**Program Summary**

The purpose of this program was to simulate the operations of a linux shell as an extension of the second homework assignment. Commands may be provided by the user at run time or else a prompt is displayed and the program waits for the user to input a command. The command is passed into the predefined parse\_command function. In this function, a character pointer iterates over each character in the line argument. When an identifier is found (space, pipe, null, or redirect), the previous characters are copied into a local token variable. The program checks if the token is null or empty to avoid adding spaces as arguments in the case of a space as the first character in the command or multiple spaces in a row. The program then checks the global flag variables to determine where the token should be copied to. If there has not been a pipe or redirect yet, cmd1 dynamically allocates space for the token. If a pipe has occurred but no redirects have, cmd2 allocates space. If a redirect has been found, the program determines which redirect has occurred and copies the token to either outfile or infile. Null terminators are appended to cmd1 and cmd2. The function returns the value of a separate function that checks all the global flags to determine the commands' complexity. If the user enters "quit" as a command, the program exits.

Separate functions were added to handle each of the 8 possible command types. When an input redirection is flagged by the program, the stdin file descriptor is overwritten to the input file descriptor. Likewise, the stdout file descriptor is overwritten by the outfile variable when output redirection is detected. In cases where a pipe is used, the main process of the program makes the pipe system call. Then the main process creates 2 sub processes. The first child process executes the command in the cmd1 variable and possibly reads input from a file if input redirection is used, and the result is fed into the stdout file descriptor. The second child process then reads in the output from the first child process and executes the command in cmd2 and possibly writes the output to a file.

The most difficult part of the assignment was getting pipe execution to work. At first I tried using only 1 child process and having the parent process execute the second command. The difficulty was with handling the correct file descriptors for each child process. The first child process needed to close the stdin file descriptor and make the dup2 system call with the stdout descriptor. The second child process did the opposite. A separate function was created which would write to a log file (foo.txt) to help understand what was happening behind the scenes.

**Program To-Do List**

While this program does fulfill the requirements laid out in the assignment, it does not fully account for all errors. The error handling in this program is limited to checking if sub processes were correctly created. Also, the algorithm of the parse function as well as the limited number of global flags used allows some commands to return as legitimate.

**Test Cases**

Several test cases were used to ensure the program had the correct output. The commands were first entered in to the turing shell and then into the program in order to compare outputs. The test cases were designed such that each possible execute function could be called. Specifically, these commands were entered and had the correct output:

ls

sort < sort\_test.txt

ls -a >> output.txt

sort < sort\_test.txt >> output.txt

ls -a > output.txt

sort < sort\_test.txt > output.txt

ls | wc

sort < sort\_test.txt | wc

ls | wc >> output.txt

sort < sort\_test.txt | wc >> output.txt

ls | wc > output.txt

sort < sort\_test.txt | wc > output.txt