UNIX SYSTEM PROGRAMMING LAB

(UE15CS352)

SHELL IMPLEMENTATION

Group members:

Abhuday Vibhanshu (01FB15ECS011)

Aditya Jain (01FB15ECS018)

Aniket Bharati (01FB15ECS036)

Introduction

Shell is a UNIX term for the interactive user interface with an operating system. The shell is the layer of programming that understands and executes commands a user enters. The shell is also called a command line interpreter. The program is an application that allows interacting with the computer. In a shell the user can run programs and also redirect the input to come from a file and output to come from a file. Shells also provide programming constructions such as if, for, while, functions, variables, etc. Additionally, shell programs offer features such as line editing, history, file completion, wildcards, environment variable expansion and programming constructions. In addition to command line shells there are also graphical shells.

The Shell

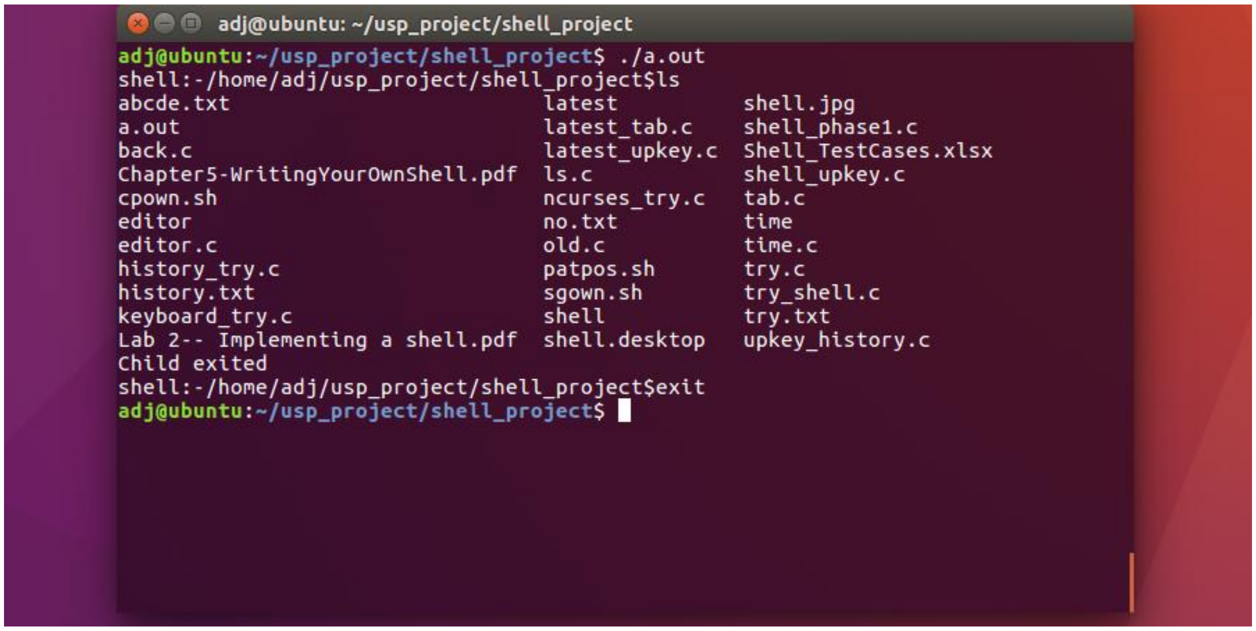
The project has been divided into three phases namely phase 1 phase 2 and phase 3. Phase 1 is implementation of basic shell where we are concerned with the scanning of the commands to read it and then execute it. In phase 2 a number of functionalities have been implemented and in phase 3 custom functions have been implemented such as “sgown” and “ps” command as explained in the next sections.

Phase 1: Implementation of basic shell

The most generic sense of the term shell means any program that users employ to type commands. A shell hides the details of the underlying operating system and manages the technical details of the operating system kernel interface, which is the lowest level. We are concerned with the scanning component of a shell which scans the commands from standard input which could be direct input or in the form of a script and executes them. The shell reads the input, parses it to separate it into commands and arguments and executing those commands. The steps followed for execution of commands by our shell are as follows:

1. Taking input (command) from user.
2. Forking a child process.
3. Calling exec to replace the child process with the input taken in the first step.
4. Waiting for the child process to finish execution.
5. Exiting.

Note: Since we do not have "cd" (change directory) command as executable file in our bin folder so we have manually implemented this using the **chdir** function.



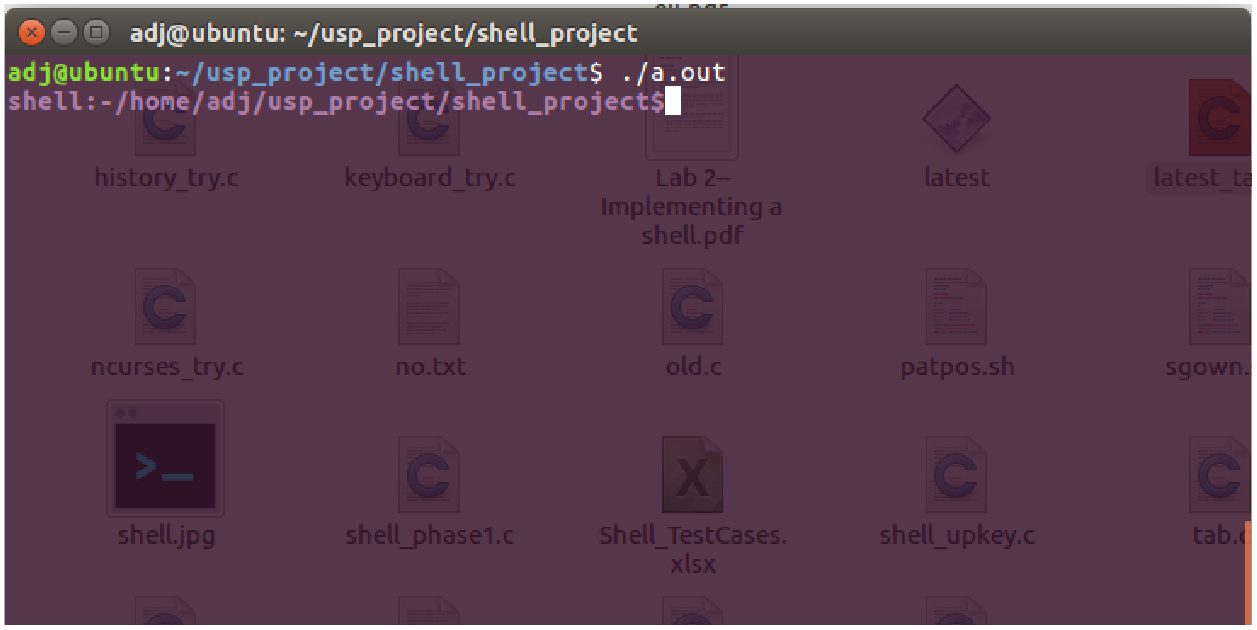
```
adj@ubuntu:~/usp_project/shell_project$ ./a.out
shell:-/home/adj/usp_project/shell_project$ls
abcde.txt                         latest          shell.jpg
a.out                             latest_tab.c    shell_phase1.c
back.c                            latest_upkey.c  Shell_TestCases.xlsx
Chapter5-WritingYourOwnShell.pdf  ls.c            shell_upkey.c
cpown.sh                          ncurses_try.c   tab.c
editor                            no.txt          time
editor.c                          old.c           time.c
history_try.c                     patpos.sh       try.c
history.txt                       sgown.sh        try_shell.c
keyboard_try.c                    shell           try.txt
Lab 2-- Implementing a shell.pdf  shell.desktop   upkey_history.c
Child exited
shell:-/home/adj/usp_project/shell_project$exit
adj@ubuntu:~/usp_project/shell_project$
```

Figure: Screen shot of how our shell looks after everything described is converted into code

Phase 2: Implementation of features

More functionality has been added in the second phase to our basic shell.

- Coloring the prompt: We have used different ansii color shell look more attractive and clean. Below are the definitions which we have used. This contains the color we want to use and its gradient.
    - #define ANSI_COLOR_MAGENTA "\x1b[35m"
    - #define ANSI_COLOR_RESET   "\x1b[0m"
    - #define ANSI_COLOR_BLUE   "\x1b[34m"

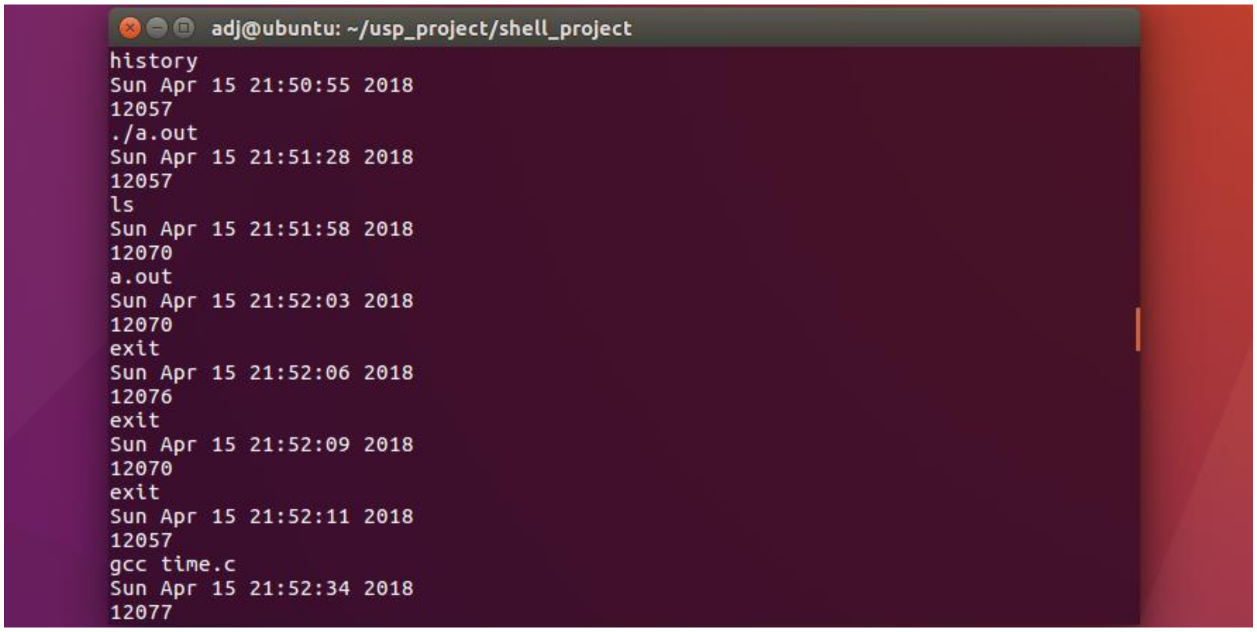


```
adj@ubuntu:~/usp_project/shell_project$ ./a.out
shell:-/home/adj/usp_project/shell_project$
```

Figure: The prompt color changes. Different colors with varying gradients has been used

- Piping and I/O Redirection: Parent process does piping and redirection before forking the processes. The children inherit redirection. The parent saves input/output and restores it at the end. “pipe” function and “dup” system call has been used to implement this

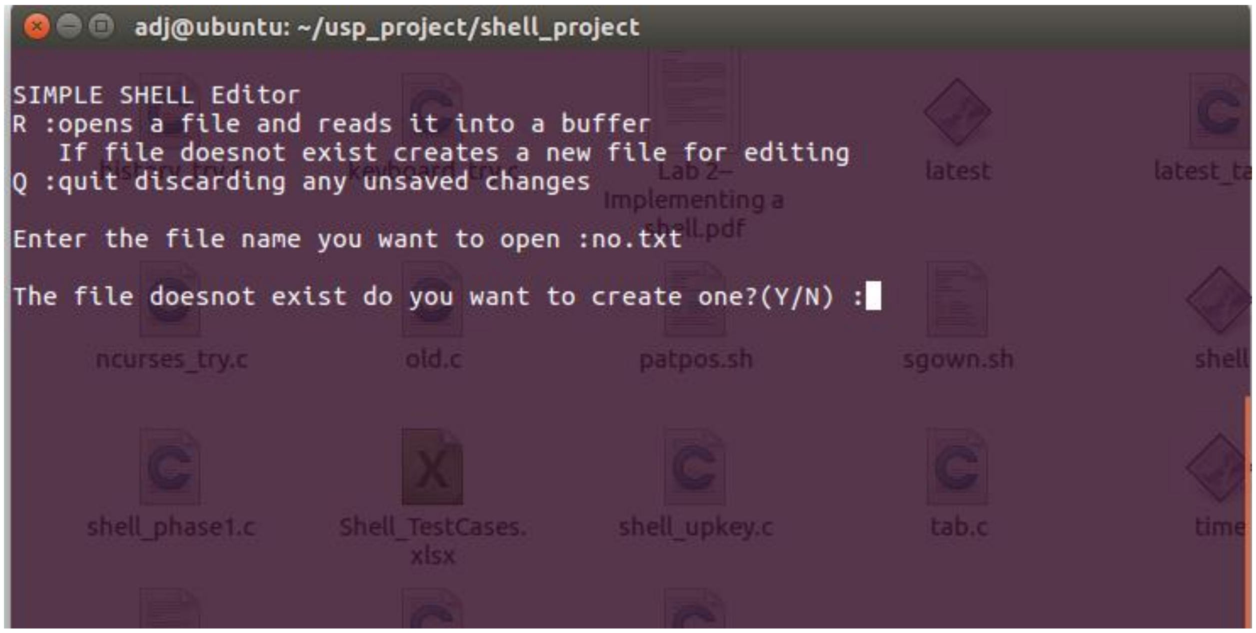
- History: The history() function takes command as argument and add the history content to “history.txt” file from which the read history() function reads. The feature gives all the commands used with their PIDs, date and timestamp.



```
history
Sun Apr 15 21:50:55 2018
12057
./a.out
Sun Apr 15 21:51:28 2018
12057
ls
Sun Apr 15 21:51:58 2018
12070
a.out
Sun Apr 15 21:52:03 2018
12070
exit
Sun Apr 15 21:52:06 2018
12076
exit
Sun Apr 15 21:52:09 2018
12070
exit
Sun Apr 15 21:52:11 2018
12057
gcc time.c
Sun Apr 15 21:52:34 2018
12077
```

Figure: Screen shot of history with their timestamp and PID's

- Editor: A customized editor has been implemented where the user can write his or her code and execute using the shell. It was implemented using different data structures such as linked list and doubly linked list. When this editor opens it prompts the user to specify the name of the file to be edited. If that file does not exist then it asks whether you want to create a new file.

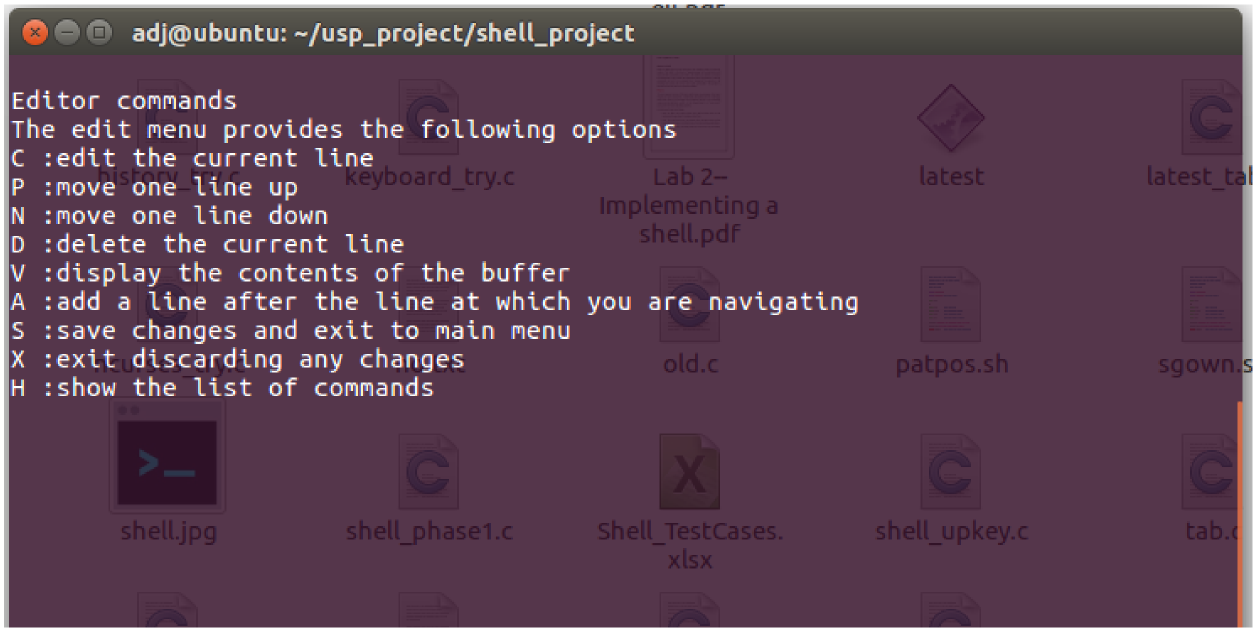


```
SIMPLE SHELL Editor
R :opens a file and reads it into a buffer
   If file doesnot exist creates a new file for editing
Q :quit discarding any unsaved changes

Enter the file name you want to open :no.txt

The file doesnot exist do you want to create one?(Y/N) :
```

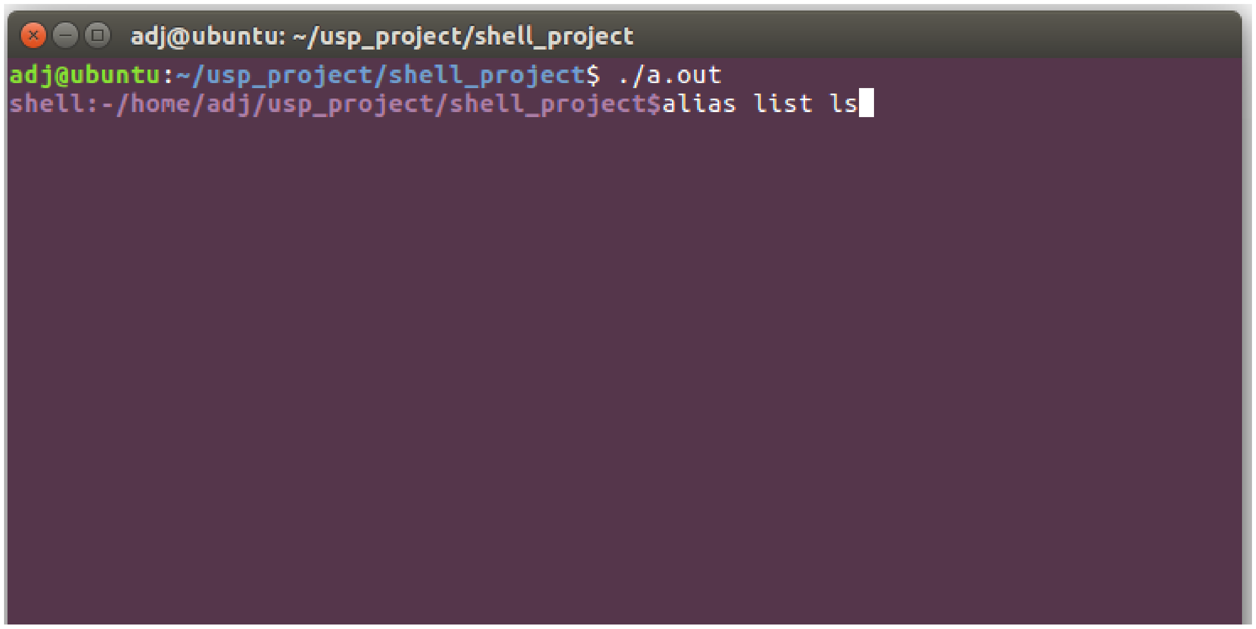
Figure: The shell editor's menu. It prompts if the given file does not exist.



```
Editor commands
The edit menu provides the following options
C :edit the current line
P :move one line up
N :move one line down
D :delete the current line
V :display the contents of the buffer
A :add a line after the line at which you are navigating
S :save changes and exit to main menu
X :exit discarding any changes
H :show the list of commands
```

Figure: The different commands the shell editor supports.

- Aliasing: An alias is a name that the shell translates into another name or command. Aliases allow defining new commands by substituting a string for the first token of a simple command.



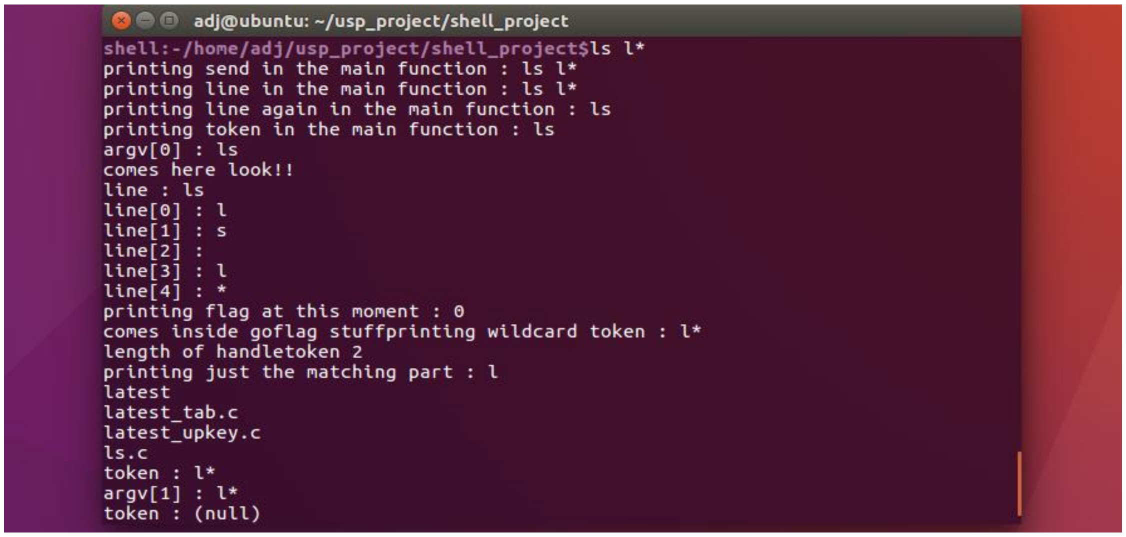
```
adj@ubuntu: ~/usp_project/shell_project
adj@ubuntu:~/usp_project/shell_project$ ./a.out
shell:-/home/adj/usp_project/shell_project$alias list ls
```

Figure: The alias command supported by our shell.

- Wildcards are a set of building blocks that allow you to create a pattern defining a set of files or directories. As you would remember, whenever we refer to a file or directory on the command line we are actually referring to a path. Whenever we refer to a path we may also use wildcards in that path to turn it into a set of files or directories.

  Here is the basic set of wildcards:

  - ***** - represents zero or more characters
  - **?** - represents a single character
  - **[]** - represents a range of characters



```
adj@ubuntu: ~/usp_project/shell_project
shell:-/home/adj/usp_project/shell_project$ls l*
printing send in the main function : ls l*
printing line in the main function : ls l*
printing line again in the main function : ls
printing token in the main function : ls
argv[0] : ls
comes here look!!
line : ls
line[0] : l
line[1] : s
line[2] :
line[3] : l
line[4] : *
printing flag at this moment : 0
comes inside goflag stuffprinting wildcard token : l*
length of handletoken 2
printing just the matching part : l
latest
latest_tab.c
latest_upkey.c
ls.c
token : l*
argv[1] : l*
token : (null)
```

- Tab completion: An extra feature has been added where hitting Tab while typing a command, option or filename will automatically complete what was supposed to be typed. This is done by using **readline** library. This library has function name **rl_bind_key**, which has its arguments as the keyboard key's ascii value and the function triggered when that key is pressed.

Phase 3: Implement custom functions

In phase 3 we have tried to be a little innovative. Firstly we created a desktop icon for our shell. This directly opens our shell when user double clicks on the icon. We have created a .desktop file to implement this. This .desktop file has options to specify the executable file which will be opened when we double click on it. This also has option to give a picture to our icon.

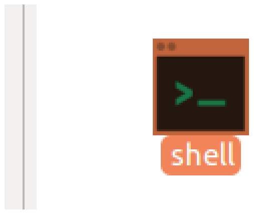
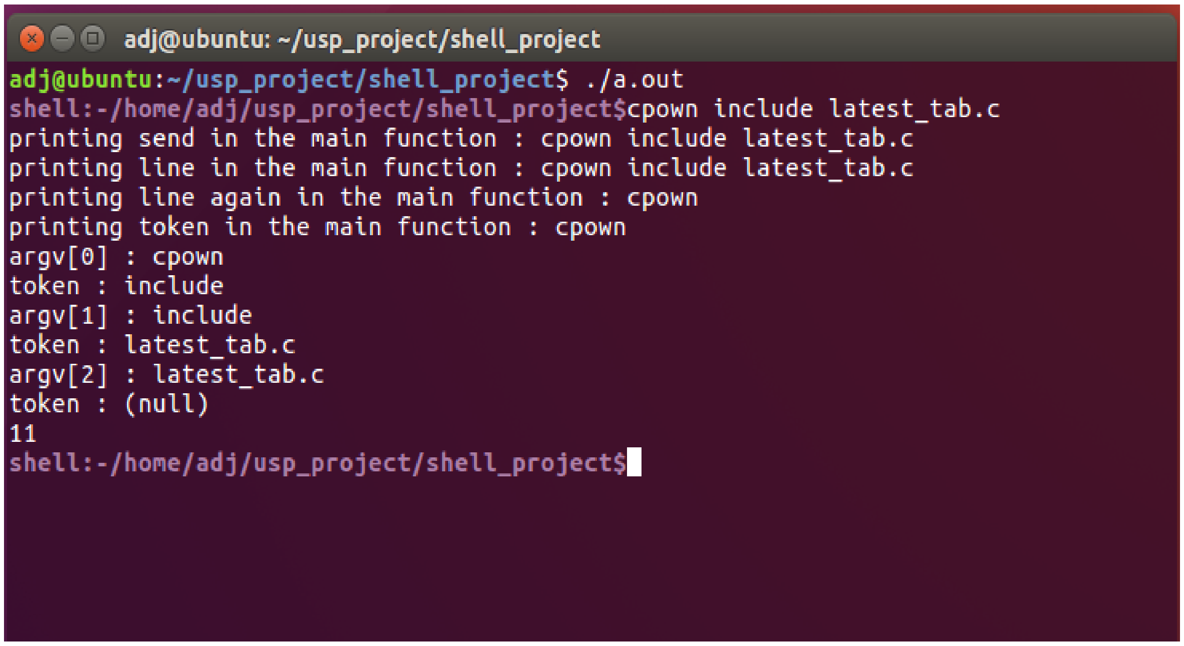


Figure: Shell desktop icon.

Secondly, we have implemented four custom functions. The "sgown" function looks for a pattern string passed as first argument in a file or directory passed as second argument. The "cpown" function counts the number of matches found for a pattern string passed as first argument to a file or folder passed as second argument. The "patpos" function shows the position of a pattern string in the file using line number. The "rmempty" function removes all the empty files which are present in a given directory.

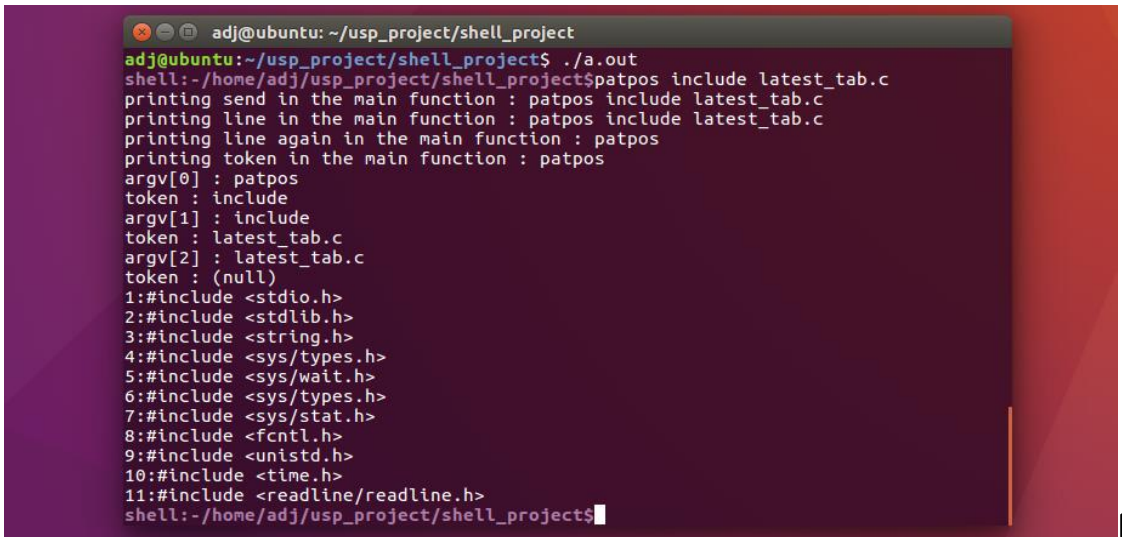
The “cpown” function here counts the pattern string “include” in the file “latest_tab.c”.



```
adj@ubuntu: ~/usp_project/shell_project
adj@ubuntu:~/usp_project/shell_project$ ./a.out
shell:-/home/adj/usp_project/shell_project$cpown include latest_tab.c
printing send in the main function : cpown include latest_tab.c
printing line in the main function : cpown include latest_tab.c
printing line again in the main function : cpown
printing token in the main function : cpown
argv[0] : cpown
token : include
argv[1] : include
token : latest_tab.c
argv[2] : latest_tab.c
token : (null)
11
shell:-/home/adj/usp_project/shell_project$
```

Figure: Picture showing the output of the cpown function

Patpos function showing the line number in for a given word as they occur in the given file. Here It shows how include occurs in a file names latest_tab.c

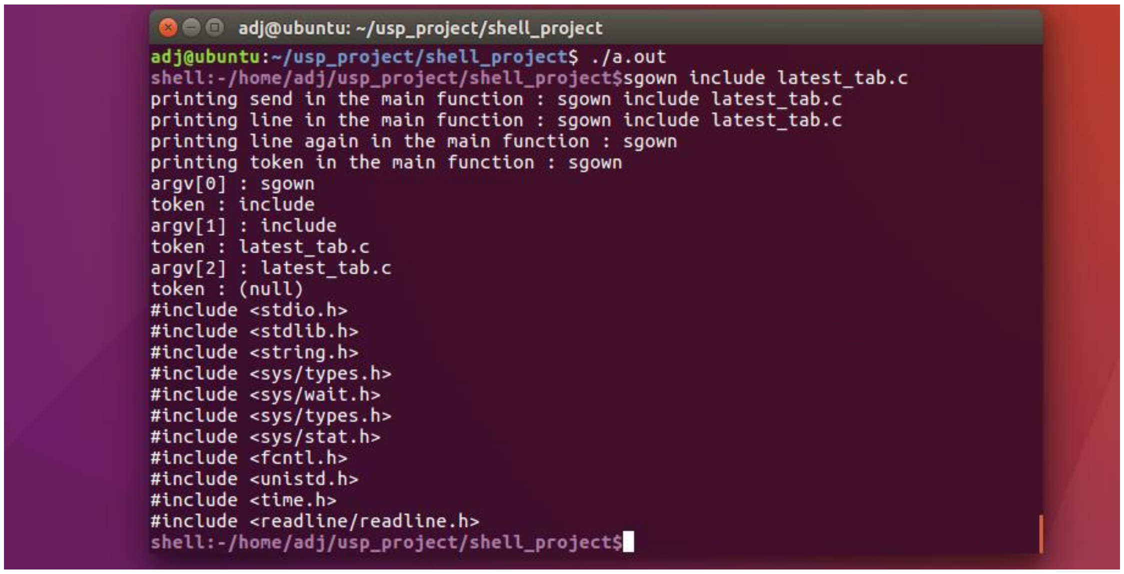


```
adj@ubuntu: ~/usp_project/shell_project
adj@ubuntu:~/usp_project/shell_project$ ./a.out
shell:-/home/adj/usp_project/shell_project$patpos include latest_tab.c
printing send in the main function : patpos include latest_tab.c
printing line in the main function : patpos include latest_tab.c
printing line again in the main function : patpos
printing token in the main function : patpos
argv[0] : patpos
token : include
argv[1] : include
token : latest_tab.c
argv[2] : latest_tab.c
token : (null)
1:#include <stdio.h>
2:#include <stdlib.h>
3:#include <string.h>
4:#include <sys/types.h>
5:#include <sys/wait.h>
6:#include <sys/types.h>
7:#include <sys/stat.h>
8:#include <fcntl.h>
9:#include <unistd.h>
10:#include <time.h>
11:#include <readline/readline.h>
shell:-/home/adj/usp_project/shell_project$
```

Fi

Figure: Picture showing how patpos function output looks like

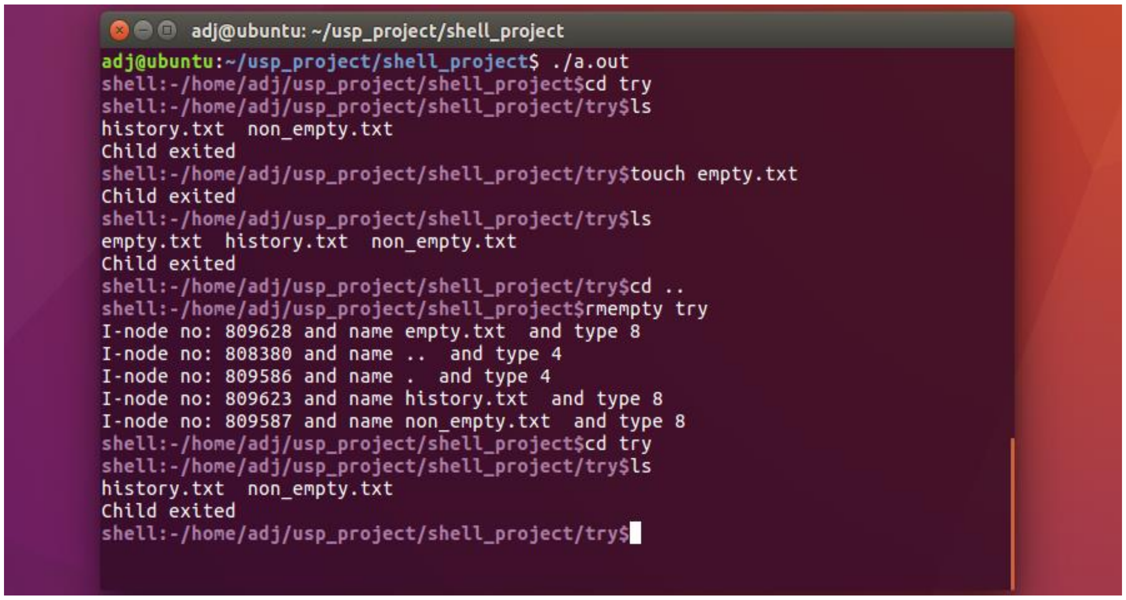
The “sgown” function used here looks for a pattern string passed as first argument in a file or directory passed as second argument. Here the pattern string is “include” and the function is looking in “latest_tab.c”



```
adj@ubuntu: ~/usp_project/shell_project
adj@ubuntu:~/usp_project/shell_project$ ./a.out
shell:-/home/adj/usp_project/shell_project$sgown include latest_tab.c
printing send in the main function : sgown include latest_tab.c
printing line in the main function : sgown include latest_tab.c
printing line again in the main function : sgown
printing token in the main function : sgown
argv[0] : sgown
token : include
argv[1] : include
token : latest_tab.c
argv[2] : latest_tab.c
token : (null)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <readline/readline.h>
shell:-/home/adj/usp_project/shell_project$
```

Figure: Picture showing the output of sgown function.

This function here removes all the empty files present in the given directory. To implement this function, we have used “readdir” function (dirent structure) and stat function. The “readdir” function reads all the files in the directory and using the stat structure it identifies the empty files. To remove the empty files, we have used the remove function.



```
adj@ubuntu: ~/usp_project/shell_project
adj@ubuntu:~/usp_project/shell_project$ ./a.out
shell:-/home/adj/usp_project/shell_project$cd try
shell:-/home/adj/usp_project/shell_project/try$ls
history.txt  non_empty.txt
Child exited
shell:-/home/adj/usp_project/shell_project/try$touch empty.txt
Child exited
shell:-/home/adj/usp_project/shell_project/try$ls
empty.txt  history.txt  non_empty.txt
Child exited
shell:-/home/adj/usp_project/shell_project/try$cd ..
shell:-/home/adj/usp_project/shell_project$rmempty try
I-node no: 809628 and name empty.txt  and type 8
I-node no: 808380 and name ..  and type 4
I-node no: 809586 and name .  and type 4
I-node no: 809623 and name history.txt  and type 8
I-node no: 809587 and name non_empty.txt  and type 8
shell:-/home/adj/usp_project/shell_project$cd try
shell:-/home/adj/usp_project/shell_project/try$ls
history.txt  non_empty.txt
Child exited
shell:-/home/adj/usp_project/shell_project/try$
```

Conclusion

Finally, we have a shell with all features existing explained above. There is also room for adding more features in future as per required. For now, our shell performs most of the basic functions with little extra features.

References

- https://en.wikipedia.org/wiki/Unix_shell
- https://ryanstutorials.net/linuxtutorial/wildcards.php
- http://man7.org/linux/man-pages/man3/readdir.3.html
- https://www.tecmint.com/use-wildcards-to-match-filenames-in-linux/