Westley Huebner

Joseph Pinrod

Tyler Clements

11/20/13

**Program 4 Report**

*Disclaimer: Tyler Clements was kicked out of his old group and joined Joseph and I for this program. This program was created using his help*

**The Problem**

Program 4 required us to process files as input, create a Path variable that is created when the program is run, and a few functions to process the input. These functions consists of line extension to allow the user to use multiple lines to input their command, processing multiple commands that are separated by a “?” character, and the echo command that displays the user’s input but still processes a few other commands. Most of these problems deal with the actual input from the user and where we process it, ranging from processing a separate file to individual words.

**Overview of echo and extension**

For line extension and echo, we decided to have separate functions for each, as it would make them easier to work with, while simultaneously cleaning up our code and providing better organization. We made a file, named “processInput.c”, where we are storing all of our functions that are built for manipulating or analyzing the user input from the command line.

**Echo**

For the nshEcho() function, we are combining all of our split inputs into one temporary character array, so we can better process the individual words. We have a simple while loop checking for variable usage (denoted by the ‘@’) and adding the next word to our output variable, which is our temporary character array. After the input from the command line has been processed, it prints out the processed string.

**Line Extension**

For the lineExtension() function, we are first checking for the ‘$’ character to see if the function even needs to be used. If the character exists, a for loop finds the location of it, starting from the back, and once it finds it, holds the location of it in a integer value, “i”. The function then cuts off everything after the ‘$’ and prompts the user for the next line of input. The comment filter is then applied to the new line of input, and the old input is then combined with the new input. After all this, the function calls itself again to check to see if another ‘$’ has been entered, and if it is, proceeds to process the next line on input, and if there is not another line extension, the function returns, ending its recursive calling.

**File Reading**

Our file parsing method takes two arguments, the filename, and whether or not the file was used as one of the command line arguments to the shell. The method is called twice at the beginning of the program. Once for the NSHRC file, and once for the command line argument. The method opens a file, creates an 80 character line array, then uses fgets on the file and parses it like user input. Afterwards, to fix a bug lingering data can cause, the arrays holding data from the lines are zeroed with memset. Once the process is complete, if the second argument ‘s value is 1 or true depending on your preferred term, the integer(cont) that keeps the program going ceases to do so. If no file exists, and error is returned.

**Multiple Commands**

To process multiple commands with the “?” symbol, we process the entire input and split it into 2 different strings. The split occurs at the first occurance of “?”, leaving the first part of the input as a complete command for the program to run. We then pass the first command block to the userInput function, then after that we pass the remaining part of the input to userInput. Since the multiple commands function occurs in userInput, any remaining “?” are processed and the input is parsed again. This keeps occurring until there are no longer anymore commands to process.

Because we are calling userInput within the userInput function, we have to cancel out of the current userInput function if multiple commands are found. To do this, we change the multiple commands function into a Boolean function. If a “?” is found, the function runs and returns 1. If it is not found, the function returns 0. In userInput (where the function is called), we check the return value of the multiple commands function to see whether or not we continue the current userInput function. If a 1 is returned, then we return back to the main loop and exit out of userInput. If a 0 is returned, we continue the function. While it does seem a little unnecessary to perform this check, we do so to ensure the order of commands is kept.