**ACKNOWLEDGEMENT**

Any achievement, be it scholastic or otherwise does not depend solely on the individual efforts but on the guidance, encouragement and cooperation of intellectuals, elders and friends. A number of personalities, in their own capacities have helped me in carrying out this project work. I would like to take this opportunity to thank them all.

I deeply express my sincere gratitude to **Assistant Professor, Srinivas B K** Department of ISE, R.V.C.E, Bengaluru, for his able guidance, regular source of encouragement and assistance throughout this project

I would like to thank **Dr. N. K. Cauvery**, Head of Department, Information Science & Engineering, R.V.C.E, Bengaluru, for her valuable suggestions and expert advice.

I thank my Parents, and all the Faculty members of Department of Information Science & Engineering for their constant support and encouragement.

Last, but not the least, I would like to thank my peers and friends who provided me with valuable suggestions to improve my project.

**Abstract**

In the course of our System Software laboratory, we have implemented a shell. A Shell is one which provides you with an interface which provides you an access to operating system’s services. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output. A shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavours of shells, just as there are different flavours of operating systems. Each flavour of shell has its own set of recognized commands and functions.

**Table of Contents**

# TABLE OF CONTENTS

**Abstract**……………………………………………………………………..2

**List of Tables**………………………………………………………………..5

**List of Figures**……………………………………………………………….5

1. **Introduction**………………………………………………………………..6
   1. Objective……………………………………………………………6
   2. Basic Working of a Shell…………………………………………..7
   3. Simple Commands………………………………………………….7
   4. Background details…………………………………………………7
   5. Input Output Redirection…………………………………………...8
   6. Pipelines and Filters………………………………………………...8
   7. Filename Generation………………………………………………..9
   8. Quoting…………………………………………………………….10
   9. Prompting…………………………………………………………..11
   10. The shell and login.............................................................................11
2. **Requirement Specification Details**………………………………………..12
   1. Hardware Requirements………………………………………….....12
   2. Software Requirements……………………………………………..12
3. **Design**............................................................................................................13
   1. Programming Language…………………………………………….13
   2. Data Flow Diagram……………………………..…………………..14
4. **Implementation**………................................................................................17

4.1Programming Language Selection…………………………………17

* 1. Platform Selection……...........................................................................18

1. **Testing**………………………………………………………………...19
   1. Levels of Testing……………………………………………….19
   2. Unit Testing………………………………………………….....19
   3. Integration Testing……………………………………………...19
   4. System Testing and Acceptance Testing……………………......20
   5. Testing of Main Modules…………………………………….....20
   6. Problems Faced During Testing………………………………...21

6. **Result, Inference & Discussions**………………………………………..22

7. **References**……………………………………………………………….23

**APPENDIX A: Source code for Shell…………….**………………………24

**APPENDIX B: Screenshots of Implementation of Shell**……………...…24

**List of Figures and tables**

# List of Tables

Table 2.1 Table of Hardware Requirements 8

Table 5.1 Working of C program of commands…………………………………………14

Table 5.2 Working of main shell……………………………………………………...….15

**List of Figures**

Fig 3.1 Parts of a DFD 5

Fig 3.2 Level-0 DFD 6

Fig 3.3 Level-1 DFD 7

Fig 3.4 Level-2 DFD 7

**Chapter 1**

**INTRODUCTION**

A UNIX shell is a command-line interpreter or shell that provides a traditional user interface for the UNIX operating system and for Unix-like systems. Users direct the operation of the computer by entering commands as text for a command line interpreter to execute, or by creating text scripts of one or more such commands. Users typically interact with a UNIX shell using a terminal emulator, however, direct operation via serial hardware connections, or networking session, are common for server systems.

The most influential UNIX shells have been the Bourne shell and the C shell. These shells have both been used as the coding base and model for many derivative and work-alike shells with extended feature sets.

The Bourne shell, sh, was written by Stephen Bourne at AT&T as the original UNIX command line interpreter; it introduced the basic features common to all the UNIX shells, including piping, here documents, command substitution, variables, control structures for condition-testing and looping and filename wildcarding. The language, including the use of a reversed keyword to mark the end of a block, was influenced by ALGOL 68.

The C shell, csh, was written by Bill Joy while a graduate student at University of California, Berkeley. The language, including the control structures and the expression grammar, was modelled on C. The C shell also introduced a large number of features for interactive work, including the history and editing mechanisms, aliases, directory stacks, tilde notation, cdpath, job control and path hashing.

**1.1 Objective**

The objective of this project is to implement a shell using the programming constructs of the C language.

The design has to kept as simple as possible so as to facilitate any modification in the design and to incorporate any new feature if possible.

## 

## 1.2 Basic Working of a Shell

When a shell is invoked it reads a special start-up file, typically located user’s home directory that contains some initialization information. It displays a prompt and it waits for a user command. If the user enters a control-D character on a line of its own, this is interpreted by the shell as meaning “end of input” and causes the shell to terminate, otherwise the shell executes the user command given by the user and then again a prompt and it waits for a user command.

### **1.3 Simple Commands**

Simple commands consist of one or more words separated by blanks. The first word is the name of the command to be executed; any remaining words are passed as arguments to the command. For example, who is a command that prints the names of users logged in.

The command ls –l prints a list of files in the current directory. The argument -l tells ls to print status information, size and the creation date for each file.

### **1.4 Background Commands**

To execute a command the shell normally creates a new process and waits for it to finish. A command may be run without waiting for it to finish. For example,

cc pgm.c&

Calls the C compiler to compile the file pgm.c. The trailing & is an operator that instructs the shell not to wait for the command to finish. To help keep track of such a process the shell reports its process number following its creation. A list of currently active processes may be obtained using the ps command.

### **1.5 Input Output Redirection**

Most commands produce output on the standard output that is initially connected to the terminal. This output may be sent to a file by writing, for example, ls -l >file

The notation >file is interpreted by the shell and is not passed as an argument to ls. If file does not exist then the shell creates it; otherwise the original contents of file are replaced with the output from ls. Output may be appended to a file using the notation ls -l >>file.

In this case file is also created if it does not already exist.

The standard input of a command may be taken from a file instead of the terminal by writing, for example,

wc<file

The command wc reads its standard input (in this case redirected from file) and prints the number of characters, words and lines found. If only the number of lines is required then

wc -l <file

Could be used.

### **1.6 Pipelines and filters**

The standard output of one command may be connected to the standard input of another by writing the ‘pipe’ operator, indicated by ‘|’, as in,

ls –l | wc

Two commands connected in this way constitute a pipeline and the overall effect is the same as

ls -l >file; wc<file

Except that no file is used. Instead the two processes are connected by a pipe and are run in parallel. Pipes are unidirectional and synchronization is achieved by halting wc when there is nothing to read and halting ls when the pipe is full.

A filter is a command that reads its standard input, transforms it in some way, and prints the result as output.

One such filter, grep, selects from its input those lines that contain some specified string. For example,

ls | grep old

Prints those lines, if any, of the output from ls that contain the string old. Another useful filter is sort. For example,

who | sort

Will print an alphabetically sorted list of logged in users.

A pipeline may consist of more than two commands, for example,

ls | grep old | wc -l

Prints the number of file names in the current directory containing the string old.

### **1.7 File name generation**

Many commands accept arguments which are file names. For example,

ls -l main.c

Prints information relating to the file main.c.

The shell provides a mechanism for generating a list of file names that match a pattern. For example,

ls -l \*.c

Generates, as arguments to ls, all file names in the current directory that end in .c . The character \* is a pattern that will match any string including the null string. In general patterns are specified as follows.

\* Matches any string of characters including the null string.

? Matches any single character.

[. . .] Matches any one of the characters enclosed.

A pair of characters separated by a minus will match any character lexically between the pair. For example,

[a-z]\*

Matches all names in the current directory beginning with one of the letters a through z.

/usr/fred/test/?

Matches all names in the directory /usr/fred/test that consist of a single character. If no file name is found that matches the pattern then the pattern is passed, unchanged, as an argument.

This mechanism is useful both to save typing and to select names according to some pattern. It may also be used to find files. For example,

echo /usr/fred/\*/core

Finds and prints the names of all core files in sub-directories of /usr/fred . (echo is a standard UNIX command that prints its arguments, separated by blanks.) This last feature can be expensive, requiring a scan of all sub-directories of /usr/fred .

There is one exception to the general rules given for patterns. The character ‘.’ at the start of a file name must be explicitly matched.

echo \*

Will therefore echo all file names in the current directory not beginning with ‘.’.

echo .\*

Will echo all those file names that begin with ‘.’ . This avoids inadvertent matching of the names ‘.’ and ‘..’ which mean ‘the current directory’ and ‘the parent directory’ respectively. (Notice that ls suppresses information for the files ‘.’ and ‘..’ .)

### **1.8 Quoting**

Characters that have a special meaning to the shell, such as <> \*?&, are called metacharacter. Any character preceded by a \ is quoted and loses its special meaning, if any. The \ is elided so that

echo \\?

Will echo a single ? , and

echo \\\\

Will echo a single \ . To allow long strings to be continued over more than one line the sequence \newline is ignored.

‘\’ is convenient for quoting single characters. When more than one character needs quoting the above mechanism is clumsy and error prone. A string of characters may be quoted by enclosing the string between single quotes. For example,

echo xx´\*\*\*\*´xx

Will echo

xx\*\*\*\*xx

The quoted string may not contain a single quote but may contain newlines, which are preserved. This quoting mechanism is the most simple and is recommended for casual use.

### **1.9 Prompting**

When the shell is used from a terminal it will issue a prompt before reading a command. By default this prompt is ‘$’.

It may be changed by saying, for example,

PS1=yesdear

That sets the prompt to be the string yesdear. If a newline is typed and further input is needed then the shell will issue the prompt ‘>’. Sometimes this can be caused by mistyping a quote mark. If it is unexpected then an interrupt (DEL) will return the shell to read another command. This prompt may be changed by saying, for example,

PS2=more

### The shell and login

Following login, the shell is called to read and execute commands typed at the terminal. If the user’s login directory contains the file .profile then it is assumed to contain commands and is read by the shell

Before reading any commands from the terminal.

**1.10 The Shell and login**

Following login, the shell is called to read and execute commands typed at the terminal. If the user’s login directory contains the file .profile then it is assumed to contain commands and is read by the shell.

**Chapter 2**

**REQUIREMENT SPECIFICATION DETAILS**

**2.1 Hardware Requirements**

Before you use the shell, make sure that your computer meets the minimum hardware requirements listed in table below:

**Table of Hardware Requirements for Shell**

| **Items** | **Minimum Value** |
| --- | --- |
| Memory | 5.0 GB |
| Processor Type | 64 bit |
| Processor Speed | 1.83 GHz\*1 |
| Swap Space | 2.1 GB |
| Hard Disk Space | 5.5 GB in /u01  2 GB in /tmp  500 MB in /var  500 MB in /usr |

Table (i)

**2.2 Software Requirements:**

OPERATING SYSTEM: Ubuntu 12.10

COMPILER USED: GCC version 4.3.7

EDITOR : GEDIT

PROGRAMMING LANGUAGE: C

**Chapter 3**

**DESIGN**

The High Level Design Consists of the description of the design method used and data flow diagram of the shell. The design method used consists of the programming language used. In the High level design the technical architect of the project will study the proposed application’s functional and non-functional requirements and design overall solution architecture of the application which can handle those needs. A data flow diagram helps user to visualize how the system will operate and what would be the processing components of the system and what data they would process.

**3.1 Programming Language**

The programming language used for this project is the C language. The C language is one the basic languages and also it has the following benefits.

* It is robust language whose rich setup of built in functions and operator can be used to write any complex program.
* Program written in C are efficient due to several variety of data types and powerful operators.
* The C compiler combines the capabilities of an assembly language with the feature of high level language. Therefore it is well suited for writing both system software and business package.
* There are only 32 keywords; several standard functions are available which can be used for developing program.
* C is portable language; this means that C programs written for one computer system can be run on another system, with little or no modification.
* C language is well suited for structured programming, this requires user to think of a problems in terms of function or modules or block. A collection of these modules make a program debugging and testing easier.
* C language has its ability to extend itself. A c program is basically a collection of functions that are supported by the C library. We can continuously add our own functions to the library with the availability of the large number of functions.

## 3.2 DATA FLOW DIAGRAMS

The DFD (bubble chart) is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data interchange among these functions. Each function is considered as a processing station (process) that consumes some input data and produces some output. The system is represented in terms of the input to the system, various processing carried out on it and the output generated by the system. A DFD model uses a very limited number of primitive symbols to represent the functions performed by the system and the data flow.

**Data flow**

Entity Process Control flow

Fig 3.1. Parts of a DFD

The main reason why the DFD technique is so popular is probably because of the fact that DFD is a very simple formalism. It is simple to understand and use. Starting with a set of high-level functions that a system performs, a DFD model hierarchically represents various sub functions. In fact, any hierarchical model is simple to understand because in a hierarchical model of a system, different details are slowly introduced through different hierarchies. This technique also follows a very simple set of intuitive concepts and rules. DFD is an elegant modeling that turn out to be useful not only to represent the results of the structured analysis of a software problem, but also for several other applications such as showing the flow of items or control in an organization.

**Level-0 DFD**

The level 0 is the initial level data flow diagram and it’s generally called as the context level diagram. It is common practice for a designer to draw a context-level DFD first which shows the interaction between the system and outside entities. This context-level DFD is then exploded to show more detail of the system being modeled.

Output

Commands

**Fig 3.2 Level-0 DFD**

The Level-0 of the DFD is as shown in Fig 3.2. The shell takes commands as input and gives the output.

String Tokens Command

Command

Output

**Fig 3.3 Level-1 DFD**

The Level-1 of the DFD is as shown in Fig 3.3. The shell interface takes command as input and passes it to the program. The Lexical analyzer generates tokens from the given input and passes it to the shell program,which executes the command and generates the output

.

Command

Text Database

Error

**Tokens**

Grammar

Command table

**1.4. 1.4.2 1.4.3**

Command Table

Output

Database

**ST index**

**1.3.3**

**1.4.4**

Error

Fig 3.4 Level-2 DFD

The Level-1 of the DFD is as shown in Fig 3.3. The shell program forks the process and the child process executes the command and the control is returned back to the parent for the next input.

**Chapter 4**

**IMPLEMENTATION**

Implementation of any software is always proceeding by import decisions regarding selection of the platform, the language used, etc. These decisions are often influenced by several factors such as the real environment in which the system works the speed that is required, the security concerns, other implementation of specific details etc.

## 4.1 Programming Language Selection

C is one of the most widely used programming languages of all time, and C compilers are available for the majority of available computer architectures and operating systems.

C has the following advantages which makes it preferable over other languages.

* It is robust language whose rich setup of built in functions and operator can be used to write any complex program.
* Program written in C are efficient due to several variety of data types and powerful operators.
* The C compiler combines the capabilities of an assembly language with the feature of high level language. Therefore it is well suited for writing both system software and business package.
* There are only 32 keywords; several standard functions are available which can be used for developing program.
* C is portable language; this means that C programs written for one computer system can be run on another system, with little or no modification.
* C language is well suited for structured programming, this requires user to think of a problems in terms of function or modules or block. A collection of these modules make a program debugging and testing easier.
* C language has its ability to extend itself. A c program is basically a collection of functions that are supported by the C library. We can continuously add our own functions to the library with the availability of the large number of functions.

## 4.2 Platform Selection

The C language can be run on different platforms. It has to have the right compilers installed. The C codes can be run easily on the Linux or UNIX platform. The Windows platform requires an Integrated Development Environment (IDE) which is easy to use but the speed of compilation is slow. The UNIX compilers are more effective than IDE. They are faster. Another reason for choosing UNIX is that the UNIX shell system calls only can be used for implementing shell commands.

**Chapter 5**

**TESTING**

Testing is an integral part of software development. Testing process, in a way certifies, whether the product, that is developed, complies with the standards, that it was designed to. Testing process involves building of test cases, against which the product has to be tested. In some cases, one drives the test cases from the requirements of the product/software, which is to be developed.

## 5.1 Levels of Testing

Different levels of testing are used in the testing process, each level of testing aims to test different aspects of the system. The basic levels are unit testing, integration testing, system testing and acceptance testing.

## 5.2 Unit Testing

The first level of testing is called unit testing. Unit testing focuses verification effort on the smallest unit of software design module.

In this different modules are tested against the specifications produced during design for the modules. Unit testing is essentially for verification of the code produces during the coding phase,

And hence the goal is to test the internal logic of the modules. It is typically done by the parameter of the module. Due to its close association with coding, the coding phase is frequently called ‘coding and unit testing ‘.The unit test can be conducted in parallel for multiple modules.

## Integration Testing

The second level of testing is called integration testing. Integration testing deals with finding defects in the way individual parts work together. In this, many unit tested modules are combined into subsystems, which are then tested. The goal here is to see if all the modules can be integrated properly.

**5.4 System Testing and Acceptance Testing**

Here the entire software system is tested. The reference document for this process is the requirements document and the goal is to see if the software meets its requirements. Acceptance testing is sometimes performed with realistic data to demonstrate that the software is working satisfactorily. Testing here focuses on the external behavior of the system.

## 5.5 Testing of Main Modules

Unit testing is the process of testing individual components in the system. This is a defect testing process so its goal is to expose faults in these components. Individual functions or methods are the sample type of component and tests are the set of calls to these routines with different parameters. Here different modules are tested independently and their functionality and usability is checked. There are 3 test cases for dictionary creation is tested as shown below Table6.1 to Table .

**Table 5.1 Working of C programs of commands**

|  |  |
| --- | --- |
| Sl. No. of test case : | 1 |
| Name of test : | C program of command |
| Item / Feature being tested : | Unix Command(ls,pwd) |
| Sample Input : | Command |
| Expected output : | Execution of command |
| Actual output : | Execution of command |
| Remarks : | Test succeeded |

Test case 1 as shown in table 6.1 test the working of the C programs which emulate the system commands. A C program for command is written compiled and checked whether it works according to the UNIX command.

**Table 5.2 Working of the main shell**

|  |  |
| --- | --- |
| Sl. No. of test case : | 3 |
| Name of test : | Shell Working |
| Item / Feature being tested : | Main Shell Program |
| Sample Input : | Command |
| Expected output : | Execution of command and shell prompt in next line waiting for next command. |
| Actual output : | Execution of command and shell prompt in next line waiting for next command. |
| Remarks : | Test successful. |

Test case 3 as shown in table 6.3 tests whether the main program works properly. The command executes which shows that the child process works properly. The shell prompt waiting for the next command shows that control returns back to the parent which gives the shell prompt. This shows fork works properly. All these show that the main program works properly.

**5.6 Problems faced during Implementation**

Implementation describes the important decisions taken for the development of the project. The main problem that was faced during implementation was in giving commands such as mkdir and cd to the main shell program. The use of files was tried but it didn’t give fruitful results.

**Chapter 6**

**INFERENCE**

This project provides us great learning experience and great knowledge about the working of the unix shell.

With this project, we came to know a lot, not only about the working of a UNIX shell, but also about the working of the lexical analyzer and syntax analyzer.

We came to know that using a lexical analyzer we can split any given input into different groups, each having a different meaning which is very important for verification of syntaxes.

We got more knowledge about the syntax analyzer, how actually it gets the tokens and checks the grammar for the tokens.

With this knowledge one can develop a complete shell, by getting the executables for all the commands and storing them in a directory. Whenever the input is given to the shell program it checks this directory and executes the respective command.

More commands could be executed. For this the executable has to put in that directory and the corresponding grammar should be included in the YACC file.

All the options could be added to the all the commands.

Pipelining could be done for all the commands or set of commands supporting them.

**REFERENCES**

1. Levine, John R, Mason, Tony, and Brown, Doug, Lex&Yacc, O’reilly&Assosciates, Sebastopol Ca.,2008
2. Leland L Beck, System Software: Third Edition, Pearson Education, 1999
3. Aho, Alfred V., Ravi Sethi and Jeffrey D. Ullman [2006]. Compilers, Prinicples, Techniques and Tools. Addison-Wesley, Reading, Massachusetts.
4. Gardner, Jim, Chris Retterath and Eric Gisin [1988]. MKS Lex&Yacc. Mortice Kern Systems Inc., Waterloo, Ontario, Canada
5. Unix® Shell Programming, Third Edition By Stephen G. Kochan, Patrick Wood

**APPENDIX A: Source code for shell**

**#include<stdio.h>**

**#include<sys/wait.h>**

**#include<unistd.h>**

**#include<string.h>**

**#include<stdlib.h>**

**int lsh\_cd(char \*\*args);**

**int lsh\_help(char \*\*args);**

**int lsh\_exit(char \*\*args);**

**int (\*builtin\_func[]) (char \*\*) = {&lsh\_cd,&lsh\_help,&lsh\_exit};**

**char \*builtin\_str[] = {"cd","help","exit"};**

**int lsh\_execute(char \*\*args)**

**{**

**int i;**

**if (args[0] == NULL) {**

**// An empty command was entered.**

**return 1;**

**}**

**for (i = 0; i < lsh\_num\_builtins(); i++) {**

**if (strcmp(args[0], builtin\_str[i]) == 0) {**

**return (\*builtin\_func[i])(args);**

**}**

**}**

**return lsh\_launch(args);**

**}**

**/\* Function Declarations for builtin shell commands: \*/**

**/\* List of builtin commands, followed by their corresponding functions.**

**\*/**

**int lsh\_num\_builtins() {**

**return sizeof(builtin\_str) / sizeof(char \*);**

**}**

**/\* Builtin function implementations.**

**\*/**

**int lsh\_cd(char \*\*args)**

**{**

**if (args[1] == NULL) {**

**fprintf(stderr, "lsh: expected argument to \"cd\"\n");**

**} else {**

**if (chdir(args[1]) != 0) {**

**perror("lsh");**

**}**

**}**

**return 1;**

**}**

**int lsh\_help(char \*\*args)**

**{**

**int i;**

**printf("Stephen Brennan's LSH\n");**

**printf("Type program names and arguments, and hit enter.\n");**

**printf("The following are built in:\n");**

**for (i = 0; i < lsh\_num\_builtins(); i++) {**

**printf(" %s\n", builtin\_str[i]);**

**}**

**printf("Use the man command for information on other programs.\n");**

**return 1;**

**}**

**int lsh\_exit(char \*\*args)**

**{**

**return 0;**

**}**

**int lsh\_launch(char \*\*args)**

**{**

**pid\_t pid, wpid;**

**int status;**

**pid = fork();**

**if (pid == 0) {**

**// Child process**

**if (execvp(args[0], args) == -1) {**

**perror("lsh");**

**}**

**exit(EXIT\_FAILURE);**

**} else if (pid < 0) {**

**// Error forking**

**perror("lsh");**

**} else {**

**// Parent process**

**do {**

**wpid = waitpid(pid, &status, WUNTRACED);**

**} while (!WIFEXITED(status) && !WIFSIGNALED(status));**

**}**

**return 1;**

**}**

**#define LSH\_TOK\_BUFSIZE 64**

**#define LSH\_TOK\_DELIM " \t\r\n\a"**

**char \*\*lsh\_split\_line(char \*line)**

**{**

**int bufsize = LSH\_TOK\_BUFSIZE, position = 0;**

**char \*\*tokens = malloc(bufsize \* sizeof(char\*));**

**char \*token;**

**if (!tokens) {**

**fprintf(stderr, "lsh: allocation error\n");**

**exit(EXIT\_FAILURE);**

**}**

**token = strtok(line, LSH\_TOK\_DELIM);**

**while (token != NULL) {**

**tokens[position] = token;**

**position++;**

**if (position >= bufsize) {**

**bufsize += LSH\_TOK\_BUFSIZE;**

**tokens = realloc(tokens, bufsize \* sizeof(char\*));**

**if (!tokens) {**

**fprintf(stderr, "lsh: allocation error\n");**

**exit(EXIT\_FAILURE);**

**}**

**}**

**token = strtok(NULL, LSH\_TOK\_DELIM);**

**}**

**tokens[position] = NULL;**

**return tokens;**

**}**

**char \*lsh\_read\_line(void)**

**{**

**char \*line = NULL;**

**ssize\_t bufsize = 0; // have getline allocate a buffer for us**

**getline(&line, &bufsize, stdin);**

**return line;**

**}**

**/\*#define LSH\_RL\_BUFSIZE 1024**

**char \*lsh\_read\_line(void)**

**{**

**int bufsize = LSH\_RL\_BUFSIZE;**

**int position = 0;**

**char \*buffer = malloc(sizeof(char) \* bufsize);**

**int c;**

**if (!buffer) {**

**fprintf(stderr, "lsh: allocation error\n");**

**exit(EXIT\_FAILURE);**

**}**

**while (1) {**

**// Read a character**

**c = getchar();**

**// If we hit EOF, replace it with a null character and return.**

**if (c == EOF || c == '\n') {**

**buffer[position] = '\0';**

**return buffer;**

**} else {**

**buffer[position] = c;**

**}**

**position++;**

**// If we have exceeded the buffer, reallocate.**

**if (position >= bufsize) {**

**bufsize += LSH\_RL\_BUFSIZE;**

**buffer = realloc(buffer, bufsize);**

**if (!buffer) {**

**fprintf(stderr, "lsh: allocation error\n");**

**exit(EXIT\_FAILURE);**

**}**

**}**

**}**

**}\*/**

**void lsh\_loop(void)**

**{**

**char \*line;**

**char \*\*args;**

**int status;**

**do {**

**printf("> ");**

**line = lsh\_read\_line();**

**args = lsh\_split\_line(line);**

**status = lsh\_execute(args);**

**free(line);**

**free(args);**

**} while (status);**

**}**

**int main(int argc, char \*\*argv)**

**{**

**// Load config files, if any.**

**// Run command loop.**

**lsh\_loop();**

**// Perform any shutdown/cleanup.**

**return EXIT\_SUCCESS;**

**}**

**APPENDIX B: SCREENSHOTS OF IMPLEMENTATION OF SHELL**

**test.c** *For checking error*

#include<abc.h>

#include<def.h>

#include<xyz.h>

#include<safa.h>

int main(int a,char c)

{

/\* abc and xyz \*/

scanf("%d%d",&abc,&def) // semicolon missing

for(;;)

{

a=b; /\* apurva \*/

}

return -1;

}

**test1.c** *General Program*

#include<stdio.h>

#include<stdlib.h>

#include<maths.h>

#include<limit.h>

#define N 5

#define M 30

#define J 20

int user(int a)

{

a=b;

return 1;

}

#define K 50

int a=30,j=40,z[100];

char user(char b)

{

a=b;

return a;

}

int main(int a,char c)

{

printf("Enter the value %d\n",a[i]);

scanf("%d%d",&abc,&def);

c=d;

if(a==b)

if(k>=0)

{

j=n;

}

while(b<1000)

{

b++

}

for(;;)

{

do

{

m--;

}while(k>=0);

printf("checked all statements\nending for\n");

}

return -1;

}

**test2.c** *For checking use of keywords in strings*

#include<stdio.h>

#include<stdlib.h>

int a=10;

main()

{

printf("hello how are you\n ending for");

for(;;)

{

a=b;

}

int B;

}

**test3.c** *For checking user function declaration*

#include<starabc.h>

#define M 100

int func(int a);

int main(int argc,char \*argv[])

{

a=b;

return -1;

}

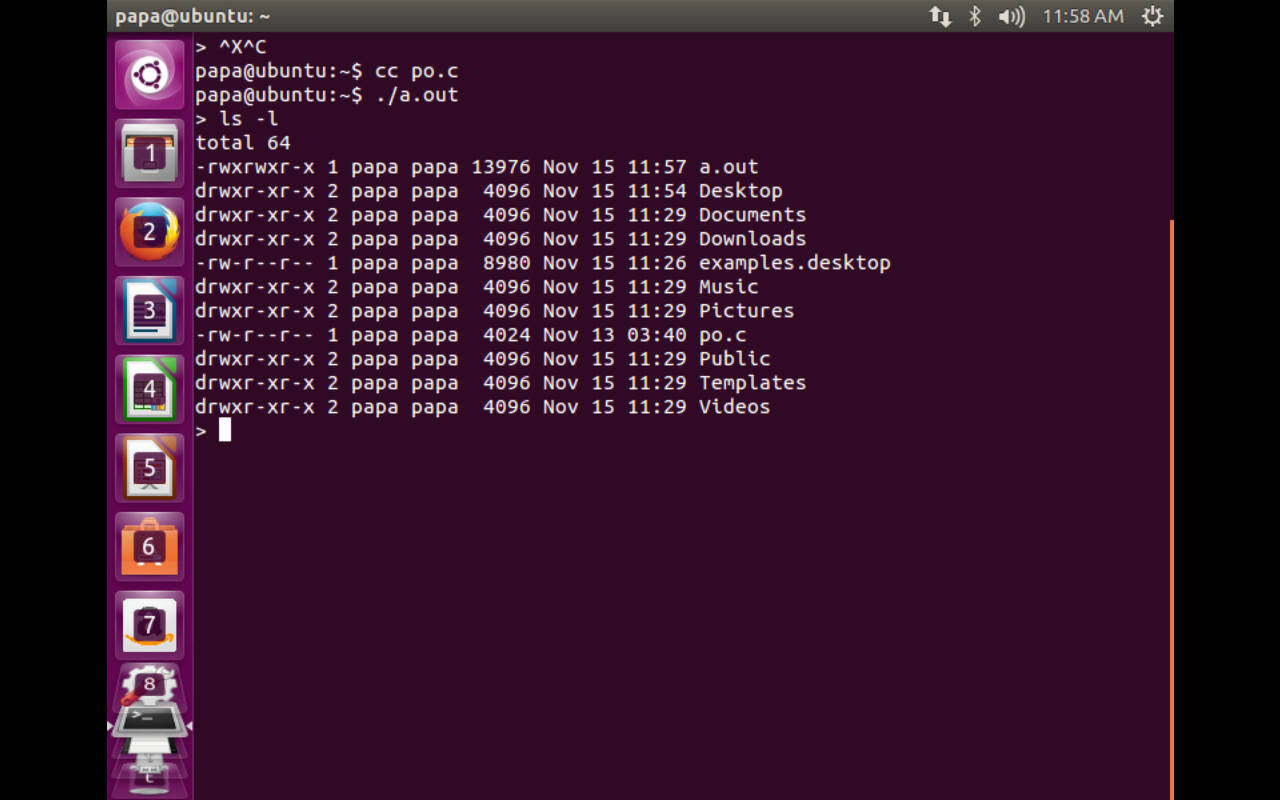
int func(int a)

{

a=b;

return -1;

}

****