For this assignment, I designed my program to have 1 function per feature. However, there were some features that I was not able to implement such as: redirection of standard error, multiple commands, and multiple piping.

1. Internal shell command exit

   For this feature, I simply put an exit(0) in the if statement detecting whether args[0] = "exit.

2. A command with no arguments

   This feature was implemented together with feature 3. It first creates a fork and checks whether the fork successfully completes. After, if the pid is 0, this means that this is the child and execvp will execute with args[0] and args as its parameters. Else, it is the parent, in which case it will wait for the child to finish.

3. A command with one or more arguments

   See above.

4. A command with or without arguments whose input is redirected from a file.

   First, there is a for loop in main that detects whether a "<" was detected in args. If so, it will set args[i] (the location of the special character) to NULL and the file to args[i+1] and then pass these parameters to a function called inputRedirect. A fork is then done and checked to see whether it successfully completes. If the pid is 0, we first call open to open the file specified by the pathname. If the file does not exist, open will create the file for us. Afterwards, we use dup2 in order to create a copy of our file descriptor, and replace the file descriptor at location 0. We then call close so that the file descriptor closes and won't be used to reference the file and then finally call execvp to execute the command. If the pid is greater than 0, this means that it is the parent which will wait for the child to finish.

5. A command, with or without arguments, whose output is redirected *to* a file.

   First, there is a for loop in main that detects whether a special character was detected in args. If so, it will set args[i] (the location of the special character) to NULL and the file to args[i+1] and then pass these parameters to a function. I split this feature up into two separate functions outputRedirect and ouputRedirectAppend to deal with the two cases of whether the loop detects a ">" or ">>" in the args. It is similar to feature 4, but I used creat(file, 0666) for inputRedirect and open(file, O_RDWR | O_APPEND | O_CREAT, 0666) for inputRedirectAppend. This is because if the args contains ">>" instead of ">", the command's output is appended to the end of the file. creat(file, 0666) would overwrite the text in the file. Both of these commands perform similar functions and open/create the file. The 0666 gives it the permission to write in the file and the flags used in open allow it to read/wrote, append or create. dup2 is then used to create a copy of a file descriptor, but instead of putting it in location 0, we put it at location 1. We then close the file descriptor and then run execvp to execute the command. If the pid is greater than 0, this means that it is the parent which will wait for the child to finish.

6. A command, with or without arguments, whose output is piped to the input of another command.

   The for loop in main checks to see whether "|" is in args and if so, it passes both args and the index of | into pipeCommand. In pipeCommand, args[i] is set to NULL because that is the location of the special characters. Then, two for loops are run to split the args into its separate parts into pipeArgs and pipeArgs2. Afterwards, the pipe is created by calling pipe(pipefd) where pipefd is an array of size 2. The fork is then created, and if pid is 0, this means that it is the child. In the child process, we perform another fork because we are trying to execute multiple commands. If pid2 is 0, this is the child of the child, in which we execute the second part of the command (e.g. the things that come after |). This is done by first calling close(pipefd[1]) to close the write end for the child process. This is because the child is only reading in the output from the parent, and does not need to write. Afterwards, we call dup2(pipefd[0], 0) to copy the file descriptor in pipefd[0]. We then call execvp to execute the second part of the command that is in pipeArgs2. Finally, we call close(pipefd[0]) to fully close the pipe. If pid2 is greater than 0, this means that it is the parent in which we will execute the first part of the command. It is similar to the execution in the child process, but flips the order in which we close because the parent needs to write into the pipe so that the child can read from it.

9. Two or more command sequences, possibly involving I/O redirection, separated by a semicolon.

   This feature was different from the other features since I did not use a fork. This is due to the fact that if we were to perform a fork and to cd in the child process, this wouldn't affect the parent process since the fork makes the child and parent separate processes. The for loop first checks to see if the args contains "cd" and then calls the function changeDirecory and passes args as the parameters. In change directory, we first check to see if args[1][0] is equal to /. If so, we use strcpy to copy the string in args[1] which is the location we want to cd into and put it into a char array called dir. Afterwards, we simply call chdir(dir) to change directory into the path specified in dir. Else, we call getcwd to get the current working directory and put it into a char buffer of SIZE (I defined SIZE to be 256 after looking at the shell.l file). We then put that into dir using strcpy and then using strcat, concatenate "/" and then args[1] into dir to create the path. We then call chdir(dir) to cd into the path specified in dir.