**Likhon Gomes**

**Documentation**

myShell is a program that is built up with multiple function modules.

**Below I have listed my functions from the code and described their functionality**

**Char readline();**

This function reads input from the keyboard character by character and stops reading it when the user hits the return key or EOF. It takes each character and append them into a string pointer and then returns it.

**Parse(char\* input);**

Parse input function takes the output char array from the readline() function and parse it by strtok() using space as a delimiter. Later outputs of strtok are saved in a char array pointer and returned from this function.

**Quit();**

Quit function sets the loop variable to -1 which is a condition to run the loop endlessly. Setting it to a negative number breaks the loop thus ends the program execution.

**Cd();**

Makes the system call chdir(args). It changes the current directory to the desired directory provided in the argument.

**Clr();**

This function prints out “\033[H\033[2J]” to clear the screen.

**Dir(char\*\* args);**

This function creates a Dir to store the output from the system call opendir() to get the directory. It returns as a struct so function also gets the data from the struct and parses it to print the names of each individual file on the screen. In the end closedir() system call is made to close the dir and clear the memory.

**Environ();**

All of the necessary info requirements are saved into the array info[] which consists 7 elements such as USER, HOMES, PATH, SHELL, OATYPE, PWD, GROUP. A system call getenv() is made by passing in the array info[], it return the desired information and prints it on the screen. If the information are not found, an error message is shown.

**Echo();**

Echo simply prints out the arguments taken at the time executing the command echo in terminal.

**Help();**

This function opens a File from the computer and gets the length of the readme file using fseek so that the array is large enough to hold the file. Uses the ftell to obtain the current value of the file position. In a do while loop the file is read and printed on the screen, It checks for the window size and fetch the information accordingly. During the process the function waits for input from the user, If the user presses return key then the functions shows remaining information (if not shown in one window) otherwise if the user presses return key the function stops and takes the user back to shell.

**makePipe(char\*\* args1, char\*\* args2);**

an array of filedescriptor is passed into the pipe just created in the function. Later these filedescriptors are used to replace the file descriptors of respective programs. A fork is done to create two processes to use the pipe as a bridge for data communication between two processes. In one of the procees dup2 was used to redirect the output file descriptor of one of the programs to the input file descriptor of pipe. On the other process the input file descriptor is changes to the output file descriptor of the pipe. Thuse data is sent from process A to the pipe and then from the pipe to process B.

**halt();**

halt() is just a fancy name to pause. When this function is invoked, it runs a loop while waiting for keyboard input from the user. The loop only breaks when the user inputs return key as an input. Thus the shell pauses for indefinite amount of time until the user presses return key.

**Execute();**

At first the function sets couple of variables to check if the command is going to run in the background, or is it a builtIn function or it’s redirectingIn or redirectingOut.

At first it checks for all the sybols such as &, |, <, >, >> and set the variables accordingly.

Later it checks if it’s an internal command or an execvp command. If it’s an internal command it copies the STDIN and STDOUT file descriptors in case of redirections.

**singleCommandMode();**

singleCommandMode activates the interactive version of the shell where user can use it just like a normal environment instead of passing in a batch file. It starts with clearing the screen using “\033[H\033[2J]” and then starts the master loop which only breaks when loop equals to a negative number. In the loop the function call readline() to read input from the user and then send it to parse() to tokenize the input into arguments. In the end execute the argument using execute function.

**batchExecution();**

This function is invoked when the user chooses to use a batch file as a input method for the shell instead of using the interacting one. In the function a File pointer is opened which hold the batch file as a readable file. Later a while loop goes through the whole file and calls the parse function to parse the batch file just like singleCommandMode and then call the execute function to execute the mentioned commands.

**Main()**

The main function check which path the program should go. Whether the single command mode or batch file execution mode. It depends on the number of arguments passed in. If the argument count is 1 then there was not batch file passed in. Therefore, it’s a singleCommandExecution. Otherwise if the argument count is 2 then there was a batch file passed in along with the execution of shell.

**Testing Methods**

To test the parser I have created a separate code test.c that includes the parser and uses it. I have tested all the other components inside the shell. Below I have proded the screenshot the parser testing.

A screen shot of a computer

Description automatically generated

**Testing of Functionalities**

**cd**A screenshot of a cell phone

Description automatically generated

In this shot I “cd ..” out of the directory and “cd Shell” into Shell directory.

**dir**

A screenshot of a cell phone

Description automatically generated

In this shot I used the “dir” command to print all of the contents of current directory.

**environ**

A screenshot of a cell phone

Description automatically generated

used the “environ” command to show all of the environ variable.

**echo**

A screenshot of a cell phone

Description automatically generated

Used the “echo hello world” to print hello world on the screen.

**help**

A screenshot of a computer screen

Description automatically generated

Used “help” command to view the readme file in shell.

**quit**

A screenshot of a computer

Description automatically generated

Used “quit” command to quit from the shell and return back to default shell.

**pause**

A screenshot of a computer screen

Description automatically generated

Used the pause command to pause the shell for unlimited time until the user presses return key on the keyboard.

**I/O Redirection**

**>A screenshot of a computer screen

Description automatically generated**

used “echo hello > out.txt” to echo the word hello into the file out.txt and then used cat to print out the file.

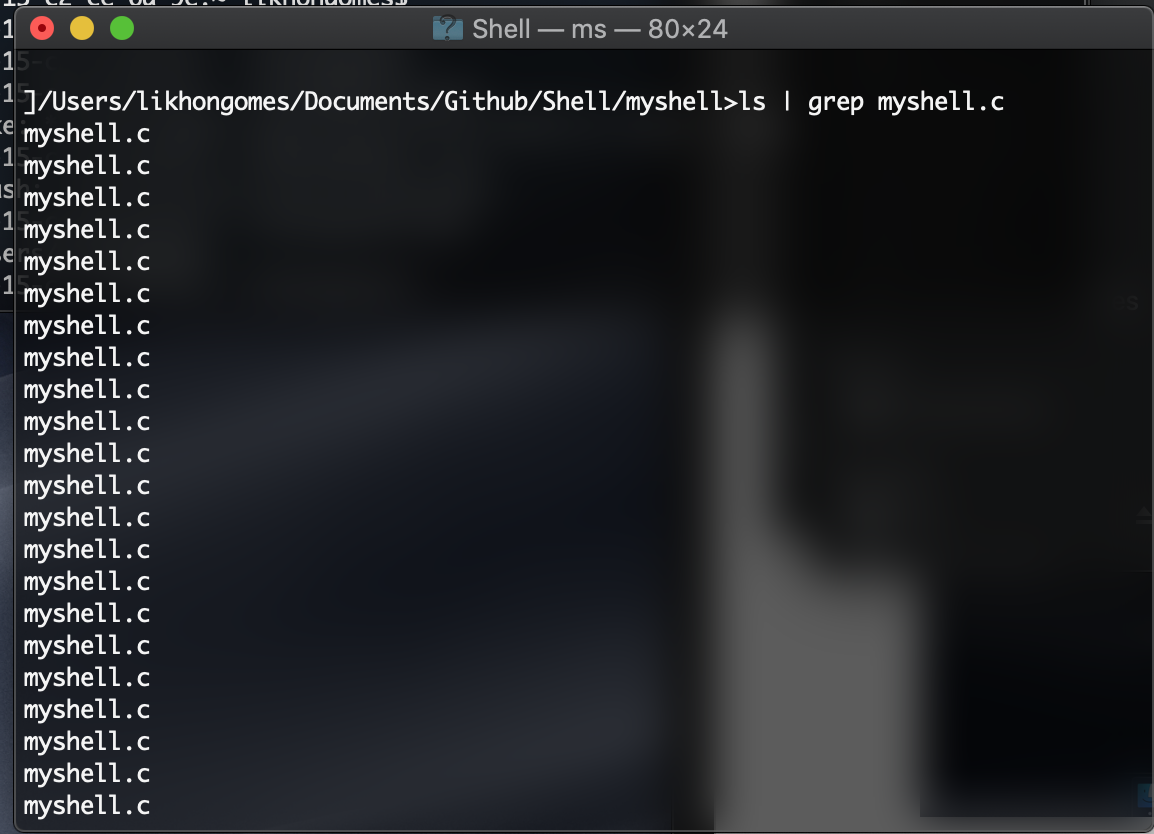
**>>**

**A screenshot of a cell phone

Description automatically generated**

used “echo world >> out.txt” to concatenate the word world with the existing word in out.txt document.

**Piping**

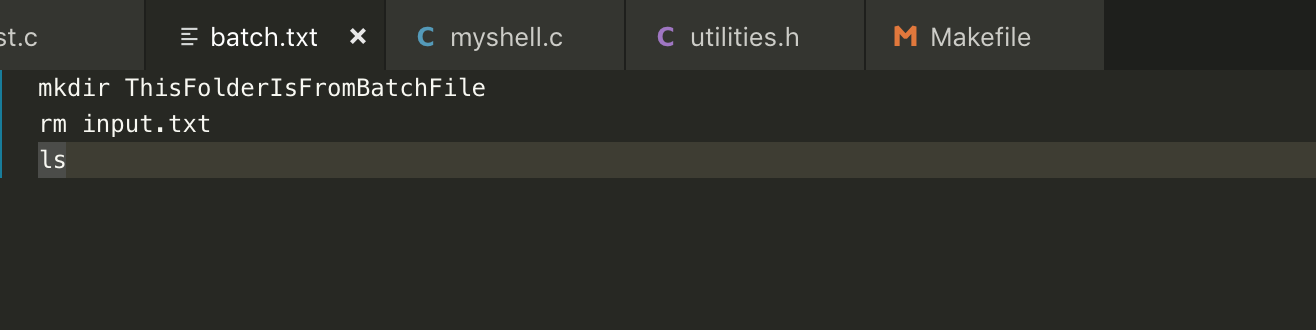
****

Used the command “ls | grep myshell.c” to find the file myshell.c in my current directory, the only problem is it prints out the file multiple times.

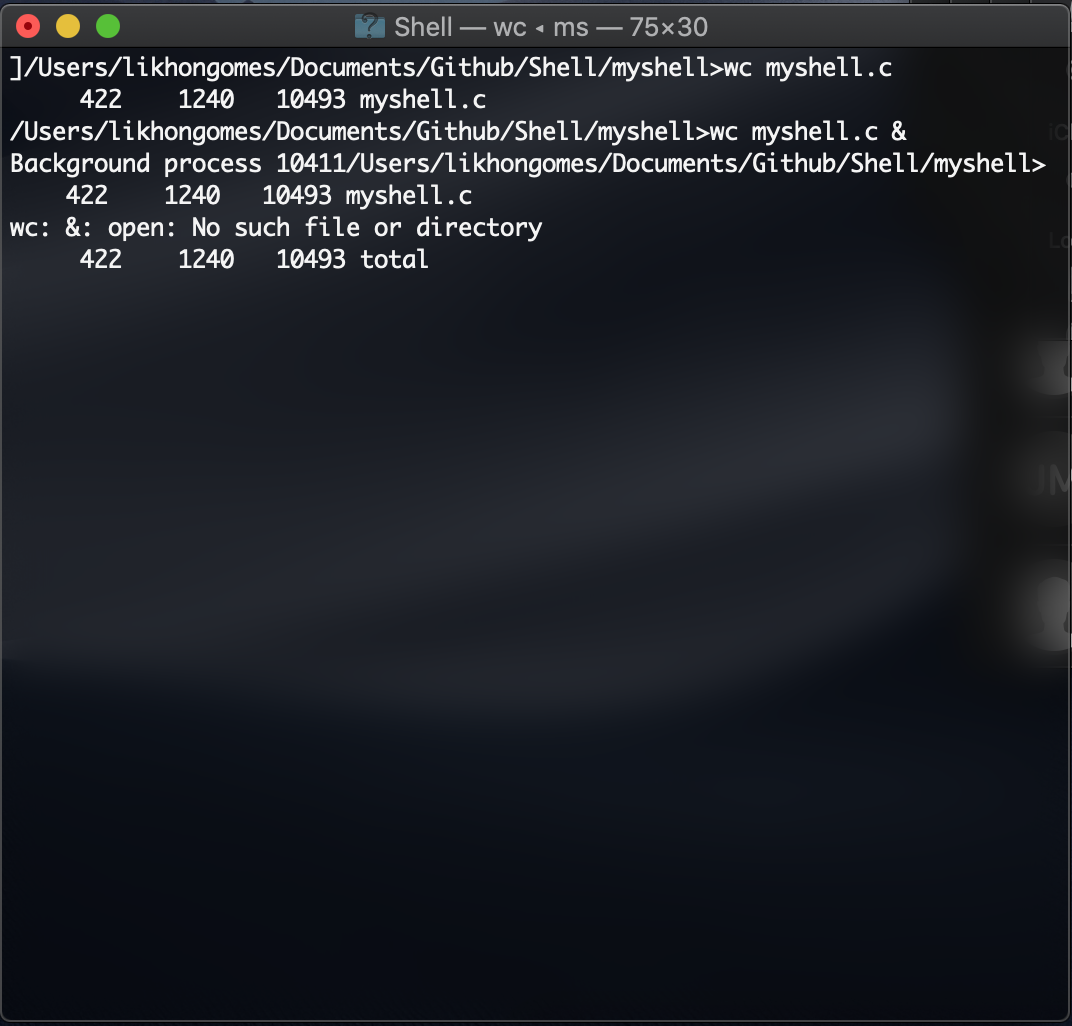
**Batch File Execution**

A screenshot of a cell phone

Description automatically generated

In this one I have executed a batch file, in the above screenshot, using batch.txt I have used mkdir to make a directory, remove a file and do ls. Below I have provided screenshot of batch.txt file

**Background Execution**



Here I have used the ampersand to execute it in the background, despite of that it still shows the result on the screen.