

# OPERATING SYSTEMS AND SYSTEMS PROGRAMMING 1

## EGGSHELL ASSIGNMENT REPORT

Russell Sammut-Bonnici  
CPS1012  
May 14, 2018

## [Contents](#)

### **Introduction to The Eggshell Implementation**

<b>1.1 Implementation Description</b>	<b>3</b>
<b>1.2 Data Flow Diagram</b>	<b>4</b>

### **Variables (vrblController.c)**

<b>2.1 Dynamic Array for Storing Variables</b>	<b>5</b>
<b>2.2 Accessing Variable Values</b>	<b>5</b>
<b>2.3 Adding Variables</b>	<b>5</b>
<b>2.4 Initializing Shell Variables</b>	<b>6</b>
<b>2.5 Testing Shell Variables</b>	<b>7</b>

### **Line Parsing (eggshell.c)**

<b>3.1 The Main Method</b>	<b>8</b>
<b>3.2 The Read-Evaluation-Process-Loop (REPL)</b>	<b>8</b>
<b>Pipeline Construction and Evaluation (pipeController.c)</b>	<b>9</b>
<b>4.1 Checking for Piping by Pipeline Construction</b>	<b>9</b>
<b>4.2 Evaluating the Pipeline</b>	<b>9</b>
<b>4.3 Testing Piping</b>	<b>11</b>

### **Input and Output Redirection (ioController.c)**

<b>5.1 Checking for Input and Output Redirection</b>	<b>13</b>
<b>5.2 Output Redirection</b>	<b>13</b>
<b>5.2 Input Redirection</b>	<b>14</b>
<b>4.3 Testing Redirection</b>	<b>15</b>

## Commands (cmdController.c)

6.1 Checks Whether Command is Internal or External	17
6.2 Internal Command Methods excluded from intrnlCmdParser.c	17

## Internal Commands (intrnlCmdParser.c)

7.1 The Variable Declaration Command	18
7.2 The Print Command	19
7.3 The Chdir Command	21

## External Commands (intrnlCmdParser.c)

8.1 Initializing Environment Variables	26
8.2 Finding the Path to the External Command	26
8.4 Testing External Commands	28

## Signal Handling (sigHandler.c)

9.1 Initializing the Sigaction Struct	29
9.2 The Signal Handler	30
9.3 Resuming Suspended Processes	30

## Conclusion

10.1 Special Features	32
10.2 Possible Future Improvements	32

### Note:

The contents are done in order of data flow, except signal handling as it is heavily reliant on external command processing, therefore it is placed after the section on external commands.

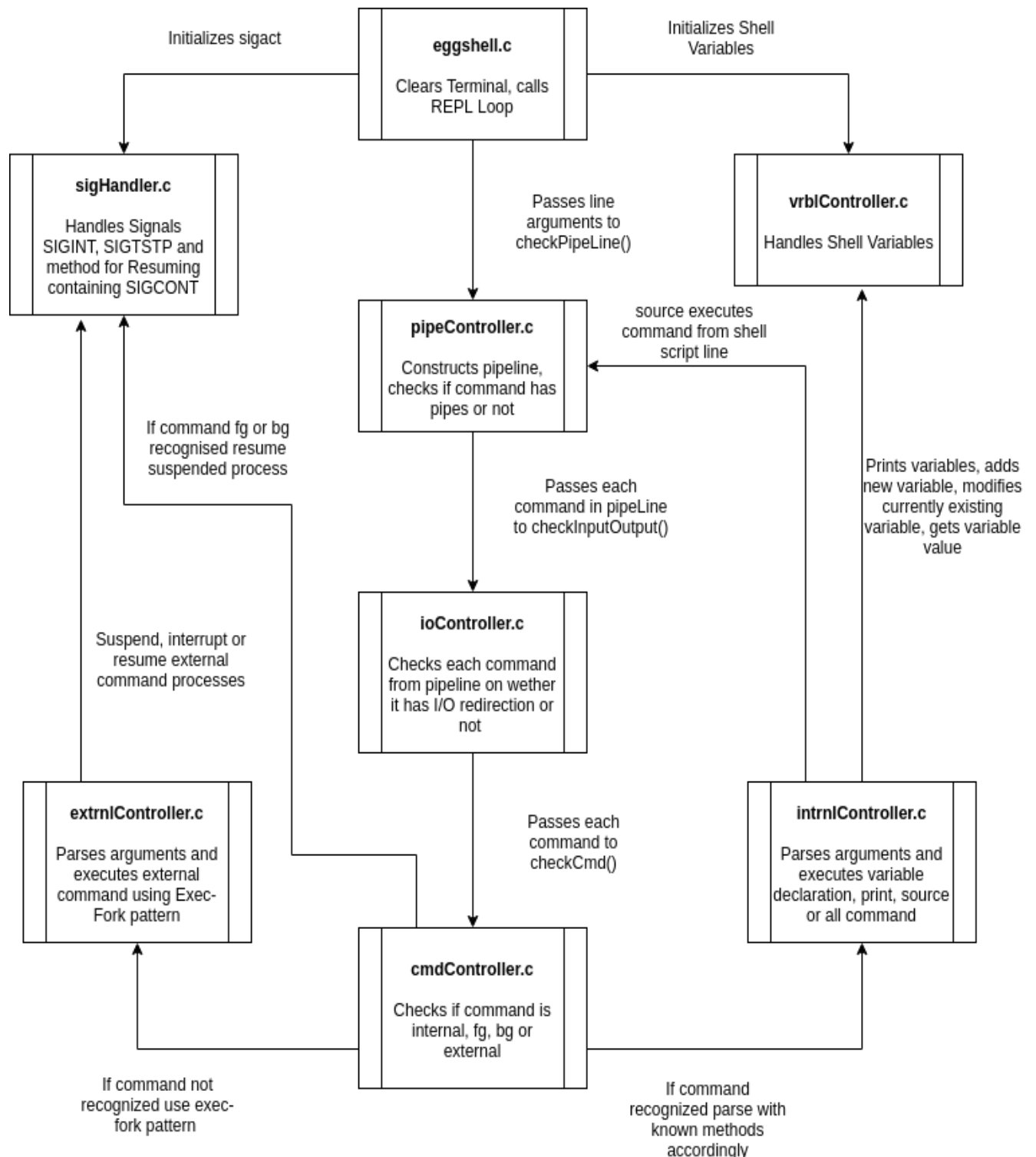
## Introduction to The Eggshell Implementation

### 1.1 Implementation Description

The implementation of the eggshell for this assignment entails the following phases:

1. The terminal of the user is cleared, a welcome message is displayed and initial shell variables as well as sigact (for signal listening) are initialized
2. The linenoise REPL (Read-eval-print-loop) is accessed and on each line entry by the user, their line is parsed into tokens with space as the delimiter. Before parsing it firstly checks if the first token is "exit", if it is it exits and clears the terminal to revert back to the normal Unix shell.
3. If not exit, the array of arguments of the line are then passed to **checkPipeLine()** where a pipeLine is constructed and the program reasons whether the line entered involves piping or not. If it does, then every command is processed to **checkInputOutput()** and the output is set to be the input of the command after. If not, then the one command is passed to **checkInputOutput()**.
4. In **checkInputOutput()** the command is checked for any I/O redirection symbols and sets the STDOUT or STDIN accordingly to/from the file the user requires in their command entry. After this the command is passed to **checkCmd()**.
5. In **checkCmd()** the command is checked on whether it is internal, fg, bg, or external. When internal it is passed to the command's appropriate method in **intrnlCmdParser.c**, and when external it is passed to the fork-exec method in **extrnlCmdParser.c**. When the command is seen to be **fg** or **bg** it calls a method for resuming processes in **sigHandler.c**.
6. When the user requests external command, during execution it can be interrupted or suspended via the signal handler initialised in phase 1. They can also then be resumed using "fg" or "bg" described in phase 5.
7. After the command is processed and the program responds appropriately it returns back through all the methods it came from and proceeds to the next linenoise prompt for the next line input by the REPL loop described in phase 2. This keeps on going until the user decides to enter exit.

## 1.2 Data Flow Diagram



## Variables (vrblController.c)

### 2.1 Dynamic Array for Storing Variables

A dynamic array was implemented for storing variables. To achieve this, structs called **shellVariable** (typedef **Var**) and **variableArray** (typedef **VarArr**) were set up. Character pointers for the variable called **name** and **value** were declared in **Var**. A double pointer **varArr** (not to be confused with Pascal-case **VarArr**) was declared to point to the **Var** struct. The variable **amount** from the **varArr** struct was used as a counter to keep track of how many variables are currently in the array.

For ease of access the dynamic array **variables** was globally declared in **vrblController.c**. This was done by making a pointer to the **VarArr** struct. It is later memory allocated in the **initShellVariables()** method.

### 2.2 Accessing Variable Values

Values of variables are accessed by calling **getVarValue("\$VAR")**. This method was designed to work with catering for '\$' as it is used heavily in the internal print and variable initialization command. It works by pointing after the \$ and looping through all of the variables in **varArr** until a name match is found using **strcmp**. If a match isn't found then an error is printed and the exitcode is stored accordingly.

Note: **getVarIndex(1)** works similarly except it doesn't require '\$' in front of the variable name and it works with returning the index location rather than the string. It's prime use is in **addVar(2)** for accessing an existing variable's index location if a match is found.

### 2.3 Adding Variables

Variables are added to the dynamic array by the method **addVar(2)**. It takes the string name and string value as arguments (in character pointer form). The method firstly check whether the variable name entered is valid by calling **validateVarName(1)**. Here the name is passed and it is checked whether it consists of alphanumeric characters and/or underscore characters. If this rule isn't followed an error is returned and the variable is not added to the array.

After this name validation the variable is checked to see if it already exists or not. If it doesn't exist then variable has memory allocated for it as an index in the double pointer **varArr**. Its name and value are then set accordingly and the **amount** counter is incremented by 1.

If it already exists then the value of the variable is overwritten as the new one entered in the method call's second argument. It accesses the already existing variable by **getVarIndex(1)** which takes the

name of the variable as an argument and returns the index of where it is stored in the **varArr**. (Note if the variable name doesn't exist then it returns -1, an error is returned and the variable isn't added).

## 2.4 Initializing Shell Variables

The method **initShellVariables()** is called from **eggShell.c** to allocate memory for the dynamic array **variables** and to add the initial variables PATH, PROMPT, CWD, USER, HOME, SHELL, TERMINAL and EXITCODE. The dynamic array first starts allocating space for one variable and everytime a variable is added by calling the **addVar(2)** method, its size increases by one.

### Easy-to-set variables

Firstly, the EXITCODE variable was added to the variable array. It's value is initially set as "null" but throughout the program it's value is changed depending on whether the last process succeeded or failed (0 for success and -1 for failure).

Secondly, all the environment variables were added by calling the library function **getenv(1)** for their values. These consist of PATH, USER and HOME.

Lastly, all the other variables CWD, PROMPT, SHELL and TERMINAL were set calling the **setShellSpecific()** method. In **setShellSpecific()** TERMINAL value was achieved by calling **ttyname(STDIN\_FILENO)**. Set methods for the rest of the variables are then called.

### Shell Specific Variables

SHELL was set in its own separate method **setSV()** as it required multiple lines for retrieving the to-be-added value. It worked by getting the current process ID of the terminal by **getpid()** to create a path string called **pathString** and pass it through the **readlink(3)** method. This then returns the desired path value in the second argument which was placed as a malloced character pointer called **path**. This is then passed as the value to be set for SHELL.

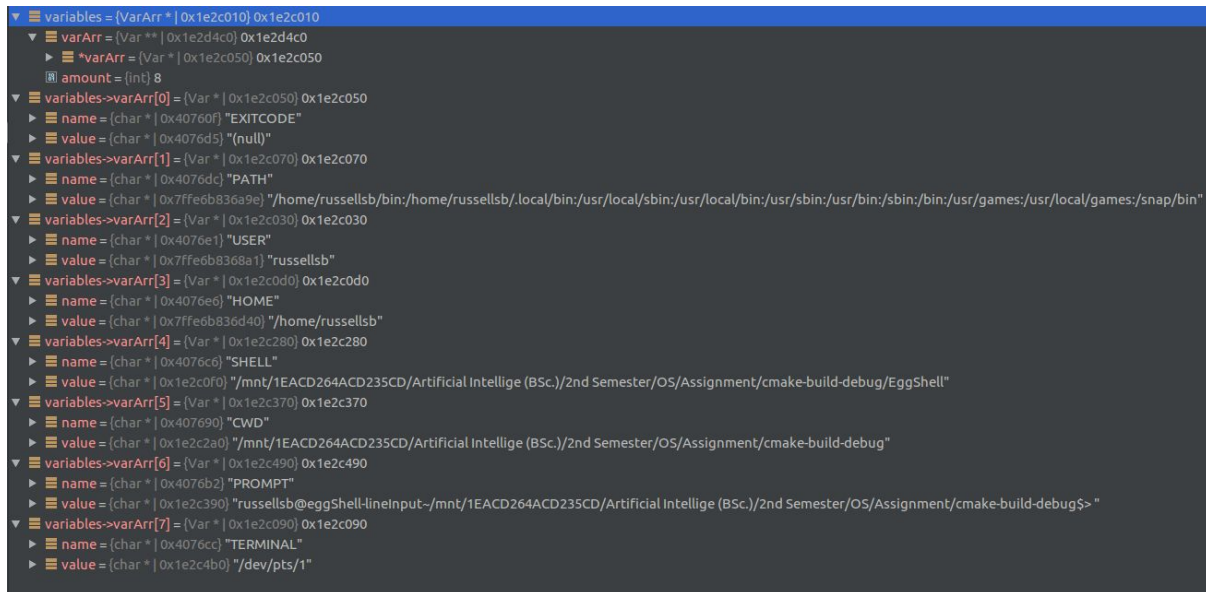
CWD and PROMPT were set up in their own separate methods because the linenoise prompt uses the variable PROMPT's value and the prompt is designed as a string made by concatenating USER's value along with the eggshell name "eggShell-lineInput" and CWD's value.

Since the linenoise prompt is dependant on the variable PROMPT value after every iteration, the variable value needs to be updated considering that the current working directory CWD can be changed using the internal command **chdir**. That's why by abstracting **setCWD()** and **setPROMPT()** they can easily be updated for later for when the current working directory is changed.

Both CWD and PROMPT made use of library function **strdup(1)** for memory allocation for the string values to be stored. This was used because when being displayed as the prompt by linenoise, without strdup it was experiencing unknown modifications to their string values after every REPL iteration. After each iteration they would at times store no string values at all and other times corrupt to a string of garbage characters. This bug was affecting the user experience tremendously but was fortunately solved with **strdup(1)**.

## 2.5 Testing Shell Variables

In order to test that the shell variables were being initialised correctly the GDB debugger (used in CLion) was used. With this it verified that they were being stored in the array as intended, as can be seen in the screenshot below. This is the variables array after all the initial shell variables are added



```

variables = {VarArr * | 0x1e2c010| 0x1e2c010
  varArr = {Var ** | 0x1e2d4c0| 0x1e2d4c0
    *varArr = {Var * | 0x1e2c050| 0x1e2c050
      amount = {int} 8
    variables->varArr[0] = {Var * | 0x1e2c050| 0x1e2c050
      name = {char * | 0x40760f| "EXITCODE"
      value = {char * | 0x4076d5| "(null)"
    variables->varArr[1] = {Var * | 0x1e2c070| 0x1e2c070
      name = {char * | 0x4076dc| "PATH"
      value = {char * | 0x7ffe6b836a9e| "/home/russellsb/bin:/home/russellsb/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin"
    variables->varArr[2] = {Var * | 0x1e2c030| 0x1e2c030
      name = {char * | 0x4076e1| "USER"
      value = {char * | 0x7ffe6b836ba1| "russellsb"
    variables->varArr[3] = {Var * | 0x1e2c0d0| 0x1e2c0d0
      name = {char * | 0x4076e6| "HOME"
      value = {char * | 0x7ffe6b836d40| "/home/russellsb"
    variables->varArr[4] = {Var * | 0x1e2c280| 0x1e2c280
      name = {char * | 0x4076c6| "SHELL"
      value = {char * | 0x1e2c0f0| "/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug/EggShell"
    variables->varArr[5] = {Var * | 0x1e2c370| 0x1e2c370
      name = {char * | 0x407690| "PWD"
      value = {char * | 0x1e2c2a0| "/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug"
    variables->varArr[6] = {Var * | 0x1e2c490| 0x1e2c490
      name = {char * | 0x4076b2| "PROMPT"
      value = {char * | 0x1e2c390| "russellsb@eggShell-linelnput-~/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug$>"
    variables->varArr[7] = {Var * | 0x1e2c090| 0x1e2c090
      name = {char * | 0x4076cc| "TERMINAL"
      value = {char * | 0x1e2c4b0| "/dev/pts/1"

```

As for the user adding variables to storage and overwriting currently existing variable values from the terminal itself, that is explained and tested later in the section about internal commands. In that section they do not need to be tested using the debugger as the print command is then used to retrieve the stored/modified values.



## Line Parsing (eggshell.c)

### 3.1 The Main Method

The main method **main()** firstly clears the terminal the user executes the binary file from. It does this using **linenoiseClearScreen()** from **linenoise.c**. This clears the screen as well as allows the user to clear the screen when in the egg-shell by pressing CTRL+L on the keyboard.

After cleared, a welcome message is printed to the user and the eggshell is called by **execEggShell()** which contains all the eggshell method calls as well as the REPL. Note that on exiting the **execEggShell()** method **linenoiseClearScreen()** is then called yet again, this is just to clear the eggshell for when the user exits and wants to revert back to their terminal's normal shell.

### 3.2 The Read-Evaluation-Process-Loop (REPL)

The method **execEggShell()** can be considered as the root function of the whole eggshell. This is because it contains the Read-Evaluate-Process-Loop (**REPL**) consisting of command/line parsing as well as all the method calls to the eggshell functions.

It firstly starts of by initialising all the variables to be used for tokenization in the REPL. Though, before the REPL it makes a call to **initShellVariables()** for shell variable initialization and **checkForSignals()** for initializing sigact and the signal handler. It then calls **linenoiseHistorySetMaxLen()** to set the max amount of line inputs from the REPL to be saved in history (this could then be accessed by the user using the up and down arrow keys).

The REPL is implemented using a while loop that follows the condition:  
(*line = linenoise(getVarValue("\$PROMPT"))*) != *NULL*. This basically makes a call to the **linenoise(1)** function in order to prompt the user and read line input by the user.

The prompt works dynamically as it is set corresponding to the value of the shell variable PROMPT. This makes it so that when the current working directory, USER value, or the PROMPT value directly is changed, the linenoise prompt would be updated on every iteration as well.

For line input, if the user enters nothing than the prompt just goes to the next prompt iteration. Else it adds the line to linenoise history by **linenoiseHistoryAdd(line)**. It then gets the first token of the line using **strtok(2)** with spaces as the delimiter. This first token is checked whether it is "exit" or not using **strcmp(2)**. If equal to "exit" than the REPL loop is broken out of using *break* and before exiting all the shell variables are freed using **freeAllVar()** from **vrblController.c**.

Otherwise, the rest of the line is tokenized and stored in the string array **args[]**. This array of user inputted arguments is then passed to **checkPipeLine(1)** stored in **pipeController.c**.

## Pipeline Construction and Evaluation (pipeController.c)

### 4.1 Checking for Piping by Pipeline Construction

In the method **checkPipeLine(1)**, called by **execEggShell()** a 2D array called **cmdArray** is initialized and filled to contain all commands in the pipeline.

If no pipe separators denoted by '|' are found in the inputted arguments then only one command is stored in the **cmdArray**. Otherwise, when an arbitrary amount of pipes are detected the piped commands are stored in the pipeline **cmdArray** accordingly, one after another.

This is all constructed using a for loop with conditions using **strcmp(2)** for detecting whether the current argument is a '|' or not. If not then the command is stored in the command array at the appropriate index, if it is then the index for the next command insertion is incremented, along with the counter for arguments resetted. On finishing the construction loop, the end of the **cmdArray** is always set to store **NULL** at the index after the last command so that the end of the current pipeline is always known. This is especially useful for traversing the pipe line in the **pipePipeLine(2)** method.

If the **cmdArray** is seen to store more than one command then it is passed through **pipePipeLine(2)** along with the integer amount of commands as the second argument (This was counted using the variable counter **c** during the pipeline construction loop). Otherwise, only one command is inputted with no need for piping so the single command is passed to the next phase **checkInputOutput(1)** in **ioController.c**.

### 4.2 Evaluating the Pipeline

#### Phase 1: Variable Initialization

In the **pipePipeLine(2)** method firstly all the required variables for piping are initialized, as can be seen in the code listing below;

```
int pAmnt = cmdAmnt - 1; //amount of pipes
int fdAmnt = 2*pAmnt; //every pipe has two file descriptors
int fd[pAmnt * 2]; //initializes needed file descriptors
int status; //used for waiting in the parent
int c = 0; //initialized new counter for commands
int j = 0; //counter for the first fd in each pipe
pid_t pid; //process id of the current process
```

## Phase 2: Pipe Creation

Then, all the pipes needed in the pipeline are created using a for loop dependant on **pAmnt**. For every pipe in the pipeline a pipe is created using **pipe(fd + i\*2)**.

Though it might appear strange at first, the argument **fd + i\*2** is passed because the **pipe()** function works by pointing to the beginning of each pipe, and the ends of each pipe are accessed by file descriptors stores in the file descriptor array **fd[]**. (Note: every pipe has two file descriptors).

```
//Creates all needed pipes at the start
for(int i = 0; i < pAmnt; i++){
    if(pipe(fd + i*2) < 0){
        perror("Error: Pipe creation has failed");
        addVar("EXITCODE", "-1"); //exit code to -1, as error occurred
        return;
    }
}
```

## Phase 3: Command Traversal for Setting Up read and/or write in Pipe Line

A for loop for traversing through every command in the pipeLine is made to set up a fork-exec pattern for every command. The child handles all the reading and writing between commands using two conditions that can be seen in the code listing below.

```
//child writes to the next command if not the last command
if(c!=cmdAmnt-1){
    //sets the current command to write to the next
    if(dup2(fd[j+1], STDOUT_FILENO) < 0){
        perror("Error: writing process whilst traversing the pipeLine has failed");
        addVar("EXITCODE", "-1"); //exit code to -1, as error occurred
        exit(EXIT_FAILURE);
    }
}

//child reads from the previous command if not the first command
if(c!=0){
    //sets the current command to read from the previous
    if(dup2(fd[j-2], STDIN_FILENO) < 0){
        perror("Error: reading process whilst traversing the pipeLine has failed");
```

```

        addVar("EXITCODE","-1"); //exit code to -1, as error occurred

        exit(EXIT_FAILURE);
    }

```

When the current command is not the last its STDOUT is set to the fd of the next command. When the current command is not at the beginning its STDIN is set to the fd of the command before it. This is achieved using **dup2(2)** and making use of the **STDOUT\_FILENO** and **STDIN\_FILENO** macros.

These conditions keep on applying for every command that is gone through, and the writing and reading ends up being piped appropriately. In each child after these two conditions, a for loop is used to close the ends of the current pipe using **close(1)**, and the current command is passed to the next phase **checkInputOutput(1)** in **ioController.c**.

## Phase 4: Closing all pipes and waiting for children in Parent

As a precautionary measure the parent closes all the pipes in the pipeline using a for loop on **close(1)** and waits for all the children using a for loop on **wait(&status)**.

```

//parent closes all the pipes in the pipeLine
for(int i = 0; i < fdAmnt; i++){
    close(fd[i]);
}

//parent waits for children
for(int i = 0; i <= pAmnt; i++){
    wait(&status);
}

```

## 4.3 Testing Piping

Using the terminal and external commands piping was tested in the eggshell for the piping scenarios of one pipe, two pipes, three pipes and no pipes.

### One Pipe

```
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD$> ls | sort
```

```
28829778_1836440316386973_1173085128_n.jpg
```

Artificial Intelligence (BSc.)

Config.Msi

gig\_Songs.jpg

Individual stuff

MusicWork

\$RECYCLE.BIN

Sixth Form

System Volume Information

Windows10Upgrade

WindowsFiles

## Two Pipes

```
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD$> ls | sort | wc
```

```
11  17  203
```

## Three Pipes

```
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD$> ls | sort | figlet | wc
```

```
96 1178 5754
```

## No Pipes

```
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD$> print this is detected as one command!
```

```
this is detected as one command!
```

## Input and Output Redirection (ioController.c)

### 5.1 Checking for Input and Output Redirection

In the method **checkInputOutput(1)** each argument of the command is gone through and checked for **>**, **>>**, **<** or **<<<** symbols using **strcmp(2)**, and when found the indicating variable **flag** is set to 1, 2, 3 or 4 for **>**, **>>**, **<** or **<<<** respectively. If the flag is left as initialized, that is 0, then it doesn't direct any input or output and the command arguments are passed to **checkCmd(1)** in **cmdController.c**.

If the flag is 1 or 2 then output redirection is detected so the command along with filename **args[i]** and the flag are passed into **redirectOutput(3)**. If the flag is 3, input redirection from file is detected so the command and the filename are passed to **redirectInputFile(2)**. If the flag is 4 then input redirection from a here-string is built by a string being built from all of the tokenized arguments after **<<<** and is passed along with the command to **redirectInputString(2)**.

### 5.2 Output Redirection

In the method **redirectOutput(3)** the standard output is connected to the given file using **stdoutToFile(2)**. This method returns the file descriptor of the file **fd2** so that the output could be reverted back to normal in **stdoutToNormal(1)**. The command is then passed to **checkCmd(1)** to execute the command and after that **stdoutToNormal(1)** is called using the file descriptor returned from **stdoutToFile(2)** as an argument. This resets the stdout back to output to the terminal as normal.

In **stdoutToFile(2)**, In initialization it either sets file descriptor **fd1** to create a file when the flag is 1 for **>**, or it sets **fd1** to append to an existing file when the flag is 2 for **>>**. This is done using the library function **open(3)** having **fileName** as the first argument, **oflag** macros as the second and **mode\_t** mode as the third. Note that for appending to a file it is made so that if it doesn't exist it is created.

After initialization, **fd1** is checked for failure in opening the file (specified by **fileName**) for writing, then **fd2** is set using **dup(STDOUT\_FILENO)**. This sets the output to the file descriptor instead of stdout. After this method call, **fd2** is checked for failure. On succession the method then returns the integer variable **fd2** to be used in **stdoutToNormal(1)** for reverting back to standard output.

**stdoutToNormal(1)** works by calling **dup2(fd2, STDOUT\_FILENO)**, here it sets the file descriptor returned from **stdoutToFile(2)** to **STDOUT\_FILENO**, reverting the output of the program back to that of the terminal.

## 5.2 Input Redirection

In the method **redirectInputFile(2)**, similarly to how **redirectOutput(3)** works, the **fileName** is passed to **stdinToFile(1)** which returns a file descriptor **fd2**. **stdinToFile(1)** sets the standard input to the content of the file specified in the **fileName**. If **fd2** is -1 then no file was found so it just returns (The error message is printed from the **stdinToFile(1)** method itself). Using **fd2**, **stdinToNormal(1)** then sets the STDIN back to normal.

In the method **redirectInputString(2)**, a temporary file is created for temporarily storing the here-string contents for input retrieval. This is done using the code snippet below.

```
char * fileName = "hereString.tmp";

FILE *f = fopen("hereString.tmp", "w+");

for(int i = 0; stringArray[i] != NULL; i++){

    if(stringArray[i+1] == NULL)fprintf(f, "%s", stringArray[i]);

    else fprintf(f, "%s ", stringArray[i]);

}

fclose(f);
```

Here, the file **hereString.tmp** is created using library function **fopen(2)**. A for loop is then used to print all the arguments from **stringArray**, alongside an **fprintf(3)** call for writing the argument to the file. After this **stdinFile(1)** is called, the method **checkCmd(cmd)** is called for command execution then **stdinToNormal(fd2)** is called. Finally, the file **hereString.tmp** is deleted as it no longer requires use. This is done using the library function call **remove(fileName)**.

In **stdinFile(1)** the file descriptor is set to that of the file specified by **fileName** by using **open( fileName, O\_RDONLY)**. Similarly to **stdoutToFile(2)** **fd2** is set using **dup(STDIN\_FILENO)**. This sets the input to the file descriptor instead of stdin. After this method call, **fd2** is checked for failure. On succession the method then returns the integer variable **fd2** to be used in **stdinToNormal(1)** for reverting back to standard input.

**stdinToNormal(1)** works by calling **dup2(fd2, STDIN\_FILENO)**, here it sets the file descriptor returned from **stdinToFile(1)** to **STDIN\_FILENO**, reverting the input of the program back to that of the original.

## 4.3 Testing Redirection

Using the terminal and commands output redirect for writing to file `>` and appending to file `>>` were tested, then input redirection for retrieving input from file `<` and retrieving input from `<<<` were tested.

### Input Redirection: `>`

```
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug$> ls  
| sort | wc > outputTest.txt
```

//The program creates the file "outputTest.txt" and "12 12 158" is outputted to the file as expected.

### Input Redirection: `>>`

```
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug$>  
source sourceTest.sh >> outputTest.txt
```

//The program appends to the file the outputs of the command source sourceTest.sh as expected

### Output Redirection: `<`

```
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug$> wc  
< InputTest.txt
```

0 12 55

//The program, as expected, counts the words in the text stored in InputText.txt, that is, "One Two Three Four, this is a test for input using //wc"

```
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug$>  
sort < InputTest2.txt
```

1  
2  
3  
4  
5  
6  
7  
8



9

A

B

C

D

//The program as expected, sorts the numbers and characters that are retrieved as input from InputTest2.txt.

## Input Redirection: <<<

```
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug$> wc  
<<< I am to be counted as a string
```

```
0 8 31
```

//The program is seen to retrieve the input from the here-string using a temporary file, as expected

## Note

All the input and output files used for these tests can be found in the same directory as the EggShell binary “cmake-build-debug”. Here, they can be seen to store their expected outputs from the commands used, and expected inputs for the commands to use to be able to print such results to the terminal.

## Commands (cmdController.c)

### 6.1 Checks Whether Command is Internal or External

In **checkCmd(1)** the char \* args[**MAX\_ARGS**] are reasoned through a collection of if statements to detect whether it is an variable command, a print command, a chdir command, an all command, a source command, an fg or bg command, or an unrecognised command.

When the command is recognised it is passed to its appropriate parsing method in **intrnlCmdParser.c**. This stores a collection of internal parsing procedures for execution. When the command is unrecognised the program goes to the else statement, where **externalCmd(args)** from **extrnlCmdParser.c** is called to pass the data to **externalCmd(1)**'s fork-exec-pattern.

### 6.2 Internal Command Methods excluded from intrnlCmdParser.c

It is good to note that **fg** and **bg** are recognised commands but their parsing methods were not included in **intrnlCmdParser.c**. This is because they are heavily involved with signal handling as they have to do with resuming a process from the top of the stack of suspended processes. Therefore when any of these keywords are detected, appropriately, the program calls **resumeSuspended(1)** from **sigHandler.c**.

The number **1** is passed as a flag for indicating that fg was entered and **0** is passed for indicating that bg was entered. The program later reason with this and decides whether to wait for the resumed process to finish, making it in the foreground, or the not wait, making it in the background.

## Internal Commands (intrnlCmdParser.c)

### 7.1 The Variable Declaration Command

**parseVrblCmd(char \* args[MAX\_ARGS])** works by taking the command argument array as a parameter. It firstly initializes **varName** and **varValue** and then allocates memory for **args[0]** using **strdup(1)** this was done as it would previously stay crashing upon the later use of **strsep(2)**, returning a **SIGSEGV** error.

Two tokens are then achieved from the "VARNAME=value" argument. One that is the variable name which is before the delimiter '=' and the other is the variable value which is all the string after '='. The method **strsep(2)** was used as opposed to **strtok(2)** as the program caters for any '=' characters set in the value. Like this it only splits the token for the value after the first '=' character, modifying the statement argument to just the value.

Error checking to see if the **varName** or **varValue** were not set then occurs. Then another error check to see if the LHS variable has a '\$' in front of it also occurs. After error checking, the program checks if the RHS has a '\$' in front of it. If it does it calls **getVarValue(varValue)** from **vrblController.c** to get the actual value of the RHS value that is actually a Shell variable. If the variable is not found the function returns "(null)" into **varValue** therefore an error check is done comparing it to "(null)" and if matched it returns out from the method, stopping it from adding a non-existing value from a non-existing variable to the LHS variable.

The variables **varName** and **varValue** are then passed through **addVar(2)** in **vrblController.c**. This automatically reasons whether the variable value should be created or the variable name already exists so the value should just be edited.

### Testing Variable Declaration Command

In the terminal EggShell was loaded and the following commands were entered for testing;

```
russellsb@eggShell-lineInput~/home$> VAR=GeorgeOrwell1984
russellsb@eggShell-lineInput~/home$> NoRHS=
Error: Incorrect entry of a variable declaration command. Please input in the form VAR_NAME=var_value
russellsb@eggShell-lineInput~/home$> =NoLHS
Error: Incorrect entry of a variable declaration command. Please input in the form VAR_NAME=var_value
russellsb@eggShell-lineInput~/home$> VAR1=$VAR
russellsb@eggShell-lineInput~/home$> VAR2=$NonExistentVar
Error: Requested Shell Variable not found.
russellsb@eggShell-lineInput~/home$> all
```

```

EXITCODE=404
PATH=/home/russellsb/bin:/home/russellsb/.local/bin:/home/russellsb/bin:/home/russellsb/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
USER=russellsb
HOME=/home/russellsb
SHELL=/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug/EggShell
PWD=/home
PROMPT=russellsb@eggShell-lineInput~/home$>
TERMINAL=/dev/pts/18
VAR1=GeorgeOrwell1984
VAR=GeorgeOrwell1984

```

```
russellsb@eggShell-lineInput~/home$> $VAR=3
```

Error: Right Hand Side Variable name shouldn't contain a \$ before it's name. Please input in the form VAR\_NAME=var\_value

//The program works as expected as all the variables can be declared and modified appropriately, including functional error cases.

## 7.2 The Print Command

**parsePrintCmd(char \* args[MAX\_ARGS])** works by taking the command argument array as a parameter. The method firstly checks if there are any arguments are given to **print**, if none are then an empty line is echoed back to the user, this mirror's how the UNIX shell's **echo** works.

At first this method was designed to simply just print back the arguments using a loop, though several edits were made to get the print function to be able to recognise variables and print back there values, as well as dealing with quotation marks and handling non-variable character lying in the same argument as a variable. Example "print \$USER.vx.,fdfs" wouldn't return variable not found but it would print back "russellsb.vx.,fdfs".

A for loop was made to traverse through the arguments of **print**, and inside its scope it goes through the following procedure of checking whether the argument contains a variable, the start of a quote by quotation marks, else it just prints the arguments normally.

### Printing back Variable Values

It stores a string word after the first character, storing this to the variable **after\$**. The argument is then checked to see if it has '\$' at the beginning. If it doesn't then **after\$** is never used. If it does, in the same conditional statement it check if the first character in **after\$** is equal to 0 or not. This indicates whether there exists characters after the dollar sign. If this condition is then also met then it enters the scope of accessing variables to print back out from the argument. If not it checks whether it has quotation mark at the beginning then goes to the else statement.

The program makes use of a nested loop for traversing characters in the argument, flag variables, string buffers and distinguishment between variable-valid characters and other characters. The variable-valid characters are appended to the buffer **tempName** and other characters after the Variable name are appended to the buffer **tempRem**.

An error check is used to see if the parsed name in **tempName** is valid through the condition on comparing the retrieved **value** to “(null)”. (This is performed after the **getVarValue(tempName)** call that stores the value to the string pointer **value**). If it is “(null)” then that means the requested variable name doesn’t exist and therefore it exits the method.

After this the value of **tempName** is concatenated to the buffer **tempString** using **strcat(2)** and the remainder characters after it are concatenated after the value also using **strcat(2)**. The current argument is then set to **tempString**, as it is the argument after parsing the variable value from the variable were required. Then this argument is printed back as per usual.

## Testing Printing Variables

The following commands were entered for testing;

```
russellsb@eggShell-lineInput~/home$> print $VAR
GeorgeOrwell1984
russellsb@eggShell-lineInput~/home$> print $VAR1-A-Great-Book!
GeorgeOrwell1984-A-Great-Book!
russellsb@eggShell-lineInput~/home$> print $Non-Existent-Variable
Error: Requested Shell Variable not found.
russellsb@eggShell-lineInput~/home$> print $
$
```

//The program works as expected as the variables are printed as well as the string after it, also error cases were proved to be functional

## Printing back text in Quotation Marks

In the else if statement after the if statement of printing back variables, the argument is checked to see if it starts with a “. This is done using pointers and checking the first character of the string pointer which is that character the string points to. When this is seen to hold true the program enters the scope.

In this scope, the program works by using two nested loops nested inside the main argument loop. The program gets the string after the “ in the first argument and appends it to the string pointer array **tempArgs**. When an argument is finally found with a quotation mark in it, using string functions and buffer **tempString**, it is checked for whether it is at the end or the letter before the last. (This caters for “quotes like this” and “quotes like this!”)

If the last is never found then an error is printed back to the user, else when found the quote is successfully parsed into **tempArgs[i]** and all its contents are printed to the terminal. The counter for arguments *i* in the main scope is also incremented to continue print checking after the last argument with the closing quotation mark.

## Testing Printing Quotes

The following commands were entered for testing;

```
russellsb@eggShell-lineInput~/home$> print "To be or not to be that is the real question"
To be or not to be that is the real question
russellsb@eggShell-lineInput~/home$> print "Not valid q"uote
Error: Please put the terminating quotation mark at the end of your word
russellsb@eggShell-lineInput~/home$> print "This is valid though!"
This is valid though!
russellsb@eggShell-lineInput~/home$> print "As you can see $USER," it prints $USER after the quote!
As you can see $USER, it prints russellsb after the quote!
russellsb@eggShell-lineInput~/home$> print "Unfinished quote
Error: No terminating quotation mark found, please finish your quote
```

//The program prints quotes as expected along for the error cases of no quote termination and improper quote termination

## 7.3 The Chdir Command

**parseChdirCmd(char \* args[MAX\_ARGS])** works by taking the command argument array as a parameter. The method has three main conditions and an else statement.

### The First Condition

The first condition is met when no arguments are given to **chdir**. Therefore the input command line would be just "chdir". When this is accessed the program aims on going to the initial working directory, that is the root directory of the executed binary for **EggShell** itself. This path string is accessed by using **getVarValue("\$SHELL")**.

A buffer is used to temporarily stores the value of \$SHELL. The string is then manipulated on buffer so that it goes it displays the root directory. This is done by deleting every character after the last '/' including the last '/' itself. This is achieved using a decrementing while loop, and setting every character that needs to be deleted to 0.

The modified buffer is then passed to the **chdir(1)** method to change the current working directory, and **setCWD()** and **setPROMPT()** are called to update both the \$CWD variable and \$PROMPT variable.

## The Second Condition

The second condition is met when the command “chdir ..” is inputted. This simply calls **chdir(“..”)** to set the current working directory to the root. As per usual, **setCWD()** and **setPROMPT()** are then called to update the variables.

## The Third Condition

The third condition is when a second argument is detected but it does not match “..” Here it is taken as a string to directly be passed to **chdir(1)**. It firstly makes use of a buffer to contain all the arguments given to **chdir**. Although this appears strange, this was done to cater for paths that have spaces in it. Like this the **chdir** does not block it out as invalid but actually checks if the directories exist. This is because there exists some directories that exist with spaces.

After building the buffer from the whole path string (possibly inclusive of multiple arguments) the path string stored in **buffer** is then passed to **chdir(1)**. If the method returns a -1, then it means the change in directory failed as the requested directory does not exist. An appropriate error message is returned and the program exits the method. If the method doesn't return a -1 then the directory was validated, changed successfully, therefore **setCWD()** and **setPROMPT()** are then called for updating.

## The Else Condition

If none of the conditions are met and the else is accessed then that means that more than one argument was put for “chdir ..” Example, the user input could be “chdir .. Random Illogical Arguments !!” An error message for this is returned appropriately and \$EXITCODE is updated.

## Testing the Chdir Command

The following commands were tested to check each condition in chdir parsing.

```
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug$>
chdir ..
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment$> chdir ..
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS$> chdir ..
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester$>
chdir /mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/1st Semester
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/1st Semester$> ls
```

Academic Reading and Writing in English ICT Students' Association Vienna Trip 2018 Terms and Conditions.pdf

```

Mathematics of Discrete Structures  Probability, Sampling and Estimation    Prolog    Foundations of Artificial Intelligence
Mathematics for Engineers    Matlab101    Programming Principles in C    transcriptSem1.pdf
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intellige (BSc.)/1st Semester$> chdir Mathematics for Engineers
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intellige (BSc.)/1st Semester/Mathematics for Engineers$> ls
Larson R., Edwards B.H. and O'Neill P., Mathematics for Engineers, Custom Edition for the University of Malta, Cengage, 2015.pdf
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intellige (BSc.)/1st Semester/Mathematics for Engineers$> chdir
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intellige (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug$> ls
a.out Assignment.cbp CMakeCache.txt CMakeFiles cmake_install.cmake EggShell InputTest2.txt InputTest.txt Makefile outputTest.txt
sourceTest.sh
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intellige (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug$>
chdir .. Not valid
Error: Please enter one argument only for "chdir ..".
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intellige (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug$>
chdir NotValid
Error: The path "NotValid" was not found please change to a directory that exists

```

```

//The program updates the CWD and PROMPT variables appropriately. Also all conditions of chdir seem to respond as predicted and the
//error cases are proved to be functional

```

## 7.4 The Source Command

**parseChdirCmd(char \* args[MAX\_ARGS])** works by taking the command argument array as a parameter. The method firstly checks if more than one arguments are attempted to be fed to **source**, if so it returns an error and exits.

After this error check a file pointer **f** is declared for use with **fopen(2)**. This method is called in an if statement with the condition (**f = fopen(args[1], "r") != NULL**). This attempts to open the file name. If the file is found then the pointer is not equals to null and it accesses the condition's scope. Otherwise, if equal to NULL, the file does not exist in the current working directory, therefore an error is returned and the method is exited.

When the file is opened successfully, the program initialises various variables for parsing through the shell script's line commands and tokenizing them into arguments. It then accesses a while loop that makes use of **fgets(3)** to read the shell script line by line. Each line scanned is set so that the '\n' character at the end is removed (as it interferes with argument parsing). If the line is detected as empty then the program doesn't pass any commands and goes to the next iteration.

Otherwise, the line is tokenized into arguments using **strtok(2)** just like in the **REPL**. After parsed into tokens the **args2** array is passed to **checkPipeLine(args2)** to execute the command as if inputted from the terminal itself. At the end of this iteration **memset(3)** is called to clear the character array **line** for the next line and the variable **flag** is set to 1 to indicate that the buffer **linePtr** is malloced. This is done so that when exiting the while loop it detects that **linePtr** has been malloced and it frees it.



## Testing the Source Command

The following statements are stored in the file "sourceTest.sh" in "cmake-build-debug"

```
"
print
print Hey $USER! This source command seems to be working pretty well right?
print You know what? I'm going to have some fun
print First I'll print all your variables!
all
print Ooo, interesting "$CWD" I'll take you up to the root directory now
chdir ..
print it's now: $CWD
print Great now that's happened, I'll set your "$PROMPT" to:
PROMPT=INSANELY_LEGIT_PROMPT_:)_==>_
print $PROMPT
print If you'd like to reset the prompt use "chdir".
print Russell made it so putting "chdir" on its own sets it back to the initial directory and it affects the prompt!
print
"
```

These command statements are then called and executed with source command from the terminal. It displays the following;

```
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intellige (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug$>
source sourceTest.sh
```

```
Hey russellsb! This source command seems to be working pretty well right?
You know what? I'm going to have some fun
First I'll print all your variables!
```

```
EXITCODE=0
PATH=/home/russellsb/bin:/home/russellsb/.local/bin:/home/russellsb/bin:/home/russellsb/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/
sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
USER=russellsb
HOME=/home/russellsb
SHELL=/mnt/1EACD264ACD235CD/Artificial Intellige (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug/EggShell
CWD=/mnt/1EACD264ACD235CD/Artificial Intellige (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug
PROMPT=russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intellige (BSc.)/2nd
Semester/OS/Assignment/cmake-build-debug$>
TERMINAL=/dev/pts/18
```

```
Ooo, interesting $CWD I'll take you up to the root directory now
it's now: /mnt/1EACD264ACD235CD/Artificial Intellige (BSc.)/2nd Semester/OS/Assignment
Great now that's happened, I'll set your $PROMPT to:
INSANELY_LEGIT_PROMPT_:)_==>_
If you'd like to reset the prompt use chdir
Russell made it so putting chdir on its own sets it back to the initial directory and it affects the prompt!
```

```
INSANELY_LEGIT_PROMPT_:)_==>_source nonExistent
Error: The file "nonExistent" was not found.
INSANELY_LEGIT_PROMPT_:)_==>_source dsf dsf dsf
Error: Please enter a single file name argument for "source".
```

//The program seems to parse source commands appropriately, error cases are seen to act functionally.

## 7.5 The All Command

When the command entered is recognised as “all” then the method **printAllVar()** from **vrbIController.c** is called where basically a for loop is used to traverse through all the variables in the **variables** dynamic array and print their corresponding name and value individually.

### Testing the All Command

The following commands were inputted;

```
russellsb@eggShell-lineInput~/home$> all
```

```
EXITCODE=0
PATH=/home/russellsb/bin:/home/russellsb/.local/bin:/home/russellsb/bin:/home/russellsb/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
USER=russellsb
HOME=/home/russellsb
SHELL=/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug/EggShell
PWD=/home
PROMPT=russellsb@eggShell-lineInput~/home$>
TERMINAL=/dev/pts/18
```

```
russellsb@eggShell-lineInput~/home$> NEWVAR=WOW!!!
```

```
russellsb@eggShell-lineInput~/home$> chdir
```

```
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug$> all
```

```
EXITCODE=0
PATH=/home/russellsb/bin:/home/russellsb/.local/bin:/home/russellsb/bin:/home/russellsb/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
USER=russellsb
HOME=/home/russellsb
SHELL=/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug/EggShell
PWD=/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug
PROMPT=russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug$>
TERMINAL=/dev/pts/18
NEWVAR=WOW!!!
```

//The all command is seen to work as expected, printing back all the currently stored shell variables appropriately.

## External Commands (intrnlCmdParser.c)

### 8.1 Initializing Environment Variables

In the method **externalCmd(1)** a call is firstly made to **getEnviron()** here an array of environment variables are returned into the variable **envp** to later be passed into **execve(3)**.

In the method **getEnviron()** extern char \*\*environ is declared to access all the environment variables. Then memory is malloced for storing the terminal and cwd in the conventional environment variable form "VAR\_NAME=varValue." After this, using **strcat(2)**, the conventional string form is set up and **putenv(1)** is called to add them to **extern char \*\*environ** which contains all the environment's variables, with the now added terminal and cwd variables. Finally **environ** is returned.

### 8.2 Finding the Path to the External Command

In **externalCmd(1)** after getting the environment variables it moves on to getting the path to the external command binary file. This is done by initializing **char \* paths[MAX\_PATHS]** and then filling it with all possible paths to the command binary of the first argument using **fillPaths(paths, args[0])**.

After all possible paths are received **findSuccPath(paths)** attempts to filter down all possible paths for an existing path, when found the program copies that successful path into the string **succPath** using **strcpy(2)**. If the string of **succPath** is seen to be set to **-1** then that means that no possible path was found, an error is outputted appropriately and the method is exited.

#### Filling Possible Paths

**fillPaths(2)** works by using **getenv("PATH")** to get all the possible root directories to the external command binary. This was chosen to be used instead of **getVarValue(\$PATH)** as if the user were to modify the \$PATH variable, external command would simply not work as no paths would be found. This way with using **getenv(1)** over **getVarValue(1)** the EggShell's external command functionality is protected from the user.

After this the method constructs an array of all the possible paths separated by the delimiter ":" using **strtok(2)**. After separation each string is manipulated so that "/" + **args[0]** is concatenated to the end of it using **strcat(2)**. This is done to find every combination of paths

possible to the external command's file. After string manipulation the string is then left in the string array **paths** which is to be returned.

## Filtering Possible Paths to the Successful Path

**findSuccPath(1)** works by receiving all possible paths as a parameter. Traversing through all the paths using a for loop, it makes use of the library function call **access(paths[i], X\_OK)** which basically returns 0 when the path is able for access and therefore valid.

When 0 is returned from this method call then this successful path is returned. Otherwise, when success is never reached then the path simply doesn't exist and the string "-1" is returned as an indicator for this predicament.

## 8.3 The Fork-Exec Pattern

After the environment variables are updated with cwd and terminal, and a path to the external command file is found, the method gets to fork-exec pattern. Firstly **pid** is declared for storing the current process ID in this fork-exec pattern. The variable **pid** is then forked using **fork()**. Two if conditions are made for accessing the child and the other for the parent. Else, an error has occurred during the forking process, the program prints this using **perror(1)** and the method is exited.

### In the Child

In the child, accessed by the condition **pid == 0**, the library function **execve(3)** is called passing the successful path to the command binary **succPath**, the string array of arguments **args** and the string array of environment variables **envp** as arguments. After **execve(3)** is performed successfully **exit(0)** is called to show that the child exited successfully and did not run into an error in that external command's execution.

### In the Parent

In the parent, accessed by the condition **pid > 0**, the global variable **current\_pid** is set to **pid**. This is a variable that is declared in the header **eggshell.h**, this is used to store the current\_pid from this fork-exec pattern so that it can be accessed from the signal handler in **sigHandler.c**. After this the process is set to have its own unique process group id using the library function **setpgid(current\_pid, current\_pid)**. This is done to allow sending signals to resumed processes after suspension without sending it to the whole children process group but just the singular most recently resumed process only.

An integer variable **andOperatorAtEnd** is then initialized to 0 as a flag for detecting if the last argument is &. The last argument is then scanned using a for loop and a conditional statement, when seen to match '&' the variable **andOperatorAtEnd** is set to 1. If no match is found then it is left as 0.

If **andOperatorAtEnd** is detected as 0 then **waitpid(pid, &status, WUNTRACED)** is used to wait for the child process to finish executing. This contains several status check for the exit status of the child process. Otherwise, in the else, **andOperatorAtEnd** is 1 therefore the program doesn't bother about waiting for the child process, making it run in the background.

## 8.4 Testing External Commands

The following commands were inputted to test external command functionality;

```
russellsb@eggShell-lineInput~/home$> ps
  PID TTY          TIME CMD
 25719 pts/18    00:00:00 bash
 25780 pts/18    00:00:00 EggShell
 25781 pts/18    00:00:00 ps
russellsb@eggShell-lineInput~/home$> firefox &
russellsb@eggShell-lineInput~/home$> firefox
^C
russellsb@eggShell-lineInput~/home$> notARealCommand!
Error: Path to external command was not found: No such file or directory
russellsb@eggShell-lineInput~/home$> env
```

As can be seen in the above commands, output is returned as expected. When the '&' operator is set firefox is executed in the background, otherwise the program waits for it as it's in the foreground and CTRL+C has to be used to send a signal to it to interrupt it.

In the last command where "env" was entered, the output was too large to document but at the end of enlisting all the environment variables the variables **TERMINAL** and **CWD** are seen to be amongst them at the end. This ensures they were passed correctly. Here is a snippet of the output;

```
UPSTART_EVENTS=xsession started
XDG_SESSION_DESKTOP=ubuntu
LOGNAME=russellsb
COMPIZ_BIN_PATH=/usr/bin/
QT4_IM_MODULE=xim
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-ODliYR935c
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/usr/local/share:/usr/share:/var/lib/snapd/desktop
LESSOPEN=| /usr/bin/lesspipe %s
UPSTART_JOB=unity7
INSTANCE=
XDG_RUNTIME_DIR=/run/user/1000
DISPLAY=:0
XDG_CURRENT_DESKTOP=Unity
GTK_IM_MODULE=ibus
LESSCLOSE=/usr/bin/lesspipe %s %s
LC_TIME=en_GB.UTF-8
LC_NAME=en_GB.UTF-8
XAUTHORITY=/home/russellsb/.Xauthority
BASH_FUNC_generate_command_executed_sequence%%=() { printf '\e\7'
}
OLDPWD=/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment
_=./EggShell
TERMINAL=/dev/pts/18           \< ----- TERMINAL's here!
CWD=/home                     \< ----- CWD's here!
```

## Signal Handling (sigHandler.c)

### 9.1 Initializing the Sigaction Struct

In **checkForSignals()** the **struct sigaction sa** is initialized in order to send signals to the signal handler. The method was adapted for **sigaction(3)** over **signal(2)** as **signal(2)** varies between systems, whereas **sigaction(3)** is safer in functionality and generally more used.

For **sa** the handler is set to the **signalHandler()** method, an empty set of signals to be blocked is initialised and it is finally set that the function restarts if interrupted by the handler. This can be seen in the code snippet below;

```
struct sigaction sa; //initialize sigaction struct

sa.sa_handler = signalHandler; //sets handler to the above signalHandler method
sigemptyset(&sa.sa_mask); //initializes additional set of signals to be blocked during signalHandler()
sa.sa_flags = SA_RESTART; //restarts function if interrupted by handler
```

Signal handlers for CTRL+C and CTRL+Z for interrupting and suspending respectively are set up;

```
//creates signal handlers for CTRL+C and CTRL+Z
if(sigaction(SIGINT, &sa, NULL) == -1) printf("Error: Couldn't trap CTRL+C\n");
if(sigaction(SIGTSTP, &sa, NULL) == -1) printf("Error: Couldn't trap CTRL+Z\n");
```

This is made to access the **signalHandler()** method on key-combination detection.

Two global variables are also initialized in **sigHandler.c** when **checkForSignals()** is called, these variables are **suspendedPids**[**MAX\_SUSPENDED**] and **topOfTheStack**. The array **suspendedPids** stores suspended process IDs in a stack-like manner, meanwhile integer variable **topOfTheStack** is used to point to the top of the stack. It is initialized as -1 as the stack doesn't as of yet exist.

## 9.2 The Signal Handler

In the method **signalHandler()** a new line is printed to **stdout** using **fprintf(2)**. This is used over **printf()** as it is apparently much safer than **printf()** in signal handlers. This new line is printed so that the prompt of the next **REPL** iteration doesn't get printed in the same line as **^C** or **^V**

After this two conditional statements are set up. One is for detecting the signal number **signo** as **SIGINT**, the other for detecting **signo** as **SIGTSTP** which are for interrupting a process and suspending a process respectively.

When **SIGINT** is detected using the library function call **kill(current\_pid, SIGINT)**, an interrupt signal is sent to the current process and it is interrupted successfully. Note that **current\_pid** is accessed from the header file **eggshell.c** and it is set to the current process ID from the exec-fork pattern in **extrnlCmdParser.c**.

Otherwise, when **SIGTSTP** is detected **topOfTheStack** is incremented by one as the suspended stack is to be increased and the **current\_pid** is pushed into the stack using the new **topOfTheStack** as an index to **suspendedPids**, i.e. **suspendedPids[topOfTheStack]**. After this **kill(current\_pid, SIGTSTP)** is called to send a signal to the current process for suspension.

## 9.3 Resuming Suspended Processes

The method **resumeSuspended(1)** is accessed from **cmdController.c** when internal command **fg** or **bg** are detected. The program firstly caters for an error case when the user tries to resume a process when no processes are as of yet added to the stack. This is done by checking whether the top of the stack is equals to **-1**, if so it returns an error and exits the method otherwise, the program continues.

The **current\_pid** is firstly set to the **pid** at the top of the **suspendedPids[]** stack, using **topOfTheStack** as an index. Using the function call **kill(current\_pid, SIGCONT)**, the process is then resumed. The previous top of the stack **pid** is then set to **0** as the process is pulled out from the stack and **topOfTheStack** is decremented.

The program then comes to two conditions. One where **resumeTo** is **1**, indicating the process is resumed to the foreground with **fg**. The other where **resumeTo** is **0**, indicating the process is resumed to the background with **bg**.

When resumed to the foreground the same waiting for the **current\_pid** procedure is used, as in the fork-exec pattern in **extrnlCmdParser.c**. Otherwise, when resumed to the background the program simply does not wait and just leaves it running in the background.

## 9.4 Signal Testing

Signaling was tested by executing firefox, gedit then rstudio through their external commands. They are all suspended and added to the stack and one by one are then pulled from the stack and resumed.

```
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug$>
firefox
^Z
Exited with status 20.
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug$>
gedit
^Z
Exited with status 20.
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug$>
rstudio
^Z
Exited with status 20.
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug$> fg
load glyph failed err=6 face=0x2a3a330, glyph=2797
load glyph failed err=6 face=0x2a3a330, glyph=2797
^C
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug$> bg
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug$> fg
^C
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug$> bg
Currently no suspended processes found! Can't resume what isn't paused
russellsb@eggShell-lineInput~/mnt/1EACD264ACD235CD/Artificial Intelligence (BSc.)/2nd Semester/OS/Assignment/cmake-build-debug$> fg
Currently no suspended processes found! Can't resume what isn't paused
```

As expected after suspension rstudio is resumed to the foreground as well as firefox. On the other hand, the second pulled process is not waited for as it is resumed to the background. Error cases for no suspended processes to resume is also seen to hold.



## Conclusion

### 10.1 Special Features

- In **print**, the program allows variables to be printed alongside non-variable characters in the same argument. Example: “print hello \$USER,!£ how are you today!” will not try to retrieve variable from “USER,!£” and return an error that it doesn’t exist, but rather distinguish variable characters from non-variable and print back “hello russellsb,!£ how are you today!”
- In the internal command, **chdir**, if the user types “chdir” as a single argument the program will go back to the initial working directory, that is, the directory of the binary file used to execute the EggShell. This is done using the variable \$SHELL and going to the path String’s root directory with character manipulation.
- Linenoise was taken advantage of and it was made so the user is allowed to clear the EggShell using **CTRL+L** and also that they can access there previous commands through the up and arrow keys using **linenoiseHistorySetMaxLen(1)** and **linenoiseHistoryAdd(1)**.
- The prompt works by continuously updating itself every **REPL** iteration, and it was designed to consist of **\$USER** concatenated with **\$CWD** to give the user information every prompt line, just like in the Unix Shell.

### 10.2 Possible Future Improvements

- A lot of memory allocation calls throughout the program were used for storing strings. Since the program works in a **REPL** when new string variables are malloced previously malloced variables aren’t necessarily freed and possibly malloced over again. Even though this does not affect the program’s functionality too drastically, it can waste a lot of space in the heap of a computer after many iterations of overwriting variables, and lead to several memory-related problems such as memory leaks. (Even though no memory leaks have been found in current testing)

A solution to this would be to free previous values and malloc them again. This would require extensive reasoning for the program to detect whether the variable already has space allocated or not and possibly realloc if the new value requires more space. This was left out as not enough time was available to add this to the program.

- It was noticed that when the **\$CWD** is long, and the terminal width is small, linenoise seems to crash and return a **SIGSEGV** error as it tries to retrieve line data from the line when the too-long prompt pushes it to the next. This could either be a bug with linenoise itself or something with how the prompt is set using linenoise or variable value retrieval is not 100% supported. Though the EggShell doesn't crash at all when the CWD is of normal length and the terminal width isn't compressed. This could possibly be fixed using linenoise's multiline, error casing for when the line is not able to be accessed, or memory allocation for line.
- Since the EggShell's welcome message involves ASCII art the welcome message would break if the binary is executed from a terminal with compressed width. This can be either fixed by redesigning the welcome message entirely or trying to find a way to automatically resize the user's terminal appropriately when the EggShell is started up.
- The internal **source** command works through going through the shell file line by line. The shell script doesn't cater for lines that are commented starting with **#**. A quick solution to this would be to implement a condition for this when it is detected at the beginning of the line pointer, so that the program does not try to parse it as a command, ignores it, and just goes to the next line.
- Instead of making a header file for every source file a single global header file called **eggshell.h** was used. This was done simply for convenience and access purposes. For better program design, security and efficiency the header can be split into multiple, say a header for each source file, with shared method/struct/variable declarations only where appropriate.
- The program made it so that when an external command is called it's process group id is changed to a unique one so that when the process is resumed after suspension, and a signal is sent to that process like **SIGINT**, the signal would be sent to that process and that process only and not all the other child processes in the suspended stack as well. (Resuming each process one by one would then have them close on resuming when the process group id is equivalent).

Though this fixed the resume top of suspended stack problem, another problem arises were certain commands that make use of the **pgid** just crash on execution. This include the command **top**.

This could be solved by finding some way to change the process group ID when **SIGTSTP** is sent to the process. Though, this was tried and the method **setpgid()** blocked the program from changing this outside the fork-exec pattern. Another possible solution could be to find an alternative to setting a unique process group everytime when the external command process is executed, maybe some other method to fixing the resume top of suspended stack problem.