)

[2.1 Cmd.exe 5](#_Toc484649515)

[2.2 PowerShell.exe 5](#_Toc484649516)

[2.3 SygWin 6](#_Toc484649517)

[3 ОБОСНОВАНИЕ ТЕХНИЧЕСКИХ ПРИЕМОВ 7](#_Toc484649518)

[3.1 Обработка ввода пользователя 7](#_Toc484649519)

[3.1.1 Деление ввода на лексемы 7](#_Toc484649520)

[3.2.2 Обработка лексем 7](#_Toc484649521)

[4 ОБОСНОВАНИЕ ВЫБОРА ИНСТРУМЕНТОВ ДЛЯ РЕАЛИЗАЦИИ ПРОЕКТА 8](#_Toc484649522)

[4.1 Язык 8](#_Toc484649523)

[4.2 Среда разработки 8](#_Toc484649524)

[4.3 Иные средства 8](#_Toc484649525)

[4.3.1 Генератор сканеров 8](#_Toc484649526)

[4.3.2 Генератор парсеров 9](#_Toc484649527)

[4.3.3 Текстовый редактор 9](#_Toc484649528)

[4.4.4 Компилятор языка С 9](#_Toc484649529)

[4.4.5 Утилита make 10](#_Toc484649530)

[4.4.6 Сторонние библиотеки 10](#_Toc484649531)

[5 СТРУКТУРА ДАННЫХ 11](#_Toc484649532)

[5.1 Входные данные 11](#_Toc484649533)

[5.2 Промежуточные данные 11](#_Toc484649534)

[5.3 Выходные данные 14](#_Toc484649535)

[6 РАЗРАБОТКА АЛГОРИТМОВ 15](#_Toc484649536)

[6.1 Алгоритм инициализация Shell-оболочки 15](#_Toc484649537)

[6.2 Алгоритм выполнение списка команд 15](#_Toc484649538)

[6.3 Алгоритм обработки ввода пользователя с помощью flex и lemon 15](#_Toc484649539)

[7 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ И РЕЗУЛЬТАТЫ РАБОТЫ ПРОГРАММЫ 16](#_Toc484649540)

[7.1 Сборка продукта 16](#_Toc484649541)

[7.1 Удаление продукта 16](#_Toc484649542)

[7.3 Использование продукта 16](#_Toc484649543)

[7.4 Результаты работы 17](#_Toc484649544)

[ЗАКЛЮЧЕНИЕ 20](#_Toc484649545)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 21](#_Toc484649546)

**ВВЕДЕНИЕ**

Интерпретатор команд операционной системы (ОС), или shell-интерпретатор (от англ. shell «оболочка»), обеспечивает пользователю быстрое и удобное использование системных функций.

Shell-интерпретатор дает возможность оперировать объектами файловой системы на низком уровне. Предоставляет доступ ко всем объектам, хранящимся в различных директориях, т.е. используя интерпретатор пользователь может оперировать, например, файлами, который расположены в различных папках, не меняя местоположения этих файлов.

Shell-интерпретатор дает возможность производить любые настройки ОС и ЭВМ без необходимости отводить взгляд от консольного окна и хаотичного поиска нужного названия среди многочисленных ярлыков и названий.

Так же существует ряд ОС, которые не имеют графического пользовательского интерфейса (ГПИ) с целью сохранения ресурсов ЭВМ, т.к. ГПИ потребляет больше памяти в сравнении с текстовым пользовательским интерфейсом (ТПИ)).

С другой стороны, ГПИ нагляден, а ТПИ требует от пользователя опыта работы в аналогичной среде, иначе работа не будет эффективной и комфортной.

Еще один недостаток ТПИ заключается в том, что в каждой ОС будет свой интерфейс. В связи с этим опытный пользователь Ubuntu, например, зачастую испытывает трудности с работой в cmd.exe (shell-интерпретатор ОС Windows) после многолетнего использования bash (shell-интерпретатор семейства UNIX).

1. **ПОСТАНОВКА ЗАДАЧИ**

Реализовать shell-интерпретатор для ОС Windows.

Программа должна включать в себя:

* выполнение простых команд со списком аргументов;
* выполнение списка команд, связанных “&&” или “||”;
* выполнение команд, потоки ввода-вывода которых связаны каналами (piping);
* перенаправление потоков ввода-вывода (stdin, stdout, stderr);
* выполнение циклов;
* ветвление;
* объявление и использования псевдонимов (aliases);
* удобное редактирование вводимой строки;
* доступ к истории запросов.

1. **ОБЗОР СУЩЕСТВУЮЩИХ АНАЛОГОВ**
   1. **Cmd.exe**

Cmd.exe – стандартный командный интерпретатор, который пользователь ЭВМ сможет использовать на машине с ОС семейства Windows.

Файл сценария Cmd.exe имеет расширение .bat.

Недостатком работы с данным интерпретатором можно назвать отсутствие горячих клавиш копирования текста в буфер и вставки из буфера. Чтобы произвести эту операцию, необходимо использовать мышь.

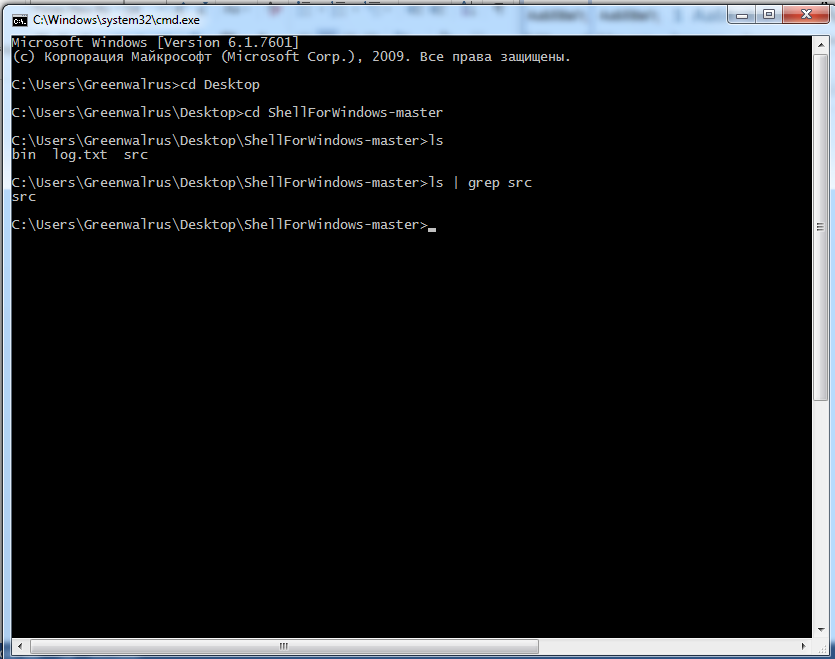


Рисунок 2.2 – Внешний вид Cmd.exe

* 1. **PowerShell.exe**

Windows PowerShell — расширяемое средство автоматизации от Microsoft, состоящее из оболочки с интерфейсом командной строки и сопутствующего языка сценариев. Файл сценария PowerShell имеет расширение .ps1.

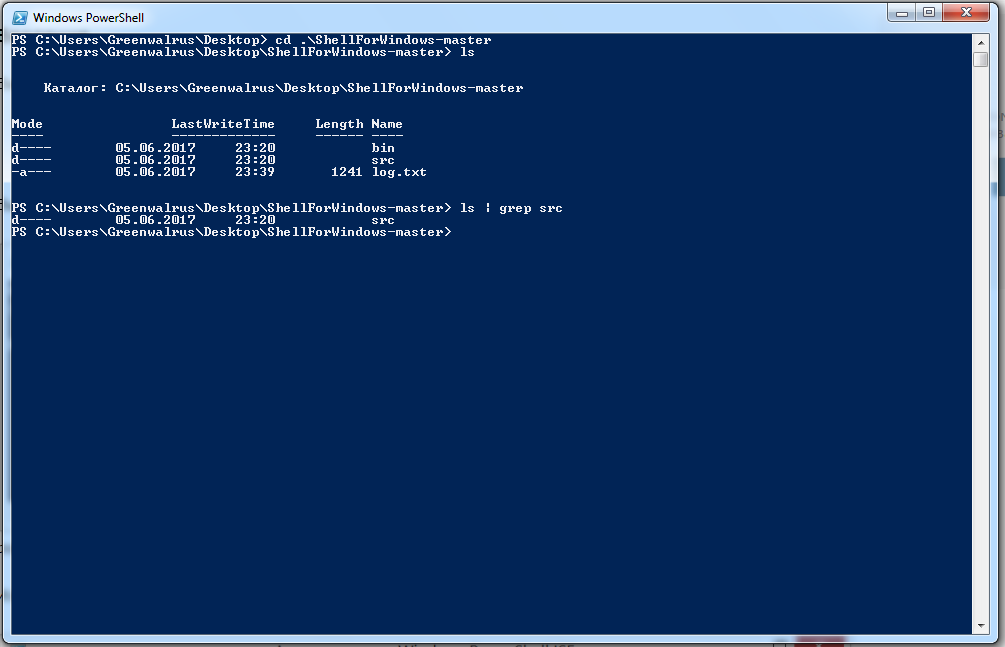


Рисунок 2.2 – Внешний вид PowerShell

* 1. **SygWin**

SygWin позволят перенести интерфейс bash в ОС Windows. Так же предоставляет для использования в среде Windows множество инструментария и библиотек, созданных для Linux-подобных систем. В их число входят: flex, библиотеки readline.h и history.h, компиляторы GNU (gcc).

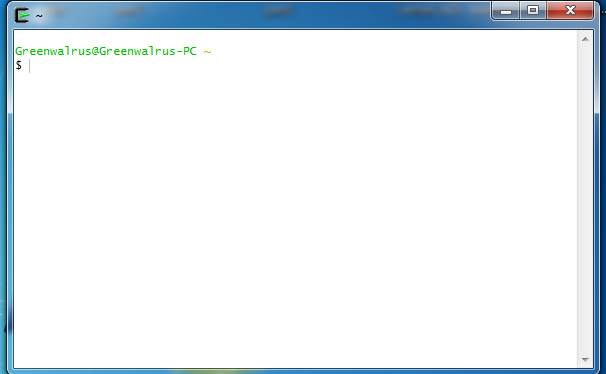


Рисунок 2.3 – Внешний вид CygWin

1. **ОБОСНОВАНИЕ ТЕХНИЧЕСКИХ ПРИЕМОВ**

## 3.1 Обработка ввода пользователя

Перед началом работы было необходимо определиться со способом обработки ввода пользователя.

* + 1. **Деление ввода на лексемы**

Можно делить строку ввода, используя функцию С strtok:

char \*strtok(

char \*strToken, //Обрабатываемая строка

const char \*strDelimit //Разделительный символ

);

Такой способ ограничит пользователя в возможности экранирования пробелов, например, такая комбинация “cd C:/daddys\ folder”, уже не будет обрабатываться правильно. Придется вводить дополнительные проверки. Следовательно, интерфейс такого приложения будет ограниченным, строгим, неудобным.

Более эффективным способом будет работа с регулярными выражениями.

Регулярные выражения дают возможность создавать шаблоны с помощью комбинаций символов и метасимволов. Далее эти шаблонов помогают выделять подходящие подстроки в строке. В этом случае можно обработать абсолютно все комбинации возможного ввода пользователя, не описывая каждую в отдельности.

### 3.2.2 Обработка лексем

Можно, используя функцию strcmp, определять является ли лексема оператором, ключевым словом ветвления или цикла, встроенной командой, объявлением переменной. Если ни один из случаев не подошел – создать процесс с названием, которое хранит данная лексема. И если и в этом случае произошла ошибка – вывести на экран информацию, что пользователь совершил неправильный ввод.

Такая последовательность действий явно неэффективна. Трудно определить какого характера ошибка (синтаксическая, лексическая), значит неточная информация об ошибке будет выведена на экран.

Данный способ приведет к большому количеству ошибок и лишних затрат ресурсов ПК.

Количество комбинаций команд стремится к бесконечности. Следовательно, обработать каждый случай по отдельности невозможно.

Наиболее эффективным будет составление грамматики, которую сможет обработать shell-интерпретатор, т.е. каждому типу лексем задать правила по которым они могут взаимодействовать с другими типами лексем.

1. **ОБОСНОВАНИЕ ВЫБОРА ИНСТРУМЕНТОВ ДЛЯ РЕАЛИЗАЦИИ ПРОЕКТА**
   1. **Язык**

Для создания проекта был выбран язык С. Неотъемлемыми и важными плюсами этого языка являются:

* Скорость выполнения команд;
* Строгая типизация;
* Хорошая документация с примерами;
* Удобный WinApi.

К минусам относятся:

* Неконтролируемость памяти, что требует от разработчика аккуратности и внимательности. Необходимо следить за динамически выделенной памятью и самостоятельно освобождать ее в случаях, когда нет нуды больше в использовании этой памяти.
* Отсутствие возможности обработки исключительных ситуаций.
  1. **Среда разработки**

В качестве IDE была выбрана Microsoft Visual Studio 2010.

Основными критериями стали:

* Доступность данного ПО. VS 2010 можно скачать с официального сайта разработчика бесплатно.
* Уже имелся опыт работы в течение 2ух лет с данным ПО.
  1. **Иные средства**
     1. **Генератор сканеров**

Выбор был между ANTRL, re2c и flex.

Почему выбор был сделан не в пользу ANTRL будет пояснено в п. 4.3.1.

Re2c – быстрый генератор сканеров. Написан на языке С. Большее быстродействие достигается за счет того, что переходы в новые состояния и сдвиги на уровне сгенерированного сканера происходят по меткам, т.е. используется goto.

Выбор пал на flex как проверенный годами генератор сканеров. Flex известен в мире разработчиков в течение нескольких десятков лет.

Наиболее часто употребляется совместно с генератором парсеров bison. Flex и bison – предки пары lex и yacc с исправленными недочетами и неудобствами.

К плюсам можно отнести полноценную документацию, большое количество обучающих сайтов и примеров.

* + 1. **Генератор парсеров**

Выбор был между ANTRL, bison, lemon.

ANTRL генерирует сканер и парсер.

Большим плюсом ANTRL является наличие визуальных сред разработки (ANTLR Works, ANTLR Studio, плагинов к Eclipse), которые позволяют создавать и отлаживать грамматики. Визуальный подход сильно облегчит работу с разработкой грамматик в больших проектах.

К минусам – для работы с ANTRL используется java-машина, что влечет за собой высокое потребление ресурсов ЭВМ.

Bison – как кажется на первый взгляд, не существует без flex и наоборот. Но lemon – хорошая альтернатива bison.

Ричард Хипп разработал lemon для последующего написания базы данных SQLight.

При разработке этого генератора парсеров были учтены недостатки bison.

Главные плюсы:

* Потокобезопасность;
* Использование нескольких генераторов в параллельных процессах, без конфликтов имен;
* Наличие деструкторов, т.е. в случае ошибки и раскручивания стека не произойдет утечки памяти, т.к. для каждого нетерминала можно отдельно определить деструктор;
* Чтобы приступить к работе, надо скачать лишь 2 \*.c исходных файла: lemon.c и lempar.с. Скомпилировать lemon.c, чтобы послучить исполняемый файл lemon, составить грамматику в текстовом файле с расширением .yи передать его в исполняемый файл lemon;
* Небольшая доступная документация;
* Наличие примеров с пояснениями на официальном сайте.
  + 1. **Текстовый редактор**

VS 2010 – удобная среда разработки, но не поддерживает использование lemon. Поэтому написание большей части работы было в текстовом редакторе(ТР) Notepad++.

Данное ПО бесплатно и находится в открытом доступе.

Очень удобно то, что данный ТР обеспечивает подсветку синтаксиса языка и автоподстановку, так же есть подсказки для некоторых функций и процедур языка С.

### 4.4.4 Компилятор языка С

Для компиляции исходного кода использовался компилятор gcc.

Плюсы: доступность, хорошая документация.

### 4.4.5 Утилита make

Данная утилита удобна тем, что после каждого изменения не приходится перекомпилировать каждый файл с исходным кодом и пересобирать проект. Следует лишь прописать зависимости в файле Makefile. Это сохраняет большое количество времени при разработке программных проектов на языке С.

### 4.4.6 Сторонние библиотеки

Readline.h и history.h позволяют обеспечить пользователю удобное редактирование ввода и перемещение по тексту ввода, также доступ к истории предыдущих запросов.

1. **СТРУКТУРА ДАННЫХ**
   1. **Входные данные**

Входные данные для программы представлены текстовым файлом “shell\_history.txt”. В этом файле хранится информация с историей ввода команд в программе. При запуске “shell.exe” библиотека “history5.dll” инициализирует список запросов согласно данным, считанным из этого файла.

* 1. **Промежуточные данные**

Генератор парсеров lemon” собирает запрос пользователя в структуры, которые в последствии будут выполнены в основном алгоритме программы. В исходном коде структуры объявлены в файле “bricks.h”.

Далее будут представлены таблицы с кратким описанием полей и пояснением назначения структур.

Таблица 5.2.1 – Структура struct list\_struct

|  |  |  |  |
| --- | --- | --- | --- |
| Название структуры | | Назначение | |
| struct list\_struct | | Структура для хранения списка команд | |
| Тип поля | Название поля | | Назначение |
| int | size | | Хранит количество элементов в списке |
| struct redirection\_struct\* | redirection | | Хранит перенаправления ввода/вывода списка команд |
| struct node\_struct\* | head | | Указатель на первый узел списка |
| struct node\_struct\* | tail | | Указатель на последний узел списка |
| BOOL | excecAtBackGr | | Флаг выполнения команд в фоне (актуально для цикла) |

Таблица 5.2.2 – Структура struct redirection\_struct

|  |  |  |  |
| --- | --- | --- | --- |
| Название структуры | | Назначение | |
| struct redirection\_struct | | Структура для хранения перенаправления ввода/вывода | |
| Тип поля | Название поля | | Назначение |
| char \* | inputFile | | Название входного файла |
| char \* | outputClearFile | | Название выходного файла |
| char \* | errorFile | | Название входного файла ошибок |

Таблица 5.2.3 – Структура struct command\_struct

|  |  |  |  |
| --- | --- | --- | --- |
| Название структуры | | Назначение | |
| struct command\_struct | | Структура для хранения информации о выполняемой команде | |
| Тип поля | Название поля | | Назначение |
| char \* | nameOfCmd | | Название команды |
| char \* | args | | Аргументы, перечисленные через пробел(ы) |

Таблица 5.2.4 – Структура struct variable\_struct

|  |  |  |  |
| --- | --- | --- | --- |
| Название структуры | | Назначение | |
| struct variable\_struct | | Структура для хранения информации о переменной, используемой в программе | |
| Тип поля | Название поля | | Назначение |
| char \* | varName | | Название переменной |
| char \* | varValue | | Значение переменной |

Таблица 5.2.5 – Структура struct for\_cycle\_struct

|  |  |  |
| --- | --- | --- |
| Название структуры | | Назначение |
| struct for\_cycle\_struct | | Структура для хранения информации о цикле |
| Тип поля | Название поля | Назначение |
| char \* | from | Нижняя граница |
| struct list\_struct\* | instractionsToDo | Список исполняемых команд |
| char \* | until | Верхняя граница (пока не) |

Таблица 5.2.6 – Структура struct operate\_at\_variabe\_struct

|  |  |  |
| --- | --- | --- |
| Название структуры | | Назначение |
| struct operate\_at\_variabe\_struct | | Структура для хранения операции, производимой над переменной |
| Тип поля | Название поля | Назначение |
| char \* | name | Название переменной |
| unsigned char | operation | Битовая маска операции, выполняемой над переменной. Возможные значения:   * OPER\_LESS 0x80 (<) * OPER\_GRET 0x00 (>) * OPER\_CMP\_EQU 0x01 (==) * OPER\_CMP\_NOT\_EQU 0x02 (!=) |
| char \* | value | Значение, с которым сравнивается |

Таблица 5.2.7 – Структура struct whole\_command\_struct

|  |  |  |  |
| --- | --- | --- | --- |
| Название структуры | | | Назначение |
| struct whole\_command\_struct | | | Структура, определяющая тип исполняемой команды |
| Тип поля | | Название поля | Назначение |
| union … cmd | struct command\_struct \* | command | Команда |
| struct operate\_at\_variabe\_struct \* | operate\_at\_variable | Операция над переменной |
| struct for\_cycle\_struct \* | for\_cycle | Цикл |
| struct if\_branch\_struct \* | if\_branch | Ветвление |
| char \* | | name | Название структуры, заполняющей union cmd в данный момент. Возможные значения:   * FOR\_NAME "for"; * CMD\_NAME "command"; * OPER\_AT\_VAR\_NAME "operate\_at\_var"; * IF\_NAME "if"; |
| unsigned char | | connectionWithNextBitMask | Тип связи со следующей коммандой в списке. Воможные значения:   * CONNECT\_NO 0x00; * CONNECT\_PIPE 0x01; * CONNECT\_SEMICOLON 0x02; * CONNECT\_ANDAND 0x04; * CONNECT\_OROR 0x08; |

Таблица 5.2.8 – Структура struct node\_struct

|  |  |  |
| --- | --- | --- |
| Название структуры | | Назначение |
| struct node\_struct | | Структура для узла списка |
| Тип поля | Название поля | Назначение |
| struct operate\_at\_variabe\_struct \* | conditional | Условие для проверки |
| struct list\_struct \* | trueWay | Список команд, который будет выполнен в случае удовлетворения условия |
| struct list\_struct \* | falseWay | Список команд, который будет выполнен в случае неудовлетворения условия |

Таблица 5.2.9 – Структура struct if\_branch\_struct

|  |  |  |
| --- | --- | --- |
| Название структуры | | Назначение |
| struct if\_branch\_struct | | Структура для хранения операции, производимой над переменной |
| Тип поля | Название поля | Назначение |
| struct whole\_command\_struct \* | toDo | Непосредственно информация, хранимая в узле |
| struct node\_struct \* | next | Следующий узел в списке |
| struct node\_struct \* | prev | Предыдущий узел в списке |

* 1. **Выходные данные**

Выходные данные – текстовый файл “shell\_history.txt”, с добавленными в него командами, которые пользователь использовал в текущей сессии.

1. **РАЗРАБОТКА АЛГОРИТМОВ**

## 6.1 Алгоритм инициализация Shell-оболочки

Схема алгоритма представлена на чертеже ГУИР.400201.222 А1 в Приложении А.

## 6.2 Алгоритм выполнение списка команд

Схема алгоритма представлена на чертеже ГУИР.400201.222 A2 в Приложении А.

## 6.3 Алгоритм обработки ввода пользователя с помощью flex и lemon

Схема алгоритма представлена на чертеже ГУИР.400201.222 А3 в Приложении А.

# 7 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ И РЕЗУЛЬТАТЫ РАБОТЫ ПРОГРАММЫ

## 7.1 Сборка продукта

Для сборки проекта необходимо, чтобы были установлены flex, утилита make и компилятор gcc, также путь к этим инструментам находился в PATH переменных среды.

Этапы сборки:

1. Перейти в командной строке в папку ShellForWindows/src.

2) Ввести make.

3) В папке ShellForWindows появится папка bin. Перейти в нее.

4) Запустить исполняемый файл shell.exe.

## 7.1 Удаление продукта

Для корректного удаления продукта без потерь данных при последующей установке следует:

1. Перейти в командной строке в папку ShellForWindows/src.
2. Ввести make clean

## 7.3 Использование продукта

Для начала использования продукта следует ввести команду ‘help’. На экран будет выведен интерфейс работы с программой.

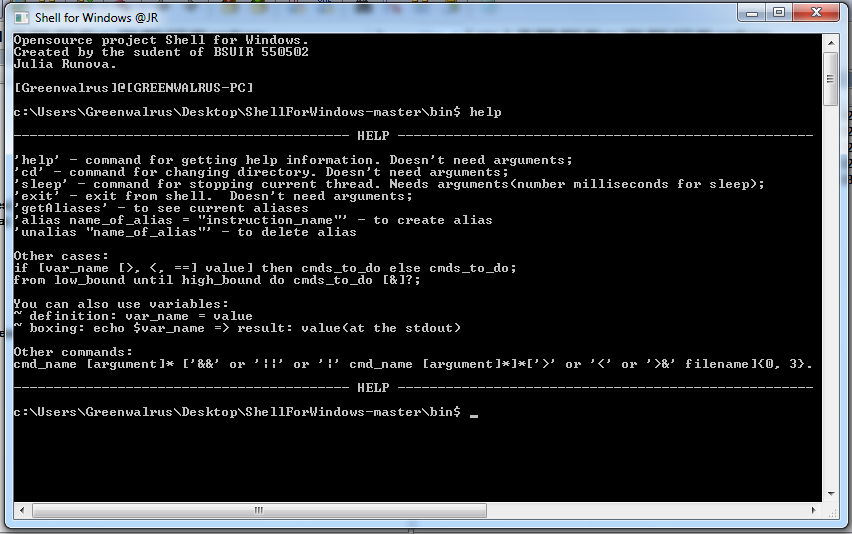


Рисунок 7.3.1 – Результат выполнения команды ‘help’

Для выполнения запроса необходимо ввести запрос с клавиатуры и нажать клавишу ‘Enter’.

Библиотека readline.h обеспечивает подстановку содержимого каталога при нажатии клавиши TAB, так же, удобное редактирование ввода с помощью следующих клавиш:

* 1. Ctrl + a — переход в начало строки;
  2. Ctrl + b — переход на 1 символ назад;
  3. Ctrl + d — удаляет символ под курсором (аналог delete) ;
  4. Ctrl + e — переход к концу строки ;
  5. Ctrl + f — переход на 1 символ вперёд ;
  6. Ctrl + k — удаляет всё, до конца строки .

Библиотека history.h дает возможность сохранять историю запросов (Ctrl + p или up).

На уровне сканера интерпретируются названия команд из bash в аналогичные – cmd.exe:

* 1. "free" - "mem"
  2. "info" - "help"
  3. "man" - "help"
  4. "diff" - "fc"
  5. "ls" - "dir"
  6. "grep" - "findstr"
  7. "ps" - "tasklist"
  8. "kill" - "taskkill"

## 7.4 Результаты работы

На рисунке 7.4.1 показано линейное выполнение списка команд.

Оператор ”||” связывает команды, которые выполняются до первой успешно выполненной команды, поэтому в первом запросе не возникнет ошибка при выполнении подставной команды.

Оператор “&&” требует от всех команд в списке успешного выполнения. Поэтому при первом же возникновении ошибки выполнение списка прекращается. Поэтому во время второго запроса была выведена ошибка на экран.

Оператор “|” – перенаправляет поток вывод текущей команды в списке в поток ввода – следующей. Команда “grep” – фильтр. Третий запрос выполнил вывод на экран содержимого текущей директории в виде списка. Но уже в четвертом запросе содержимое – проходит через фильтр “dll” и список стал значительно меньше.

Оператор “>” перенаправляет stdin в файл, название которого указывается после этого оператора.

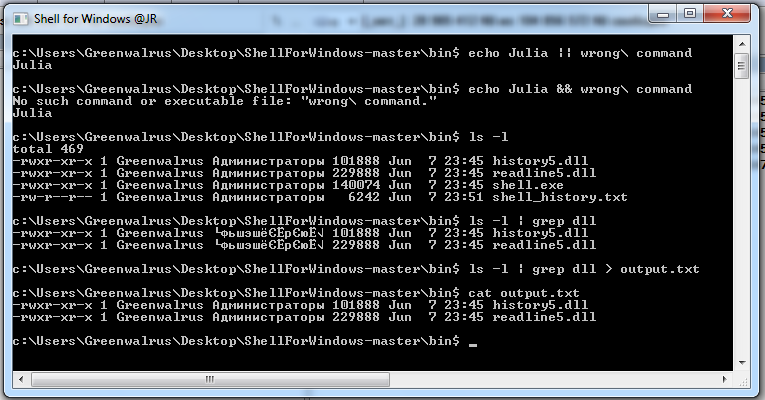
****

Рисунок 7.4.1 – Выполнение команд

На рисунке 7.4.2 представлена работа с циклом, ветвлением, объявление переменных, использование их в условных ветвлениях, подстановка значения переменной с помощью оператора “$”.

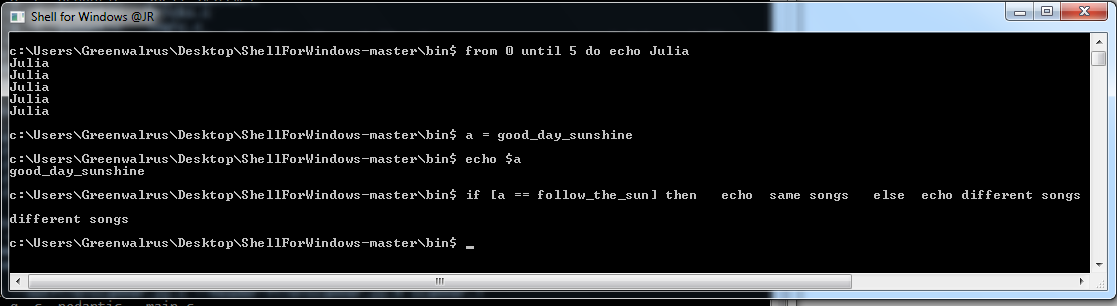


Рисунок 7.4.2 – Не линейный список команд

На рисунке 7.4.3 представлена работа с псевдонимами: объявление, использование, просмотр уже объявленных псевдонимов, удаление.

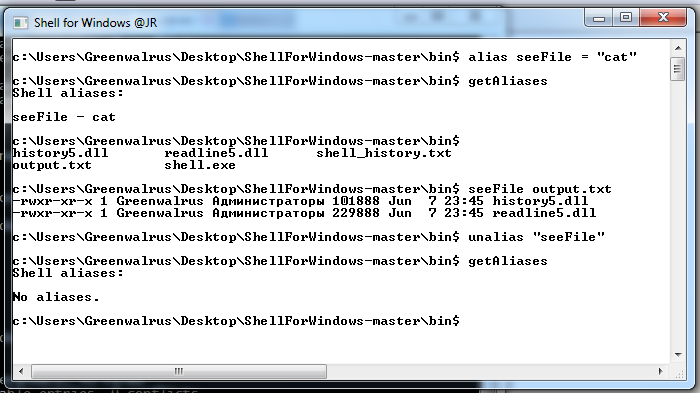


Рисунок 7.4.3 – Работа с псевдонимами

# ЗАКЛЮЧЕНИЕ

Для того, чтобы реализовать shell-интерпретатор надо было изучить основные принципы работы с ТПИ и возможности командных оболочек cmd.exe, PowerShell.

Для создания shell-интерпретатора для Windows было необходимо глубоко изучить WinApi: создания процессов, потоков и их взаимодействие.

Чтобы обработать ввод пользователя были изучены принципы работы с генератором сканеров flex и генератором парсеров lemon. Была разработана грамматика интерпретатора.

Был изучен интерфейс библиотек readline.h и history.h.

Результат работы – shell-интерпретатор для Windows.

Программа будет работать только в среде ОС Windows.

Конечно, данный проект уступает в функциональности стандартным cmd.exe и PowerShell, но главным плюсом является расширяемость проекта. С помощью дополнительного описания грамматики и создания дополнительных структур можно неограниченно расширять возможности продукта.

Shell-интерпретатор имеет некоторые встроенные команды, которые были реализованы мной (cd, sleep, aliases), для выполнения остальных команд вызываются системный утилиты Windows.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

* 1. Документация к библиотеке readline GNU [электронный ресурс] <https://cnswww.cns.cwru.edu/php/chet/readline/rltop.html#TOCDocumentation>
  2. Документация к библиотеке history GNU [электронный ресурс] <https://cnswww.cns.cwru.edu/php/chet/readline/history.html>
  3. Advanced Bash-Scripting Guide [Электронный ресурс] <http://www.opennet.ru/docs/RUS/bash_scripting_guide/>
  4. Command-line reference A-Z [Электронный ресурс] <https://technet.microsoft.com/en-us/library/bb490890.aspx>
  5. Документация С MSDN [Электонный ресурс] <https://msdn.microsoft.com/en-us/library/windows/desktop/ms682425(v=vs.85).aspx>
  6. Документация для flex [Электронный ресурс] <http://flex.sourceforge.net/>
  7. Документация для lemon [Электронный ресурс] <https://www.sqlite.org/src/doc/trunk/doc/lemon.html>
  8. Lemon Paser Generator Tutorial [Электронный ресурс] <http://souptonuts.sourceforge.net/readme_lemon_tutorial.html>
  9. John Levine Flex and Bison - O'Reilly Media; 1 edition (August 24, 2009) – 209 p.