Eggshell

an Operating Systems project

CPS1012

Andre' Jenkins

76999M

Contents

Chapter 1

Structure of the code

The code was structured into different directories, so that they may be organised according to what they are supposed to achieve.

1.1 Main files

The files here are ones which are used in the front layer of the eggshell. Here you may find a main C file that uses the eggshell functions, the main eggshell `.c / .h` files which either call multiple other functions, or are simple enough to be on the front layer, and other files such as the Makefile, any scripts, testfiles, and other miscellaneous files.

main.c – The main `C` file that the executable is retrieved from. Uses libraries such as `eggshell` and `linenoise`.

eggshell.c/h – The `eggshell library` used by the main file in order to start the eggshell and utilise it. Uses multiple libraries that are all found in the `src` directory.

Makefile/Makefile-GCC – The Makefile necessary to generate the executable. The current default Makefile uses the `Clang` compiler, for reasons stated in the `README.md` file. To use the Makefile that utilises the `GCC` compiler instead, either change the name of the `Makefile-GCC` file, or run `switch.sh`

switch.sh – A script that aids in switching compilers for the makefile. This was written for each switching between `Clang` and `GCC`, due to the ease of debugging with `Clang`, and the standard nature of the `GCC` compiler.

README.md – The `README` file holds the instructions to compiling the program, as well as a quick summary of the program and its utilities.

testinput.txt – A file used to test the capabilities of the eggshell. Running `./eggshell test` will immediately launch the eggshell and run this script, to provide a quick testing method.

LICENSE, .gitignore & .yml files – Files that are unimportant to the project itself. These were used for `git` purposes, as the project was also uploaded as a `git` repository.

1.2 Source files

The files found here are the bulk of the code making up the eggshell. In here, every single `eggshell header library` is present, all of them with their own specific and complicated purpose. These were seperated from the main `eggshell` file in order to organise the core of the project from the specific elements making up the project itself.

variables.c/h – Contains all the functions and structs relating to the variables created and stored by the `eggshell` . For example, all the **shell** variables can be found here.

printer.c/h – The main file dealing with the `print` command.

proc_manager.c/h – The file dealing with the execution of external commands.

sig_handler.c/h – Contains the `signal handler` function used in order to suspend and interrupt processes. Also contains an additional function in order to reawaken a suspended process.

redirection.c/h – The main file dealing with `input/output` redirection.

pipe_manager.c/h – Contains functions dealing with the piping system that the `eggshell` offers. Also contains a special execution function, rather than using the one found in `proc_manager.c/h`

1.3 Other files

There are also other directories that contain files that aren't integral to the functionality of the eggshell, but are still related somewhat.

documentation/ — Contains the `.tex` file that generated this report, as well as other items related to it. In order to recompile it, you'd most likely need to install TEXlive first.

ci/ — Unrelated to the main project. The Makefile here is used for Continuous Integration for the `git` repository.

add-on/ — Contains the `linenoise.c/h` library that was used in the main file to simulate a terminal's prompt with input.

.vscode/ — Contains files that helped with debugging/building the project in `Visual Studio Code` .

Chapter 2

Code documentation

2.1 `main.c`

2.1.1 Use of `main.c`

This code is mainly used in order to produce an executable using the `eggshell`. This is because the eggshell is mainly used as a sort of `API`, which interfaces with the inner functions that the eggshell offers.

As a result, the `main` implements some not-so-integral elements of the eggshell, such as an introduction/boot-up screen, an additional `test` argument feature, and some `linenoise` functions such as history *(ability to use ↑ and ↓ keys to traverse through older commands)* . It also uses the function `updatePrompt();` which updates the prompt of the eggshell in order to display the current directory.

A thing to note is that the external command `clear` is run by the line:

```
runLine("clear", "");
```

...which is an external function found in `eggshell.c` .

Note: In order to test the first part of the program, which loads `testinput.txt` and runs it, you'd need to run the eggshell with the `test` parameter: `./eggshell test`

2.2 eggshell.c

2.2.1 Use of eggshell.c

This file serves as the **core** of the eggshell itself. Its main method, `execute` , executes the line inputted by the user. Almost every other function of the eggshell is accessed through this main file automatically, using parsing.

2.2.2 void initEggshell ()

All this function does is initialise the eggshell. To initialise, all it needs to do is initialise its shell variables, which is does by calling the method `initShellVars()` from `variables.c`

2.2.3 void runLine (char *command, char *line)

This function executes the command `command` with the arguments found in `line` . An example of this is:

`command = "chdir"` and `line = "../src/"`

Which would execute `chdir` with arguments `../src/` . It does this by having a significant if/else-if block that checks which function to execute:

```
1 if(strcmp(command, "print") == 0) printLine(line);
2 else if(strcmp(command, "all") == 0) showShellVars();
3 else if(strcmp(command, "vars") == 0) displayUserVars();
4 else if(strcmp(command, "chdir") == 0) changeDirectory(line);
5 else if(strcmp(command, "source") == 0) runScript(line);
6 else if(strcmp(command, "fg") == 0) resumeProcessSignal(FOREGROUND);
7 else if(strcmp(command, "bg") == 0) resumeProcessSignal(BACKGROUND);
8 else externalCommand(command, line);
```

The signal handler is also initialised by calling the `init_handler()` function in `sig_handler.c` .

2.2.4 void changeDirectory
(char* directory)

This function changes the directory to the directory specified by the parameter. This is done using the inbuilt `chdir` function.

If changing the directory was successful, the `$CWD` shell variable is updated, and the new directory is displayed.

However, if it was failing, `perror` is used to display the error message.

2.2.5 void runScript
(char *filename)

All this function does, is load a `script` file which can have any extension, and execute its commands line by line. It contains measures, such as ignoring any line starting with `#` , or any empty lines. This effectively enables commenting support for scripts using the `#` symbol.

It also emulates a prompt, so that the output generated when running the script could be more readable to the user.

Important Note: `fgets` with a buffer was used to store all the lines into a buffer array, after which the lines from said array will be loaded one by one. This approach was chosen over using `fgets` directly because of a **critical** issue that surfaced when the source file contained an invalid line.

The invalid line would attempt to be executed as an `external command` , after which the `execve` would fail, causing the function that handles external commands to `_exit` . This function not only effectively terminates the child, but also **resets all file pointers** .

This effectively duplicated the execution of every consecutive command in the file. Not only that, but if **two** invalid commands were used, the `source` command would lock the `eggshell` into an uninterruptable loop that requires a `SIGKILL` signal to halt.

2.2.6 void execute (char *line)

This function is the main one to be executed. It is what reveals the eggshell functions to the user, as all other functions are accessed via this one.

What it does first of all, is check the line's structure in order to parse it for specific cases. For example, in this section here:

```
27 char rest[2] = "\0";
28 if(strcmp(line, "exit") == 0){runLine("clear", ""); exit(0);}
29
30 // Check for variable assignment
31 if(parse_var(line) == 0){return;}
32 if(pipe_parser(line) == 0){return;}
```

The first line checks whether the line is identical to the exit command `exit`, at which point the `clear` command is run, and the program stops executing. Line 30 and 32, use functions from `variables.c` and `pipe_manager.c` respecively, to check whether the line assigns a variable, or contains pipes in it. An exitcode of 0 from these functions means that they were detected.

If none of these are detected, the function then attempts to check whether the line contains any redirection symbols such as `>`, `>>`, `<` or `<<<`:

```
37 // Checks for different redirection symbols
38 char* redirect_to_file  = strstr(line, ">");
39 char* append_to_file    = strstr(line, ">>");
40 char* input_from_file   = strstr(line, "<");
41 char* input_here_string = strstr(line, "<<<");
42
43 int out = 0, in = 0; // flag variables for redirection
```

The results of the `strstr` function used are stored in variables. These variables have two uses: They are used as flags to check which redirection symbol was found, **and** they point to the first character of the symbol itself, which is useful. The flag variables `out` and `in`, also serve a purpose which will be gone into later on.

After this, a conditional `if-else` block is used to parse the line for redirection. Depending on which `char*` flags where set, the respective function from `redirection.c` is called:

```
45 // Checks for redirection
46 if(append_to_file != 0 || redirect_to_file != 0){
47   filename = out_redirect_parse(append_to_file, redirect_to_file,
     ↪  line);
48   out = 1;
49 }
50 else if(input_here_string != 0 || input_from_file != 0){
51   filename = in_redirect_parse(input_from_file, input_here_string,
     ↪  line);
52   in = 1;
53 }
```

One can also note, that one of the integer flag variables `in/out` is also set to 1. This is used to call the appropriate redirection function later on.

After doing so, two lines are used. One uses the `strsep` function to seperate the command from the arguments, and the other calls the redirect initialisation function from `redirection.c`

```
55 // Seperates the command from the arguments
56 char *command = strsep(&line, delimiter);
57
58 int filefd = init_redirect(filename);
```

After this, the actual execution of the function is done. Depending on the integer flags triggered, the appropriate block of code in the conditional loop is run. One begins redirecting the input, one redirects the output, and in the case that neither of these flags are set, the line is simply executed.

```
60 if(out == 1){
61   redirect_out(filefd);
62   runLine(command, line);
63   close_redirects(OUT, filefd);
64 }
65 else if(in == 1){
66   redirect_in(filefd);
67   runLine(command, line);
68   close_redirects(IN, filefd);
69 }
70 else{
71   runLine(command, line);
72 }
```

2.3 printer.c

2.3.1 Use of printer.c

This file handles the `print` command. It's main purpose is to print the line, being able to detect whether a part of the string is escaped in quotation marks, and replacing unescaped variable references with their value.

2.3.2 void printLine (char *line)

The main function of the file, it is the primary function to be called from `printer.c` . It achieves the aforementioned goal by splitting the line accordingly, and passing the segments of the split line to the other function in the same file.

The splitting is done using `strsep` , and an integer variable `escaped` was used to determine whether the segment was escaped.

For example, a line such as:

hello $USER, and wel"come to the $HOME"

Will be split into several pieces like:

hello — $USER, — and — wel — come — to — the — $HOME

...and the segments are then sequentially handed off to the other function. Note that after the `wel` segment, the `escaped` variable is flipped.

2.3.3 void printSeg
`(char *segment, int escaped)`

This function utilises its parameters to print the segment, replacing the segment [or a part of it] with a value if it is found to be a variable name.

If the `escape` variable is set to 1, then the segment is simply printed without considering the rest of the function. However, if it isn't, then the following steps are covered:

- **IF** the first character is '$', start reading what's after it.
  - { **IF** the string contains any invalid variable characters *(any character that isn't a capital english letter)* , terminate reading, and consider what has been read to be the variable name.
  - { Retrieve the value of the variable with the name that was read.
  - { **IF** the variable exists, and the reading was stopped, print the value, followed by the rest of the segment.
  - { **ELSE IF** the variable exists, and no invalid characters were found, print the value.
  - { **ELSE** , print the segment.
- **ELSE** , print the segment.

For example in the line above, $USER would be replaced by the value, whereas $HOME would be printed as it is, as it was escaped.

A limitation of this is that if a segment contains $ anywhere that isn't its first character, it is ignored. For example, if $A is B, `$Aees` would be printed as `Bees` , whereas `a$Ae Lincoln` would remain the same, instead of being printed as `aBe Lincoln` .

2.4 sig_handler.c

2.4.1 Use of sig_handler.c

This file contains functions relating to any signals, or any effects resulting from signals. To be more precise, it contains a *signal handler* , and a way to wake up a suspended process.

This was split into a seperate file for sake of organization, as having it bundled with the `eggshell.c` file would have added more complexity to the already significant file.

2.4.2 void signal_handler (int signo)

This is the signal handler that is initialised by `sigaction` .

```
11 void signal_handler(int signo){
12     pid_t process = currentpid();
13
14     // Catches interrupt signal
15     if(signo == SIGINT){
16         int success = kill(process, SIGINT);
17     }
18
19     // Catches suspend signal
20     else if(signo == SIGTSTP){
21         int success = kill(process, SIGTSTP);
22         resuspended = 1;
23         suspended_process[last_suspended+1] = process;
24         last_suspended++;
25     }
```

It is lacking in `printf` statements, as after some research, it was found that it is not `re-entrant` , or `async-signal-safe` , meaning that if, for example, in the middle of running `printf` in a signal handler, another signal is caught, the program might end up in a deadlock.

A `SIGINT` signal is handled by simply sending the signal to the current process, whereas a `SIGTSTP` is handled by sending the signal, then setting the `resuspended` flag to 1, after which the pid of the now suspended process is appended to an array of pid's using a variable that keeps count of how many suspended processes there are. This ensures that multiple processes can be suspended and restored normally.

2.4.3 void resumeProcessSignal (int state)

This function activates in the case of either the `fg` or `bg` commands. It's purpose is to resume a suspended process by calling a function from `proc_manager.c` . This is done in order to abstract the process managing *(with `waitpid` etc...)* from the signal handler, while at the same time allowing access to the pid of the suspended process.

The correct pid is handed off to the function by using the `last_suspended` variable. This makes process suspension seem like a stack, **last suspended, first resumed** . If the suspension succeeds, the `last_suspended` variable is decremented, and the exitcode is set to 0. However, if for some reason it does not succeed, there are two cases: *the process is unresponsive* or *the process was resuspended.* In this case, the exit code is set to reflect which outcome happened, `-1` being an error and `18` being an indication of another suspension *(18 is the SIGNO of SIGTSTP)* .

2.4.4 void init_handler ()

All this function does is initialise the signal handlers that are to be used. It does this by use of `sigaction` rather than `signal` , for multiple reasons:

- `signal` performs differently on different systems, making it an unpredictable function to use.
- `signal` automatically resets the signal action back to `SIG_DFL` after use. This means that during the time in which the signal is triggered and the handler is reinitialized, *the system is vulnerable to additional signals, which would not be handled* .
- `sigaction` permits the blocking of signals, which if needed in the future, would be impossible to implement with `signal`

Other than the initialisation of the `sigaction` struct, the handler is initialised to deal with `SIGINT` and `SIGTSTP` signals accordingly.

2.5 redirection.c

2.5.1 Use of redirection.c

The purpose of this file is to handle all **input** and **output** redirection. It parses a line for any redirection symbols, initialises the redirection using a filename *unless* <<< *is detected* , and also starts the redirection, depending on its own flags and how they were set.

2.5.2 Parsing functions

The functions `out_redirect_parse` and `in_redirect_parse` both have very similar code that functions differently in other to be more specialised while also simplifying the code from having too many conditional blocks.

Both of them require three strings, **two** specifying pointers to the symbols, and **one** which contains the line itself.

These are passed in the parameter in order for the functions to know which action is supposed to happen, and how the line is meant to be parsed. This is because, in the case of < and > , one can simply split according to those symbols and be left with the two elements required, albeit with additional whitespace, whereas in the case of <<< and >> , one would need to perform some pointer arithmetic to split the strings accordingly.

Let's take the `in_redirect_parse` function as an example:

```
42 char *filename;
43
44 if(in_string != 0){
45   filename = in_string+3;
46   ins = 1;
47 }
48 else{
49   filename = in_file+1;
50   inf = 1;
51 }
```

Here, `in_string` and `in_file` serve as both pointers to the symbol, and flags as to which symbol is present, with `in_string` pointing to <<< and

`in_file` pointing to `<`. If the former was detected, `filename` is set to a pointer *three characters away* from the symbol, whereas if the latter was detected instead, `filename` is set to a pointer *one character away*. In both cases, the respective flag is also set.

This ends up setting the filename to whatever is after the symbol, as whichever symbol was pushed, the pointer was moved away from it anyways.

However, this leaves two problems - the left part of the line still hasn't been parsed yet, and the start of the filename may still contain whitespace.

These are both solved by the upcoming `strsep` and pointer arithmetic. The `line` string, which still points to the whole thing, is split according to the `<` delimiter *(`>` for output)*, causing the `line` string to now contain only the command. As for the filename, it is placed in a loop, where if its first character is a space, its pointer is incremented by one. This means that it does not matter how many whitespaces are behind the filename, they will all be removed by the loop. As for the command, any whitespace at the end will not interfere, as the `execution` function in `proc_manager.c` will ignore those.

2.5.3 `int init_redirect (char *filename)`

All this function does is open the correct file descriptor, depending on which flags were set. This is done using an integer variable `filefd`, which would hold the descriptor, and the `open` function. The differences between `>` and `>>>` is simply whether the `O_TRUNC` macro or the `O_APPEND` macro is used, and for `<`, it's as simple as using `O_RDONLY`.

However for `<<<`, the `filename` string would contain an actual string, rather than a filename. As a result, a temporary file `stdin.tmp` is created in order to store the string in, and the temporary file is then subsequently opened. This has the benefit of having the redirection be the same for both `<<<` and `<`, abstracting it from the actual redirection function.

2.5.4 Redirection functions

These are grouped together for the same reason the parsing functions are. The code is similar, however to avoid unnecessary long conditionals, they were split into two.

In this case, we'll take the example for the output redirection:

```
86  int redirect_out(int filefd){
87      // Duplicates file transcriptor 'stdout' to saveout
88      save_out = dup(fileno(stdout));
89
90      // Sets stdout file transcriptor to our transcriptor 'filefd'
91      if(dup2(filefd, fileno(stdout)) == -1){
92        perror("Failed in redirecting stdout");
93        setExitcode(255);
94        return 255;
95      }
96
97      return 0;
98  }
```

The first `dup` call is used in order to duplicate the file descriptor of `stdout` to `save_out`. This is needed later in order to restore `stdout`.

Then, the `dup2` in the if-condition sets our readymade file descriptor, which was set in the `init_redirect` function, as the replacement for `stdout`. If this fails, the redirection aborts, and the error code is reported, as well as the exitcode being set accordingly.

However if it succeeds, 0 is returned, showing that the redirection occured successfully.

This differs from `stdin` in that a different variable is used to store the file descriptor for `stdin`, and `stdin` is replaced instead of `stdout`.

2.5.5 void close_redirects (int direction, int filefd)

What this function does is reset all of the redirection that was initialised and used so far. It does this by flushing `stdout` or `stdin`, depending on which kind of redirection occured, and then closing the file descriptor `filefd`.

Our saved file descriptor, `save_in/save_out`, then replaces the place our old, now closed file descriptor occupied, after which that is closed as well.

Depending on whether the temporary file was created, it is then removed, as we have no more use for it.

After doing so, to ensure that everything is reset, every single flag variable within the file is set to 0, including the `save_out/save_in` descriptors. This is so that the next time redirection happens, it would be as if a redirection never happened at all.

2.6 proc_manager.c

2.6.1 Use of proc_manager.c

The use of this file is to manage everything doing with processes. This includes a fork-exec pattern for external commands and handling the resumption of suspended processes.

2.6.2 char** pathsToCommArr (int *pathn, char *program)

This function is meant to be a *helper* to the main function of the file, as it produces an array of possible PATH s that show where the product is.

Firstly, the PATH variable is retrieved, and copied into a seperate string, so that any modifications to the new string do not reflect back to our shell variable. Then, the array of paths is created and initialised.

After doing so, we have the following code:

```
99  void *tofree = paths;
100
101 strcpy(paths, pathORIG); // Done to keep original paths intact
102
103 char **patharr = (char**) calloc(1,100);
104 char delimiter[2] = ":";
105
106 int pathnL = 0; // used for path indexing
107
108 while(paths!=0){
109   char *path = strsep(&paths, delimiter);
110   patharr[pathnL] = (char*) malloc(100);
111   strcpy(patharr[pathnL], path);
112
113   // Appends program to be run to the individual PATH
114   strcat(patharr[pathnL], "/");
115   strcat(patharr[pathnL], program);
116
```

The delimiter's purpose is to store what we will be splitting the paths by — the colon. pathnL will also be the temporary variable in which we'll store

the amount of paths.

Inside the loop, `path` will store the freshly seperated path from the `paths` string, at which point it is stored in the `patharr` array. Said path then has `/<program name>` appended to it, as that would be the possible full path of the program.

After doing so, the path array is reallocated some additional memory in advance, so that it would be able to store an additional path.

Once the loop breaks, the parameter `*pathn` is set to `pathnL`. This is so that the length of the paths can be used outside this function. After doing so, the array of paths is then returned.

2.6.3 `void externalCommand` `(char *command, char *varargs)`

This function executes a command that does not belong to the `eggshell` using `fork-exec`. The parameters are the name of the command itself, and a single string containing all the arguments of the command. A command with no arguments will have an empty string as `varargs`.

The function first initialises all the variables it will need:

```
14  char **args = (char**) calloc(1, 80); // Array of arguments
15  char *arg; // Individual argument used to add to array
16
17  int argc = 1; // Counts how many arguments are present
18  char arg_delimiter[2] = " "; // To delimit the varargs
19
20  int BG = 0; // Runs process in background or foreground
21
22  char **envp = environEGG(); // Retrieves env variables from eggshell
23
24  pid_t pid = fork();
25
26  // Splits the path variable into an array of paths and stores them in
     ↪  'paths'
27  int pathn;
28  char **paths = pathsToCommArr(&pathn, command);
```

Here's a small description for each of these variables:

args – Where the array of arguments will be stored.

arg – Temporary variable used to store the argument to be added to `args`

argc – The number of arguments present.

arg_delimiter[2] – Used to split the `varargs` string into individual arguments

BG – Flag whose purpose is to signal whether the process should be run in the background or foreground. 0 means foreground.

envp – Contains all environment variables, the ones belonging to the main shell, **and** `eggshell` variables.

pid – Stores the process id of the forked process.

pathn – Contains the amount of paths that are present in `paths`

paths – The array of paths to cycle through when executing a command.

The following loop then splits `varargs` into `args` using the `arg_delimiter` , after which the main conditional block is encountered.

Here, there are three possibilities. We are in the child, we are in the parent, or the forking failed. First, we will consider the child.

For the child, it is pretty simple. The first argument in `args` is set to the current `path` we are attempting to access, and the numerous paths are cycled through until the program is found and then executed, or all the paths are exhausted and an error message is returned. `_exit()` was used rather than `exit()` , as the latter was causing issues when combined with the `source` command.

For the parent, it is slightly more complicated. First, the `current_pid` global variable is set to the pid of the process to be used by the signal handler, and then the `pgid` of the forked process is set to itself. This means that if a signal is recieved, the signal sent by the signal handler will only affect the current process running in the `foreground` . If this line was omitted, all children would receive the same signal, even ones who are suspended or in the background.

After doing so, an if block will execute if `&` was not detected at the end of `varargs`. If it was, the function will ignore this block, hence letting the process run in the background. However if it wasn't, the parent will wait for the process to finish:

```
65  if(setpgid(pid, pid) != 0) perror("setpid");
66
67  // Waits if background flag not activated.
68  if(BG == 0){
69    // WUNTRACED used to stop waiting when suspended
70    waitpid(current_pid, &status, WUNTRACED);
71
72      if(WIFEXITED(status)){
73        setExitcode(WEXITSTATUS(status));
74      }
75      else if(WIFSIGNALED(status)){
76        printf("Process received SIGNAL %d\n", WTERMSIG(status));
```

The `status` variable stores the status code of how the process terminated. The `WIFEXITED` and `WIFSIGNALED` macros check whether the process terminated normally or with a signal. Then, `WEXITSTATUS(status)` or `WTERMSIG(status)` will show us the exitcode of the process, or the signal that stopped the process respectively.

`WUNTRACED` was used in `waitpid` as it waits for the process to finish, however if the process is suspended, it will immediately stop waiting and continue on. After this point, the function will then return.

If, however, the forking failed, `perror` is used to showcase why that is the case.

2.6.4 int resumeProcess (int state, pid_t process)

The purpose of this function is to resume a previously suspended process. This is done by using the `pid_t` given in the parameter, and checking whether the program should be resumed in the foreground or background using the `state` variable.

Firstly, the `resuspended` variable is reset to 0, after which a `SIGCONT` signal is sent to the suspended process. If this failed, `perror` is used to show why, and an error code of 1 is returned.

If it doesn't fail, the `current_pid` is then set to the now recovered process. The same waiting conditional block that was in `externalCommand` is placed here, depending on whether `state` was set to the macro `FOREGROUND` or `BACKGROUND` .

There is an extra step however. `resuspended` is checked again, to know whether the resumed process was suspended again. If it is still set to 0, then an exitcode of 0 is returned, showing everything executed normally. However, if it is now set to 1, an exitcode of 18 is returned, showing that the process was suspended again. This number was chosen because it is the signal number of `SIGTSTP` .

Note: The `currentPid` function is omitted from the documentation as its function is obvious, and it only contains one line of code as a result. Its only purpose is to make a variable accessible by other files.

2.7 variables.c

2.7.1 Use of variables.c

This file handles everything to do with variables. This includes assignment, modification, retrieval, and manipulation. It also helps initialise the `eggshell` by setting the shell variables. It is one of the most used files in the system, as multiple other functions require the use of variables, whether it be as a `struct` or just their values.

2.7.2 int parse_var (char *line)

This function detects whether the line passed is attempting to assign a variable, or modify its value. It also validates the line, stopping the function if the assignment detected seems to be invalid. There are three rules for valid assignment:

- The variable name should be **capitalised**
- No spaces should surround the `=` symbol.
- The variable name cannot have whitespace.

These rules are put into place as they are the same syntax rules used by the `bash shell` .

First, we start with the assumption that the line **is** an assignment line, and then proceed to **use** or **validate** it:

```
11 int assign_check = 0;
12
13 char *vardelimiter = "=";
14
15 char *vartest = strdup(line);
16 void *tofree = vartest;
17
18 char *varname = strsep(&vartest, vardelimiter);
```

We copy the line into `vartest` so as to not modify it, and then we split the line into two, storing the supposed right side into `vartest`, and the left side in `varname`. It should stand to reason that, a line which does not have the `=` symbol is not trying to achieve assignment. Therefore the first check is whether `vartest` is empty.

The second check is whether `vartest` starts with whitespace, *and* whether `varname` has any spaces within it at all. Followed by a third check that checks all the letters within `varname` to see whether they are all capitalised. This is done using the capital-letter slice in the ASCII table.

If any of these checks fail, `1` is returned, meaning that the line should continue to be parsed, incase it should be treated as a command instead.

If they all succeed however, the `createVar` function is called in order to either create, or modify a variable.

2.7.3 `void createVar (char *line)`

This is the largest function in the file, due to the sheer amount of possibilities that the assignment is attempting to achieve, such as:

- Create a new variable with a string value.
- Create a new variable with the value of an existing variable.
- Modify an existing variable with a string value.
- Replace an existing variable with the value of a different one.

Each of these possibilities need to be handled, and some even have error cases, for example, the variable name in the righthand side of the line does not exist, hence the assignment is aborted.

Firstly, the name and value of the variable are retrieved from the line, and the name is checked to belong to an existing variable. This is done by attempting to retrieve a variable with the same name. Needless to say, if

something is returned, then the name belongs to an existing variable. If nothing is returned, then a new variable should be created.

Considering something is returned, the first character of the right hand side is checked to be a `$`, the symbol seperating a variable from a normal string. If it isn't, then simply set the variable value to the string as normal. However if it is, then the variable with the name after `$` is retrieved. If said variable does not exist, then the function displays an error, along with returning immediately. However if the variable exists, the value of the old variable is overwritten with the value of the new one. After doing so, the function returns with a successful code.

After doing so, possibilities 3 and 4 have been handled. Now it's time to handle the case when a new variable should be created.

In the case that the first character of the right hand side is found to be `$`, the same thing as before happens, only the function does not return at the end, and the value overwritten is the right hand side of the line.

After doing so, the new variable is initialised by allocating memory to the empty space in the `variable array`. Then, the now initialised empty space in the array will be set to hold the newly created variable. The following code will then reallocate enough additional memory to the `variable array` to be able to hold an additional variable the next time around, and the `integer variable` showing the amount of variables in the array is incremented.

After doing so, possibilities 1 and 2 have now also been handled. All that is left is the next block of code, which checks whether the `$EXITCODE` variable exists, and sets it if it does. This is important as at the start of the eggshell, when it is being initialised, numerous variable will be initialised before the `EXITCODE`. If the function tries to set the exitcode before the exitcode is even created, that would cause a segmentation fault.

2.7.4 void initShellVars
(char *ex)

This function serves to initialise the environment variables that will be used and stored by the eggshell . Said variables occupy the first 8 spaces of the variables array, making it easy to differentiate between the environment variables and the user-made variables.

The first thing the function does is initialise the SHELL and the CWD variables. This is because they have distinct ways of initialising them when compared to the other variables. SHELL utilises the personal getExecPath function, whereas CWD uses the inbuilt getcwd function.

After doing this, the variables array is initialised, in order to start storing the shell variables. Since the createVar expects a formatted string of <VARNAME>=<VALUE> , we'd need to pass a preformatted string as the parameter. To do so, 8 variables are initialised.

Using sprintf and functions to retrieve the rest of the environment variables, the 8 *injection strings* are formatted into a string that will be processed by createVar . After doing so, all 8 strings are passed to the function in order to create the variables. Once this is finished, all temporary strings are then freed.

2.7.5 char** environEGG
()

This function returns a character array of all environment variables, including the ones from the shell and eggshell. It does this by retrieving extern char **environ and using setenv in order to place our variables in environ . setenv was used over getenv as the latter uses the **pointer** to the string, complicating matters the same string was used to place multiple variables.

2.7.6 Retrieval and `value` functions

The following functions are used to retrieve either a variable struct, mostly for modification, or the value of the variable:

- `int varExists(char *varname)`
- `Var* retrieveVar(char *varname)`
- `char *value(char *varname)`

The first function serves as a helper to the other two functions. What it does is check whether a variable with the name `varname` exists in the array, and returns its index, with `-1` being returned if no such variable exists. This is done using a simple loop.

The second function uses the first to check whether a variable with that name exists, and using the index returned, either the variable is returned, or `0` is. In other words, `retrieveVar` is a more useful version of `varExists` for other functions.

The third function uses the second to check whether a variable with said name exists, and using the variable returned, returns its value. If such a variable does not exist, `0` is returned instead.

Each function is a step over the other, which allows the user to pick the right one depending on what they need precisely.

2.7.7 Display functions

There are two functions that display variables. These were seperate for the `all` and `vars` commands to have a simple way of executing:

- `void showShellVars()`
- `void displayUserVars()`

The former was done using a combination of `printf` and the `value` functions. The latter consisted of a loop that traversed the variable array while also excluding the environment variables.

2.7.8 Get, Set, and Update functions

The purpose of these functions is to either **get** a value, in order to initialise the shell variables, **set** a value easily, as the value would be modified frequently, and **update** a value, due to them requiring frequent updates for numerous reasons. Such functions include

- `char* getExecPath(char *ex)`
- `void setExitcode(int ec)`
- `void updateCWD()`
- `void updatePrompt()`

The first function utilises the parameter, and the current working directory, to construct the path that the `eggshell` is being executed in.

The parameter is meant to be `argv[0]`, which is the exact command used to launch the `eggshell`. Depending on whether the relative path *(./eggshell)* or the absolute path *(/???/eggshell)* was used, the path is then constructed by either keeping the absolute path, or concatenating the `CWD` with the relative path. After doing this, `realpath` is used to resolve the path, in case relative symbols remain in it.

The second function converts the integer given in the parameter to a string, and then sets the value of the `$EXITCODE` variable to it.

The third and fourth function \reinitialise" the `$CWD` and `$PROMPT` variables using similar means than what was used before to first initialise them. These are important to update as `$CWD` needs to update to reflect the effects of `chdir`, and `$PROMPT` needs to update to reflect the changes in `$CWD` and `$EXITCODE`.

2.8 pipe_manager.c

2.8.1 Use of pipe_manager.c

This file manages everything to do with the piping functionality that the program offers. This includes parsing a line for pipes, and then piping the output of one command into the piping of the input. As a result, only two functions are present, one for the parsing, and another for the actual piping mechanic.

2.8.2 int pipe_parser (char *line)

This function parses a line to detect any piping symbols, and uses the specialized `piping execute` function if such symbols are detected.

The detection is as simple as using `strstr` to detect `|`, returning `1` if the function returns a null pointer. If it doesn't fail however, the line given will then be split according to the pipe symbols, placing them in an array of strings which are then passed onto the `execute` function.

The parse function aids the execution function as well, in that it sets global variables, showing how many pipes/commands there are within the line.

2.8.3 int pipe_executer (char **commands)

This function requires an array of commands given by the `parsing` function. One could fake an array, but the function will still fail due to the global variables not being initialized.

```
59 int pipefd[pipeAmnt*2];
60
61 // Creates pipes out of the array of integers.
62 for(int i = 0; i < pipeAmnt; i++){
63     if(pipe(pipefd + i*2) < 0){
64         perror("piping");
65         _exit(-1);
66     }
67 }
```

The following crafts an array of integers, which will be used to initialize pipes using the `pipe` function. An array of specialized `pipeAmnt*2` is required due to every pipe requiring an *input* and an *output*. Meaning that for a command with 3 pipes, an array of six integers is crafted, `piping` every two spaces.

After this, three important variables are declared. `j` is used to iterate through all the commands, `p` is used to iterate through all the pipes within the array, and `pid` is used to store the process id of the current process.

The first loop then starts, used to iterate through all the commands. What follows is then command specific - the current command is split into the command itself and its arguments, which are placed in a single string. After this, the string is copied into a seperate variable, before being split up and stored in a string array to be used as arguments.

A process is then forked, and a conditional loop follows that seperates the child from the parent.

Child – Forking

```
106 /* If it isn't the last command,
107    then replace the current output pipe with stdout */
108 if(j != cmdAmnt-1){
109     if(dup2(pipefd[p+1], 1) < 0){
110         perror("dup2 piping output");
111         _exit(-1);
112     }
113 }
114 
115 /* If it isn't the first command,
116    then replace the last input pipe with stdin */
117 if(j != 0){
118     if(dup2(pipefd[p-2], 0) < 0){
119         perror("dup2 piping input");
120         _exit(-1);
121     }
122 }
123 
124 /* Close all current pipes */
125 for(int i = 0; i < 2*pipeAmnt; i++){
126     close(pipefd[i]);
127 }
```

The following blocks are complicated. Firstly, the command is checked to be either the **first** or the **last** command in the array. If it is the *first*, then only its pipe's output is wired to its `stdout`. If it is the *last*, then only the previous's pipe's input is wired to its `stdin`. If it is neither, then both of these steps occur.

To explain further, lets say we have 3 commands, `a`, `b` and `c`.

a outputs `"Hello World! "`. This is wired to the `stdout` of `PIPE 1`. b takes the input, and repeats it. It gathers its input from the `stdin` of `PIPE 1`, and places it into the `stdout` of `PIPE 2`. In the previous pipe, `"Hello World! "` remains, so that is used as input. c then takes the input, and displays how many characters there are. It gathers the input from `PIPE 2`, which contains the output of `b`.

This is effectively how the piping mechanism presented works. However, after the `stdout` and `stdin` of the current process are set to the pipes, the pipes are now no longer relevant to the process. As a result, they are now **closed.**

After this, the command is then executed. Since the command could also be any of our internal commands, the command is then parsed through to check whether it is internal. If it is, then the prospective function is run as a result. If it isn't however, `execve` is used to execute the external command.

The parent block only waits until the status of the current child is gathered, at which point the `$EXITCODE` shell variable is set to it.

Before exiting the loop, two things happen. The array of arguments, and the array of paths are both freed, as they were only useful for the execution of the current function, and the `p` variable is incremented by 2, hence moving onto the next pipe.

Once the loop exits, all the pipes are then closed, as the piping has stopped, and the `wait` function is used in order to wait for any remaining processes that are still active.

Chapter 3

Testing the *eggshell*