CPS1012 – Operating Systems and Systems Programming 1 – Assignment Report

1. Design of the System

The Tiny Shell (tish) program is designed to provide a small, simplified command shell interface which is very similar to a regular Unix command shell. It allows users to execute various pipeline and built in commands, such as navigate the file system, access details about the shell program, print certain messages, and even install certain packages. The design follows a basic architecture with components responsible for Command Parsing, Command Execution, and Process Control.

1.1. System Architecture

This program follows a single-threaded architecture with the following components: Command Parser, Command Executor, and Process Manager. The Command Parser's job is to handle user input, tokenizes commands according to the user input, and identifies the appropriate command and arguments so that the process can be executed in a correct manner. The Command Executor then receives the parsed commands and executes them by invoking the corresponding system utilities or external programs. The Process Manager's role is keep track of any active processes and provide functionalities for process control.

1.2. Data Flow

Data flow in this Tiny Shell program is very simple and straightforward to understand. Any input from the user is passed from the Command Parser to the Command Executor, which processes the command and may interact with the Process Manager as needed. The output of executed commands is displayed to the user.

1.3. System Components

The major components of the system include:

Command Parser: This is responsible for tokenizing user input, identifying commands and arguments, and validating syntax. It also provides the necessary data structures to represent the parsed commands.

Command Executor: This receives the parsed commands from the Command Parser and executes them by invoking the appropriate system utilities or external programs. It also captures the output and any error streams and provides them to the user.

Process Manager: This is responsible for managing any active processes, including process creation, termination, and monitoring. It keeps track of process IDs, statuses, and provides utilities for process control, such as background execution or process termination.

1.4. System Constraints and Assumptions

The Tiny Shell program assumes a Unix-like environment with access to standard system utilities and commands.

It also assumes a single-threaded execution model, where only one command can be processed at a time.

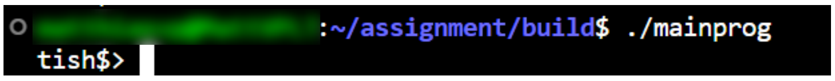
The final program for each task is assumed to be taken from the program of the last part from the respective task. So for instance, task1.c is taken from 1e.c, task2.c is taken from 2b.c, task3.c is taken from 3d.c, and task4.c is taken from 4c.c. The program maintish.c is assumed to execute task4.c, as it is built from the previous tasks and includes all the intended working functionality of the full shell program.

The system is designed to handle basic command execution and process management, including the functionality for input/output redirection and pipelines. The program does not contain any functionality for very advanced features like shell scripting or complex command-line processing.

2. How the System was Tested

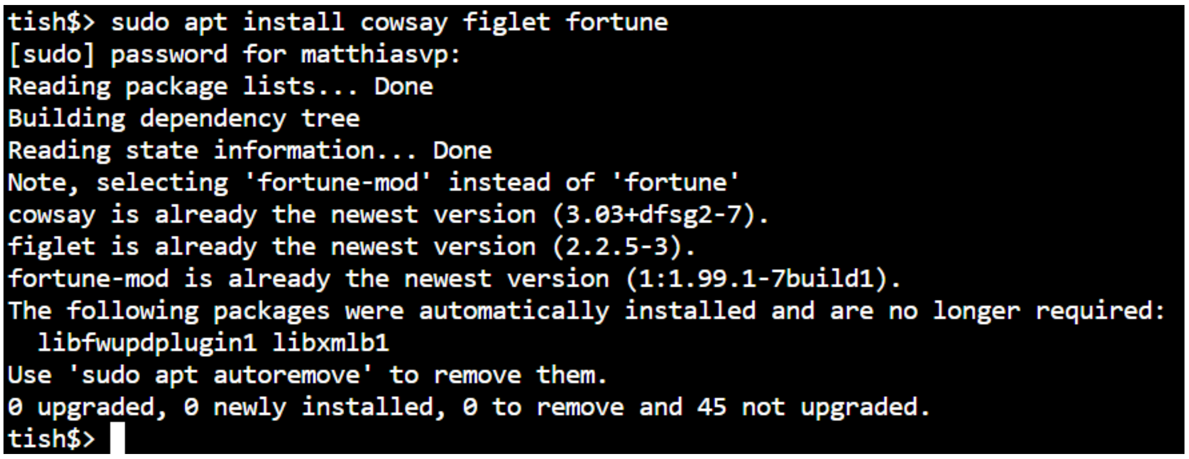
All the tests were performed on maintish.c, as it contains all the functionality from every task.

As expected, whenever the program is called by typing the following command: **./mainprog**, the program begins execution by printing out **tish$>**, signalling that the shell is ready to accept any commands. An example of this working can be seen below:



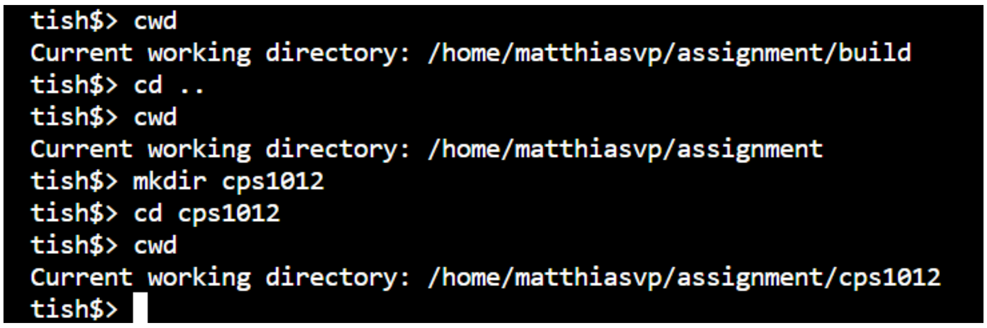
```
o [redacted]:~/assignment/build$ ./mainprog
  tish$>
```

To perform some of the remaining tests, the **cowsay**, **figlet** and **fortune** packages were installed. The installation process was done in the tiny shell program by typing in the following command: **sudo apt install cowsay figlet fortune**. This was done to test the program ability to accept installation commands. As expected, the program installs these three packages just like the regular Linux shell. However, for this case, since they were already installed, the process will say that no new packages were installed. An example of this can be seen below:



```
tish$> sudo apt install cowsay figlet fortune
[sudo] password for matthiasvp:
Reading package lists... Done
Building dependency tree
Reading state information... Done
Note, selecting 'fortune-mod' instead of 'fortune'
cowsay is already the newest version (3.03+dfsg2-7).
figlet is already the newest version (2.2.5-3).
fortune-mod is already the newest version (1:1.99.1-7build1).
The following packages were automatically installed and are no longer required:
  libfwupdplugin1 libxmlb1
Use 'sudo apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 45 not upgraded.
tish$>
```

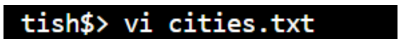
A new directory called cps1012 was also created to store any test files that are created for these tests. This procedure was also done in the shell program to test the program's ability to create new files. The **cd** and **cwd** commands, which are designed as built in commands, were also used throughout this process in order to access the newly created directory. More information about testing the built in commands is explained later. This whole process can be seen below:



```
tish$> cwd
Current working directory: /home/matthiasvp/assignment/build
tish$> cd ..
tish$> cwd
Current working directory: /home/matthiasvp/assignment
tish$> mkdir cps1012
tish$> cd cps1012
tish$> cwd
Current working directory: /home/matthiasvp/assignment/cps1012
tish$>
```

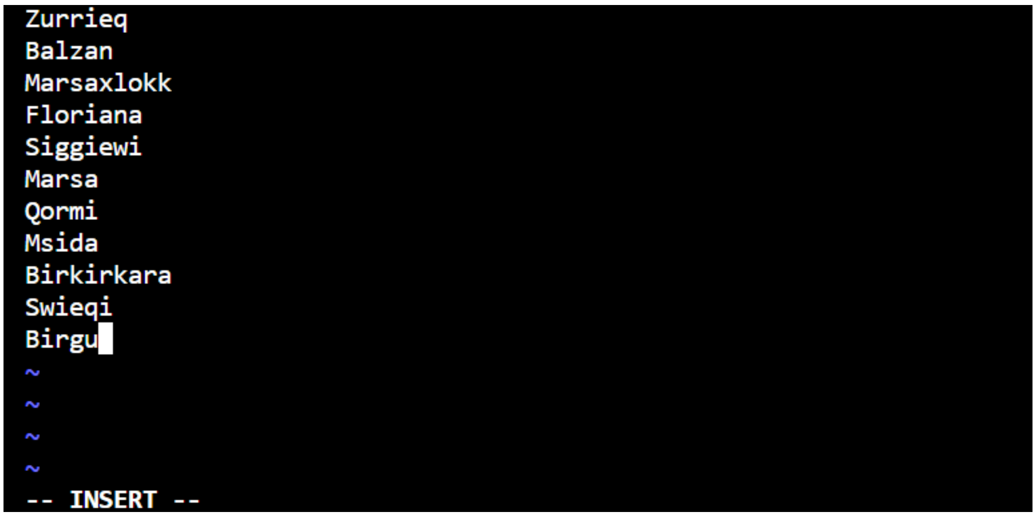
As a setup to the new folder, a file called cities.txt was created using the Tiny Shell program, by entering the command **vi cities.txt**. This file contains a list of some Maltese cities, which includes the following city names in this order: Zurrieq, Balzan, Marsaxlokk, Floriana, Siggiewi, Marsa, Qormi, Msida, Birkirkara, Swieqi, Birgu. Each city is entered on a separate line. Again, this process was done in the shell program to test the program's ability to create a new file and add text to it, just like the regular shell. This ended up working just as expected. This can be seen below:

<ins>Entering the command</ins>



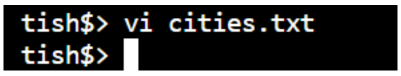
```
tish$> vi cities.txt
```

<ins>Entering the names of the cities</ins>



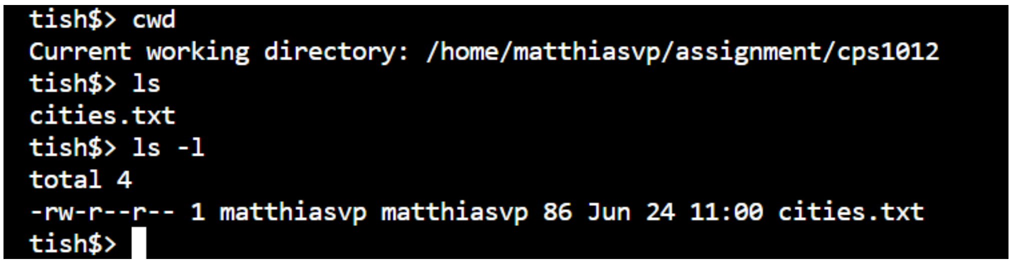
```
Zurrieq
Balzan
Marsaxlokk
Floriana
Siggiewi
Marsa
Qormi
Msida
Birkirkara
Swieqi
Birgu
~
~
~
~
-- INSERT --
```

After exiting the **vi** editor



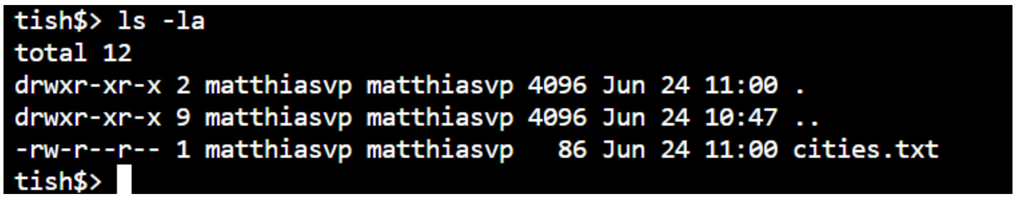
```
tish$> vi cities.txt
tish$>
```

To ensure that the file is present in the cps1012 directory, the commands **ls** and **ls -l** was entered in the program, with the current working directory being situated in the cps1012 folder. This was also performed to test the functionality from Task 1. As expected, the program found the new file in the folder. An example of this working can be seen below:



```
tish$> cwd
Current working directory: /home/matthiasvp/assignment/cps1012
tish$> ls
cities.txt
tish$> ls -l
total 4
-rw-r--r-- 1 matthiasvp matthiasvp 86 Jun 24 11:00 cities.txt
tish$>
```

Like the **ls -l** command, the **ls -la** command was also tested on the cps1012 folder to make sure that only the new file is detected inside the folder. This also worked as expected, as seen below:

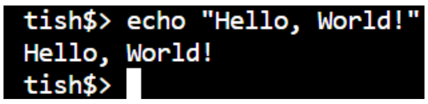


```
tish$> ls -la
total 12
drwxr-xr-x 2 matthiasvp matthiasvp 4096 Jun 24 11:00 .
drwxr-xr-x 9 matthiasvp matthiasvp 4096 Jun 24 10:47 ..
-rw-r--r-- 1 matthiasvp matthiasvp   86 Jun 24 11:00 cities.txt
tish$>
```

2.1. Command Pipeline Tests

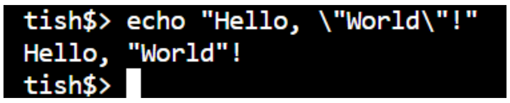
With everything set up, the next tests were about the Command Pipeline functionality which was implemented in Task 1. All pipeline commands, except the **ls** and **vi** commands, were tested, as the two previous commands was already tested previously. The goal for all these tests was to make sure that the commands would execute just like how they would in the regular Linux command shell.

First, the **echo** command was tested by typing the command **echo “Hello, World!”**. This worked just as expected, with the program returning the phrase “Hello, World!” without the double quotation characters. This can be seen below as follows:



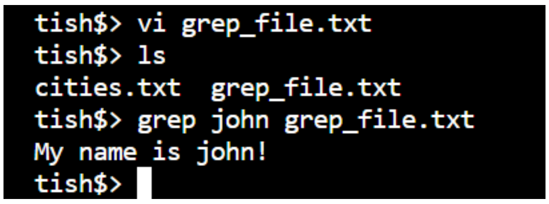
```
tish$> echo "Hello, World!"
Hello, World!
tish$>
```

The same thing was done, but this time two escape characters \” were inputted between the word “World”. This was done to test the program’s ability to also return the double quotation marks, just like in the regular shell. This also worked as expected, with the returning phrase being “Hello, “World”!”, without the beginning and end quotation marks. This can be seen below as follows:



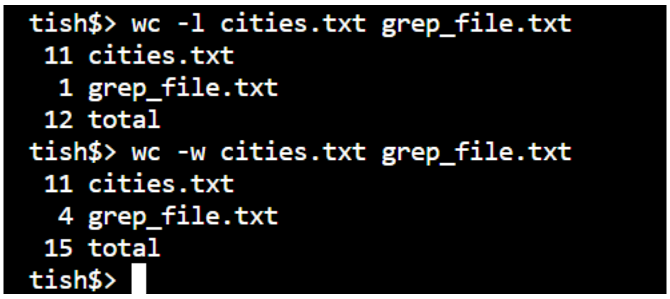
```
tish$> echo "Hello, \"World\"!"
Hello, "World"!
tish$>
```

The **grep** command was tested next. For this test, a new file called grep_file.txt was created, which contains the phrase “My name is john!”. This process was done in the shell program. Afterwards, the command **grep john grep_file.txt** was entered. This ended up working as expected, with the program returning the same phrase which included the word “john”. This can be seen below as follows:



```
tish$> vi grep_file.txt
tish$> ls
cities.txt  grep_file.txt
tish$> grep john grep_file.txt
My name is john!
tish$>
```

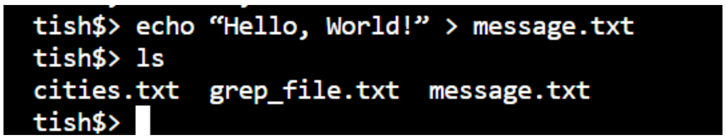
Afterwards, the command **wc -l** and **wc -w** were tested on the two files cities.txt and grep_file.txt. The first command returns the number of lines in each file whilst the second one returns the number of words. Both ended up working as expected. This can be seen below:



```
tish$> wc -l cities.txt grep_file.txt
 11 cities.txt
  1 grep_file.txt
 12 total
tish$> wc -w cities.txt grep_file.txt
 11 cities.txt
  4 grep_file.txt
 15 total
tish$>
```

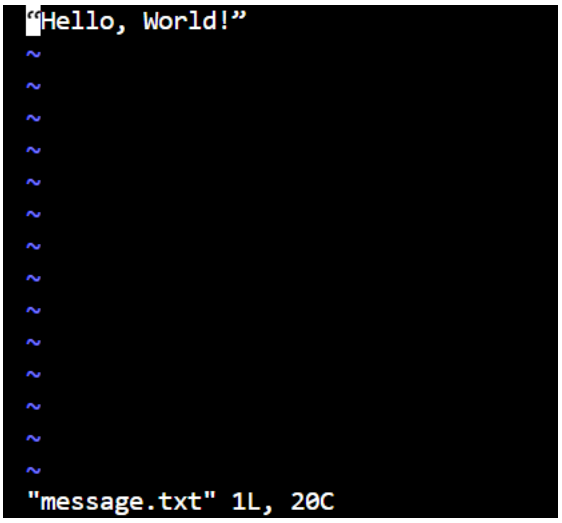
Once all these commands were tested successfully, the input/output redirection functionality for Task 3 was tested. Output redirection with the symbol ">" was first tested. The first test was done by entering the command **echo "Hello, World!" > message.txt** in the shell program. The goal of this is to put "Hello, World!" in a new file called message.txt. As expected, this ended up creating a new file called message.txt, since it did not exist in the cps1012 folder at the time of the test, with the phrase "Hello, World!" displayed. However, the double quotation marks happened to also show up in the file, which was not expected to happen. This can be seen below:

<u>Entering the command and checking the directory</u>



```
tish$> echo "Hello, World!" > message.txt
tish$> ls
cities.txt  grep_file.txt  message.txt
tish$>
```

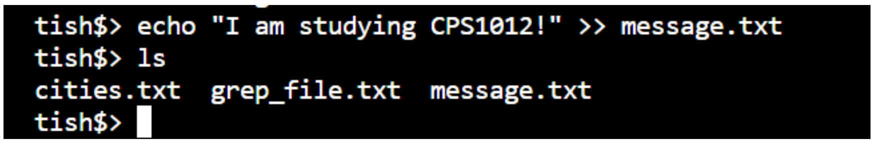
<u>Checking the message.txt file</u>



```
"Hello, World!"
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"message.txt" 1L, 20C
```

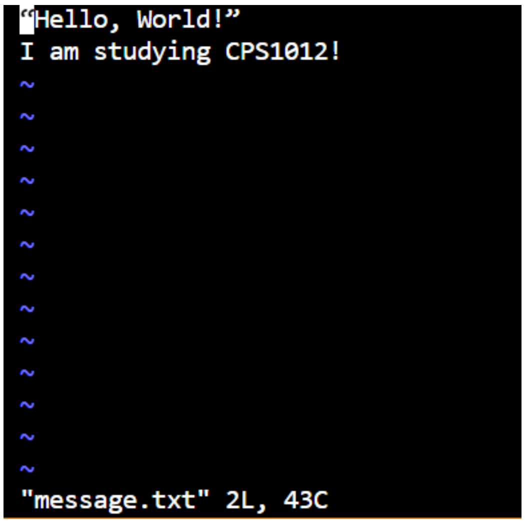
A similar operation was done, but this time to test append redirection with the symbol ">>" by entering the command **echo "I am studying CPS1012!" >> message.txt** in the shell program. The goal of this is to put "Hello, World!" in a new file called message.txt. As expected, this added the phrase "I am studying CPS1012!" after the phrase "Hello, World!" in a new line. However, unlike with output redirection the double quotation marks did not show up. This can be seen below:

<u>Entering the command and checking the directory</u>



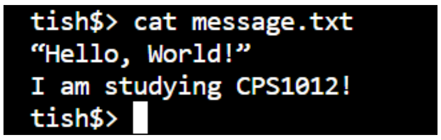
```
tish$> echo "I am studying CPS1012!" >> message.txt
tish$> ls
cities.txt  grep_file.txt  message.txt
tish$>
```

<u>Checking the message.txt file</u>



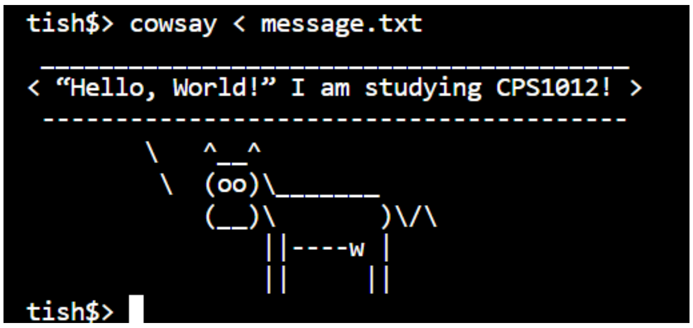
```
"Hello, World!"
I am studying CPS1012!
~
~
~
~
~
~
~
~
~
~
~
~
~
"message.txt" 2L, 43C
```

Before testing input redirection, the **cat** command was tested by entering **cat message.txt** in the shell program. The goal of this is to return the contents of message.txt. As expected, the program returned the two phrases that were added to the file using output redirection, both using the ">" and ">>" symbols. The result of this is shown below as follows:



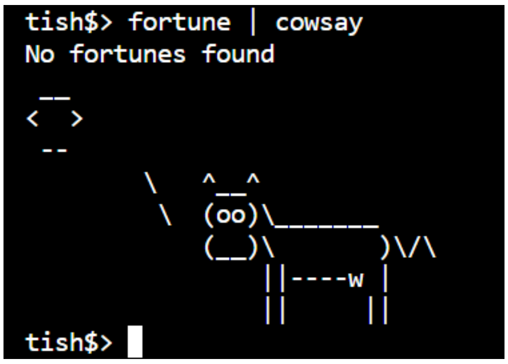
```
tish$> cat message.txt
"Hello, World!"
I am studying CPS1012!
tish$>
```

When it was complete, input redirection with the symbol "<" was tested by entering the following command in the shell program: **cowsay < message.txt**. This uses the cowsay package that was installed previously. As expected, the program printed out the contents inside message.txt and printed out a cow, exactly like what happens when using the cowsay package. However, the contents were printed on a single line, not on two separate lines. At first, I was unsure if that was what supposed to happen. However, after testing it using the main shell, I found out that it worked perfectly fine. This can be seen below:

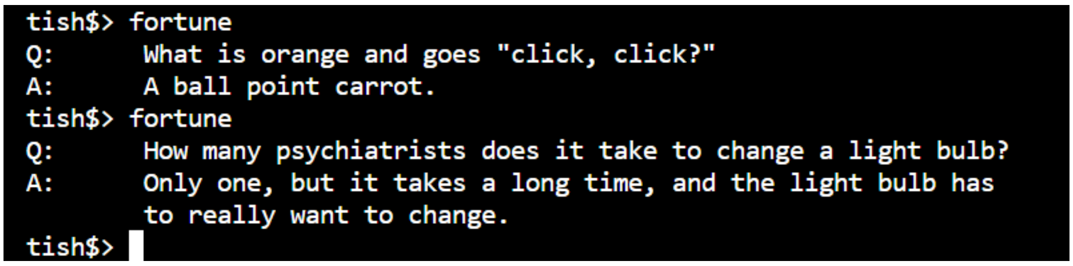


```
tish$> cowsay < message.txt
 ________________________________________
< “Hello, World!” I am studying CPS1012! >
 ----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
tish$>
```

Next, the pipe functionality was tested. This was done by typing the command **fortune | cowsay**. This uses the fortune and cowsay packages, both of which were installed previously using the shell program. As expected, both operations were performed. However, for the **fortune** command, the statement “No fortunes found” was returned, which was unexpected. Two **fortune** tests were done to validate the previous command. As expected, these two operations returned different fortunes. This was a small error that I could not get to fix on time. The result of this can be seen below:



```
tish$> fortune | cowsay
No fortunes found
 __
<  >
 --
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
tish$>
```



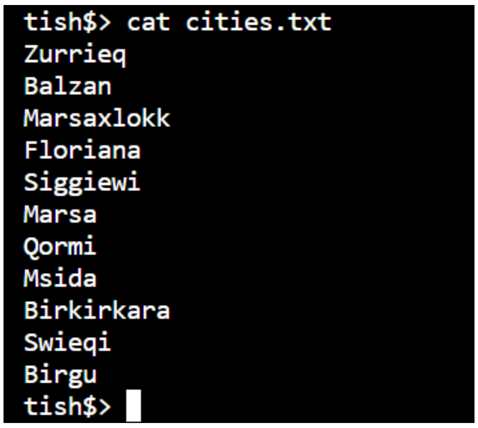
```
tish$> fortune
Q:      What is orange and goes "click, click?"
A:      A ball point carrot.
tish$> fortune
Q:      How many psychiatrists does it take to change a light bulb?
A:      Only one, but it takes a long time, and the light bulb has
        to really want to change.
tish$>
```

A couple more tests were performed for the pipe functionality, but they resulted in some errors, which will be discussed more in the next section.

The last few tests involved the use of the cities.txt file created previously and input/output redirection. These tests also used the sort operation.

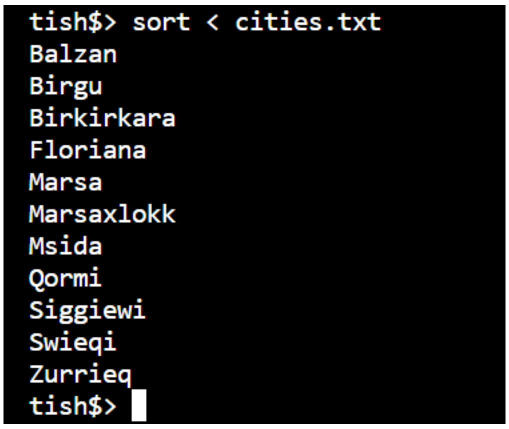
Firstly, the contents of the cities.txt file were successfully printed out to the shell program using the **cat** command. Afterwards, the command **sort < cities.txt** was entered. The aim of this is to sort the names of the cities in alphabetical order, without saving the operation. This ended up working as expected, with the program returning the sorted list of city names. This can be seen below:

Accessing the city names



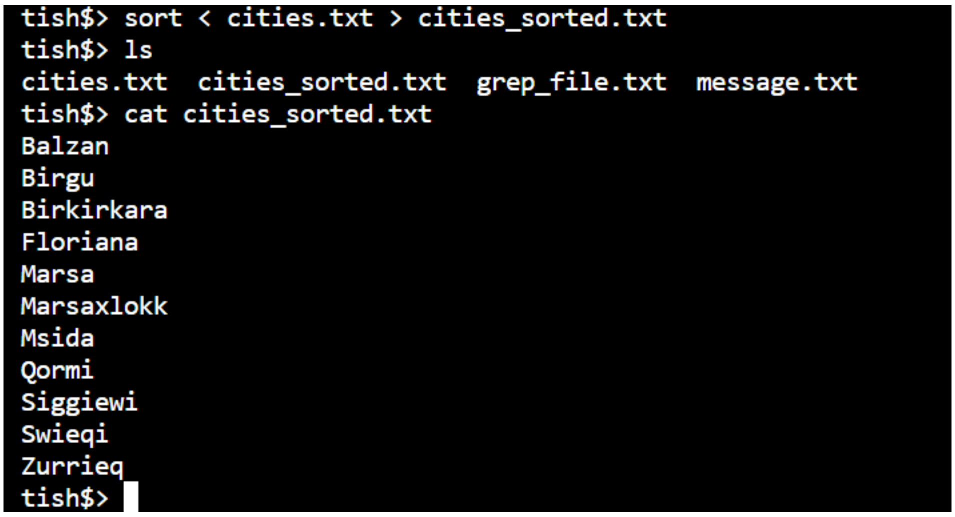
```
tish$> cat cities.txt
Zurrieq
Balzan
Marsaxlokk
Floriana
Siggiewi
Marsa
Qormi
Msida
Birkirkara
Swieqi
Birgu
tish$>
```

Entering the command



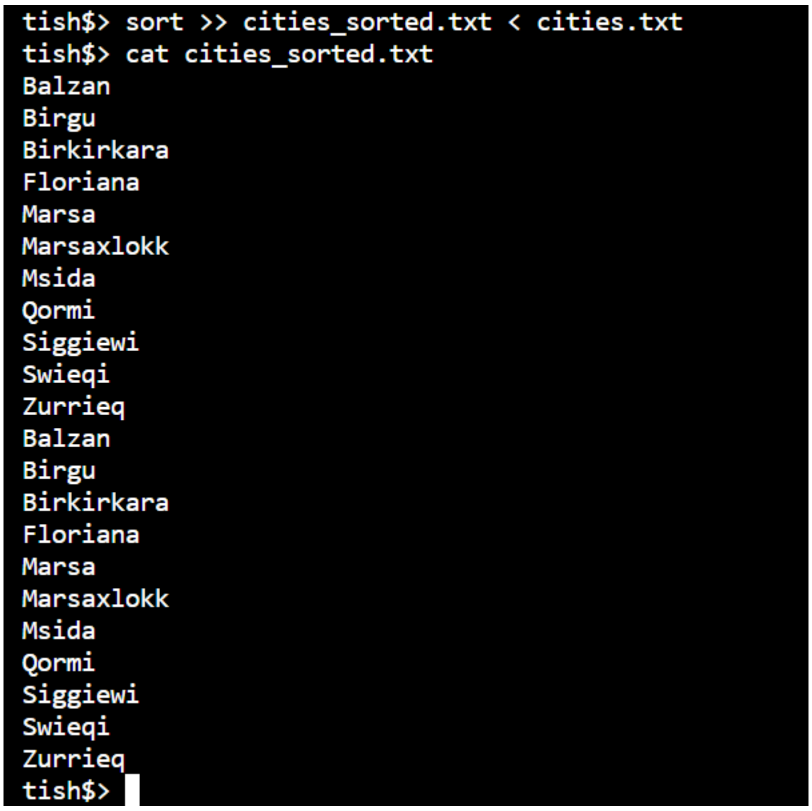
```
tish$> sort < cities.txt
Balzan
Birgu
Birkirkara
Floriana
Marsa
Marsaxlokk
Msida
Qormi
Siggiewi
Swieqi
Zurrieq
tish$>
```

A similar operation was performed but this time the command **sort < cities.txt > cities_sorted.txt** was entered in the shell program. Its objective is to perform the sort operation as done previously, but then store the sorted list in a new file called cities_sorted.txt. This was done successfully, with a **cat** operation on the new file returning the sorted list of cities. The result of this can be seen below:



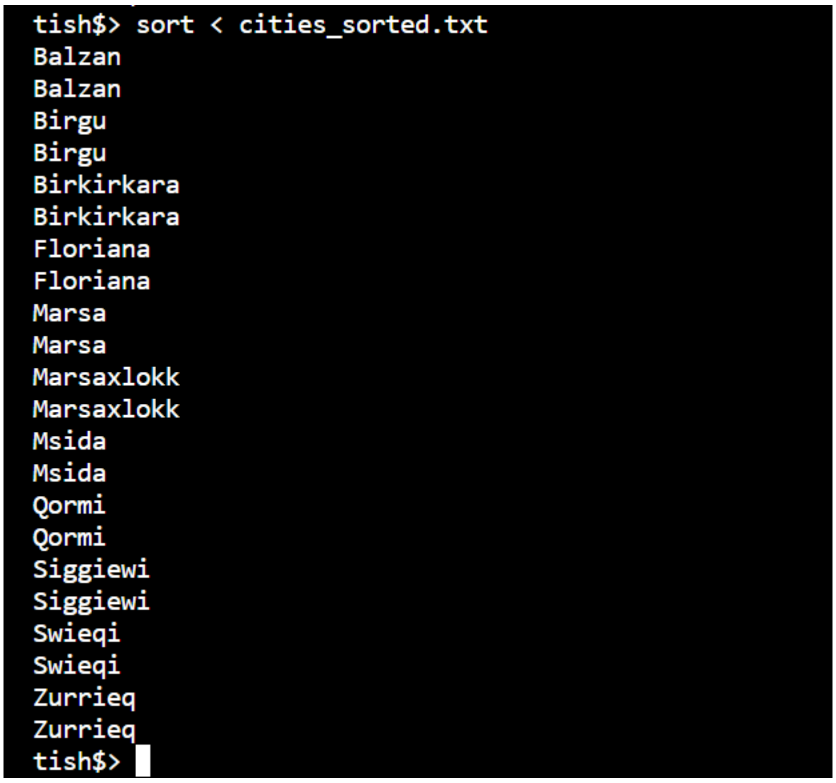
```
tish$> sort < cities.txt > cities_sorted.txt
tish$> ls
cities.txt  cities_sorted.txt  grep_file.txt  message.txt
tish$> cat cities_sorted.txt
Balzan
Birgu
Birkirkara
Floriana
Marsa
Marsaxlokk
Msida
Qormi
Siggiewi
Swieqi
Zurrieq
tish$>
```

Afterwards, the command **sort >> cities_sorted.txt < cities.txt** was entered. Its function is to basically duplicate the contents of the cities_sorted.txt file. This proved to work just as expected, with a **cat** operation on the file showing that there are two instances of the same city names. The result is shown below:



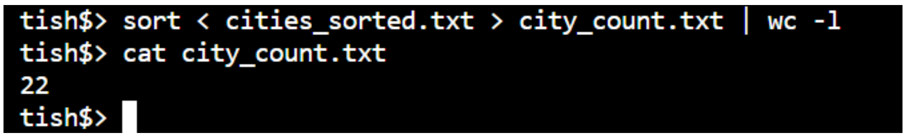
```
tish$> sort >> cities_sorted.txt < cities.txt
tish$> cat cities_sorted.txt
Balzan
Birgu
Birkirkara
Floriana
Marsa
Marsaxlokk
Msida
Qormi
Siggiewi
Swieqi
Zurrieq
Balzan
Birgu
Birkirkara
Floriana
Marsa
Marsaxlokk
Msida
Qormi
Siggiewi
Swieqi
Zurrieq
tish$>
```

Then, a very similar operation was done to the first sort test, where the command **sort < cities_sorted.txt** was entered in the shell. This basically sorts the new contents inside cities_sorted.txt. This worked as expected. The result of this is shown below:



```
tish$> sort < cities_sorted.txt
Balzan
Balzan
Birgu
Birgu
Birkirkara
Birkirkara
Floriana
Floriana
Marsa
Marsa
Marsaxlokk
Marsaxlokk
Msida
Msida
Qormi
Qormi
Siggiewi
Siggiewi
Swieqi
Swieqi
Zurrieq
Zurrieq
tish$>
```

Pipe functionality was tested next, through the use of the following command **sort < cities_sorted.txt > city_count.txt | wc -l**. This basically returns the number of lines inside cities_sorted.txt and stores the result in a new file called city_count.txt. This worked just as expected. The result of this is shown below:



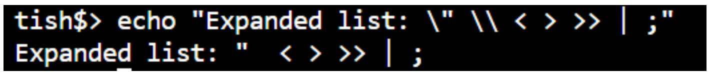
```
tish$> sort < cities_sorted.txt > city_count.txt | wc -l
tish$> cat city_count.txt
22
tish$>
```

A couple of other commands involving the functionality of pipes and sort were tested, but these led to some errors relating to input/output redirection. These will be discussed in the next section.

2.2. Advanced Scanning Tests

The next set of tests was about the Advanced Scanning functionality which was implemented in Task 4. This involved the use of all possible pipeline commands as well as some special symbols which are "|", "<", ">", ">>" and ";".

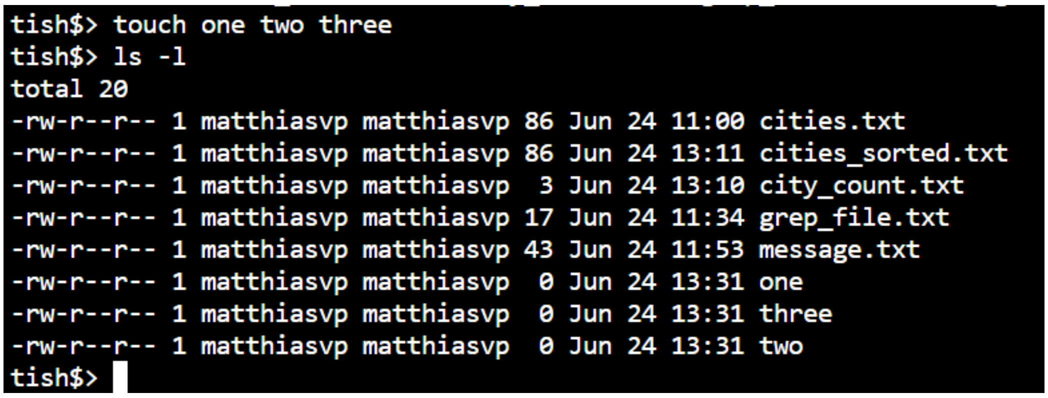
An **echo** operation was put into practice by typing the command **echo "Expanded list: \" \\ < > >> | ;"** into the shell program. As expected, the program returned the whole phrase, including the special symbols and the escape character, without the beginning and end double quotation characters. The result of this is shown below:



```
tish$> echo "Expanded list: \" \\ < > >> | ;"
Expanded list: "  < > >> | ;
```

A few other commands were also tested but these led to some serious bugs that I unfortunately was not able to fix. This will be discussed in further detail the next section.

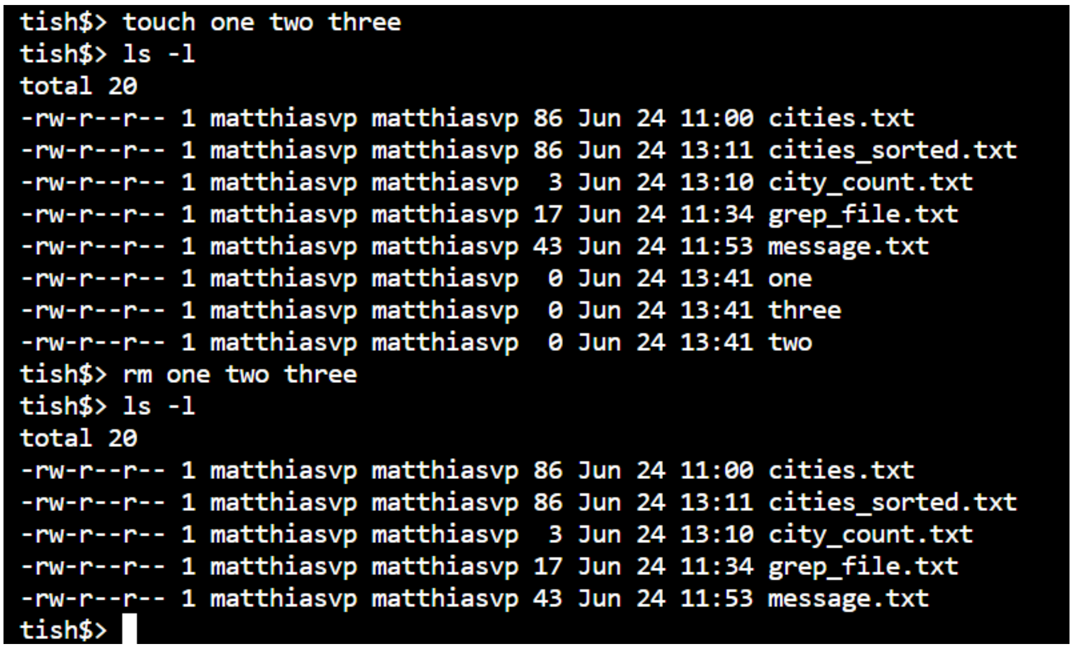
The next few tests involved the use of the **touch** pipeline command. The first test that was performed was done by typing in the command **touch one two three**. This creates three separate files called "one", "two" and "three" respectively. This ended up working just as expected. The result from the above command can be seen below:



```
tish$> touch one two three
tish$> ls -l
total 20
-rw-r--r-- 1 matthiasvp matthiasvp 86 Jun 24 11:00 cities.txt
-rw-r--r-- 1 matthiasvp matthiasvp 86 Jun 24 13:11 cities_sorted.txt
-rw-r--r-- 1 matthiasvp matthiasvp  3 Jun 24 13:10 city_count.txt
-rw-r--r-- 1 matthiasvp matthiasvp 17 Jun 24 11:34 grep_file.txt
-rw-r--r-- 1 matthiasvp matthiasvp 43 Jun 24 11:53 message.txt
-rw-r--r-- 1 matthiasvp matthiasvp  0 Jun 24 13:31 one
-rw-r--r-- 1 matthiasvp matthiasvp  0 Jun 24 13:31 three
-rw-r--r-- 1 matthiasvp matthiasvp  0 Jun 24 13:31 two
tish$>
```

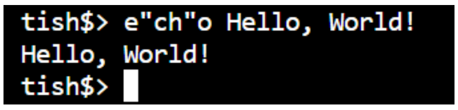
The next test was performed by typing in the following command: **touch "one two three"**. This is supposed to create a single file called "one two three". Unexpectedly, the shell program created three files, with the same names as above, instead of one. Obviously this is a small bug that could not be fixed on time. This will be explained in further detail later.

However, the **rm** command worked just as expected. An example of this working can be seen below:



```
tish$> touch one two three
tish$> ls -l
total 20
-rw-r--r-- 1 matthiasvp matthiasvp 86 Jun 24 11:00 cities.txt
-rw-r--r-- 1 matthiasvp matthiasvp 86 Jun 24 13:11 cities_sorted.txt
-rw-r--r-- 1 matthiasvp matthiasvp  3 Jun 24 13:10 city_count.txt
-rw-r--r-- 1 matthiasvp matthiasvp 17 Jun 24 11:34 grep_file.txt
-rw-r--r-- 1 matthiasvp matthiasvp 43 Jun 24 11:53 message.txt
-rw-r--r-- 1 matthiasvp matthiasvp  0 Jun 24 13:41 one
-rw-r--r-- 1 matthiasvp matthiasvp  0 Jun 24 13:41 three
-rw-r--r-- 1 matthiasvp matthiasvp  0 Jun 24 13:41 two
tish$> rm one two three
tish$> ls -l
total 20
-rw-r--r-- 1 matthiasvp matthiasvp 86 Jun 24 11:00 cities.txt
-rw-r--r-- 1 matthiasvp matthiasvp 86 Jun 24 13:11 cities_sorted.txt
-rw-r--r-- 1 matthiasvp matthiasvp  3 Jun 24 13:10 city_count.txt
-rw-r--r-- 1 matthiasvp matthiasvp 17 Jun 24 11:34 grep_file.txt
-rw-r--r-- 1 matthiasvp matthiasvp 43 Jun 24 11:53 message.txt
tish$>
```

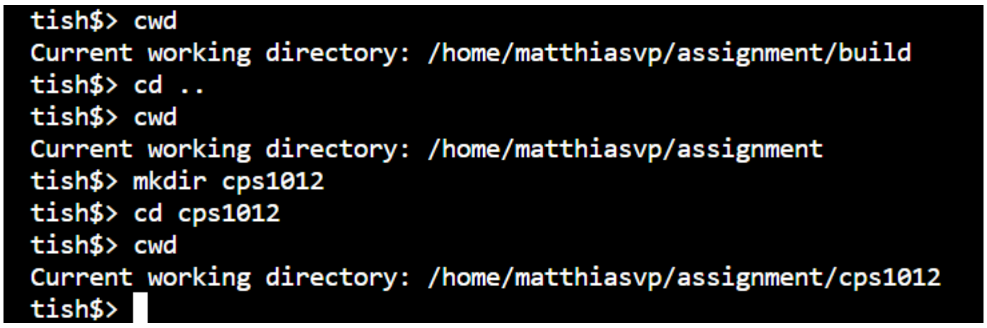
The last test performed for this sub-section was done to handle any strange commands. As an example, the following command was entered in the shell: **e"ch"o Hello, World!**. This ended up working just like a regular **echo** operation, with the program returning the phrase: Hello, World!. This can be seen below:



```
tish$> e"ch"o Hello, World!
Hello, World!
tish$>
```

2.3. Builtin Commands

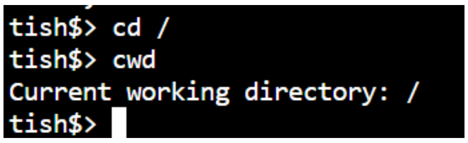
The next set of tests was performed to test the functionality of the built in commands that was implemented in Task 2. The **cd** and **cwd** built in commands were already partially tested previously. Below is an image which shows the use of both commands whilst setting up the cps1012 directory:



```
tish$> cwd
Current working directory: /home/matthiasvp/assignment/build
tish$> cd ..
tish$> cwd
Current working directory: /home/matthiasvp/assignment
tish$> mkdir cps1012
tish$> cd cps1012
tish$> cwd
Current working directory: /home/matthiasvp/assignment/cps1012
tish$>
```

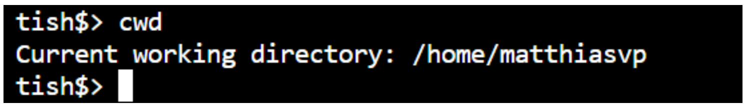
The **cd** command was further tested by typing in the commands **cd /** and **cd /home/USER**, where USER stands for the username of the Linux OS. The first command sends the user to the root directory and the second command sends the user to the home directory under the username being accessed. Both of these commands ended up working as expected. The results for both of these commands are shown below:

Going to root directory



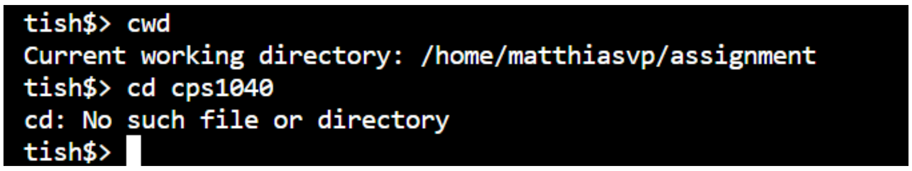
```
tish$> cd /
tish$> cwd
Current working directory: /
tish$>
```

Going to home directory under username



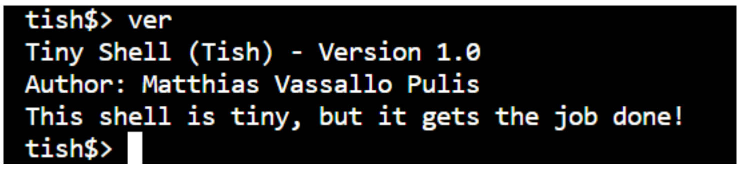
```
tish$> cwd
Current working directory: /home/matthiasvp
tish$>
```

The **cd** command was tested even further by trying to access a sub-folder which does not exist in the current folder being accessed. For example, with the current working directory being **home/USER/assignment**, the command **cd cps1040** was entered. As expected, the program returned with an error that there does not exist a directory under the name “cps1040”. The result of this is shown below:



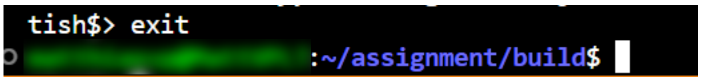
```
tish$> cwd
Current working directory: /home/matthiasvp/assignment
tish$> cd cps1040
cd: No such file or directory
tish$>
```

The next test was about the **ver** built in command. This was performed by simply typing in **ver** in the shell program. As expected, a bunch of lines were printed out giving details about the shell program itself. This is shown below as follows:



```
tish$> ver
Tiny Shell (Tish) - Version 1.0
Author: Matthias Vassallo Pulis
This shell is tiny, but it gets the job done!
tish$>
```

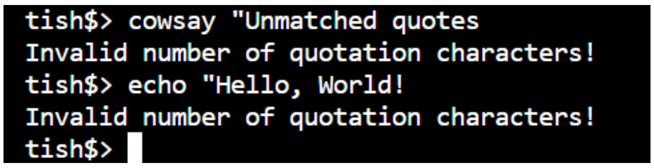
The last test done for this sub-section was done by entering **exit** in the program. This built in command's job is to exit the program and return to the main shell if the program was accessed using it. This worked as expected. The result for this can be seen below:



```
 tish$> exit
[redacted]:~/assignment/build$
```

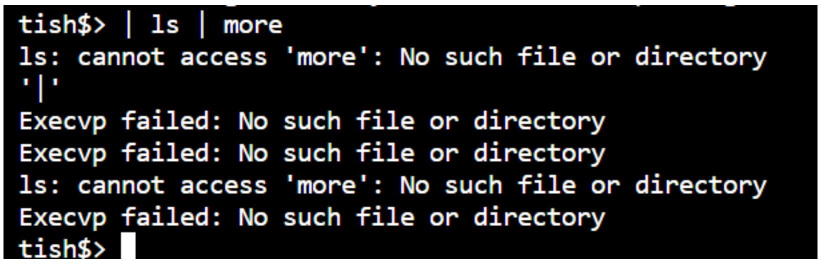
2.4. Error Handling

The last few tests were performed to handle any errors that might come out if the user inputs something wrong in the shell program. The first tests done were to test if there exist an odd number of double quotation characters. The two commands tested were **cowsay “Unmatched quotes** and **echo “Hello, World!**. As expected, an error message came up stating “Invalid number of quotation characters!” and both operations do not execute. The result for both commands can be seen below:



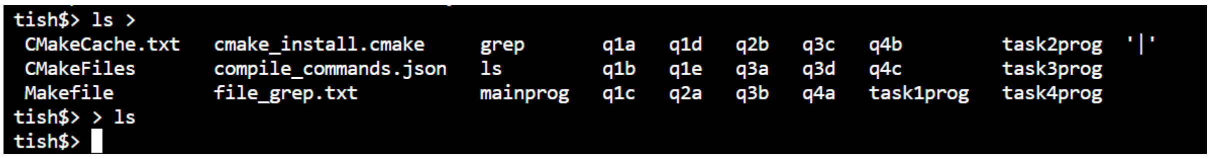
```
tish$> cowsay "Unmatched quotes
Invalid number of quotation characters!
tish$> echo "Hello, World!
Invalid number of quotation characters!
tish$>
```

The next command that was tested was **| ls | more**. This was done to test the program’s ability to realise that the first pipe does not have a left operation to execute, as it is an infix operator. Although the command did not execute, a bunch of unrelated error messages were displayed. The latter was not expected. This can be seen below:



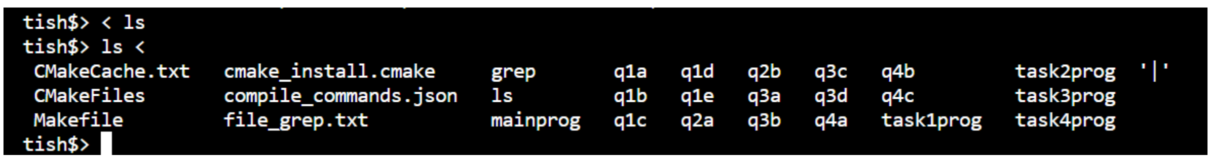
```
tish$> | ls | more
ls: cannot access 'more': No such file or directory
'|'
Execvp failed: No such file or directory
Execvp failed: No such file or directory
ls: cannot access 'more': No such file or directory
Execvp failed: No such file or directory
tish$>
```

The last two commands entered for this section were **ls >** and **> ls**. These are supposed to return an error stating that there are no right and left operators for each of these respective commands. However, no error was returned at all. Instead, the first operation from each command was executed with no problems. This was not expected at all. This was also a bug which could not be fixed on time. The results for both commands are shown below:



```
tish$> ls >
 CMakeCache.txt   cmake_install.cmake     grep       q1a   q1d   q2b   q3c   q4b         task2prog  '|'
 CMakeFiles       compile_commands.json   ls         q1b   q1e   q3a   q3d   q4c         task3prog
 Makefile         file_grep.txt           mainprog   q1c   q2a   q3b   q4a   task1prog   task4prog
tish$> > ls
tish$>
```

The same commands were later tested, but the “>” symbol was switched to a “<” symbol for both commands. The results were pretty much identical to the previous commands. This can be seen below:



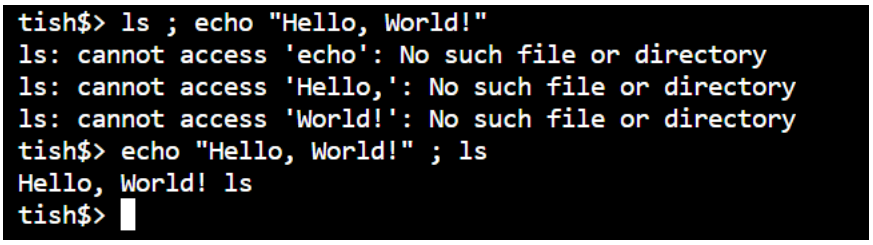
```
tish$> < ls
tish$> ls <
 CMakeCache.txt   cmake_install.cmake     grep       q1a   q1d   q2b   q3c   q4b         task2prog  '|'
 CMakeFiles       compile_commands.json   ls         q1b   q1e   q3a   q3d   q4c         task3prog
 Makefile         file_grep.txt           mainprog   q1c   q2a   q3b   q4a   task1prog   task4prog
tish$>
```

3. List of Bugs Present

Throughout testing of the program, various bugs were identified, some of which have been previously mentioned. However, there are others that were not touched in the previous section due to testing accuracy. This section's aim is to list all the bugs that are present in the Tiny Shell program and give an explanation as to why they occur.

3.1. Use of the ";" symbol

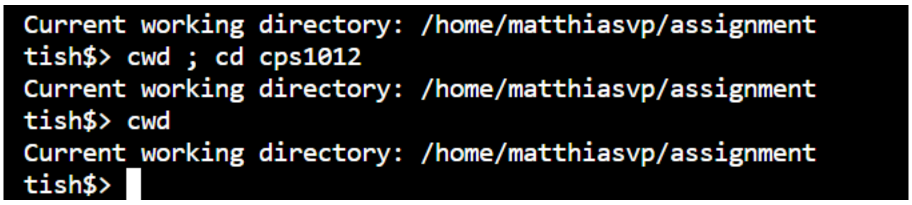
One significant bug present in the program is the use of the ";" character, whose aim is to separate to commands and process them one by one, similar to the pipe operation. However, this only works depending on the order of the operations entered by the user. For example, a ';' separating an **ls** and an **echo** operation in that order leads to some errors. However, if the **echo** operation is placed before the **ls** one, then the word "ls" is printed out by the **echo** operation, which is not what is supposed to happen. Therefore, the ';' character is partially functional. Below is an example demonstrating the above.



```
tish$> ls ; echo "Hello, World!"
ls: cannot access 'echo': No such file or directory
ls: cannot access 'Hello,': No such file or directory
ls: cannot access 'World!': No such file or directory
tish$> echo "Hello, World!" ; ls
Hello, World! ls
tish$>
```

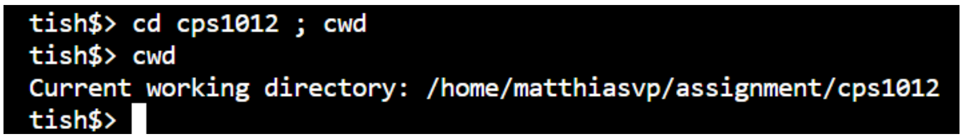
The same thing applies for the built in commands. However, in this situation, the ';' character appears to perform only the first operation. In the first example, the user types **cwd** before adding a **cd** operation, separating with a ';'. The result is that the **cwd** operation gets executed, and when the user types in **cwd** again, the same working directory is displayed. This appears to be the case as in the second example, the order of the operations was swapped, with the **cd** operation entered before **cwd**. The result was that the **cd** operation took place just fine, with the **cwd** check showing that the user is in a new directory. In the final example, the user types **ver** before **exit**, signalling that the user would like to view the program's details and then exit. However, only the **ver** operation takes place, resulting in the program still being executed. This again is not what is supposed to happen. Unfortunately, this bug could not be fixed on time. Below are three screenshots of the program demonstrating the three examples as explained above.

Example 1



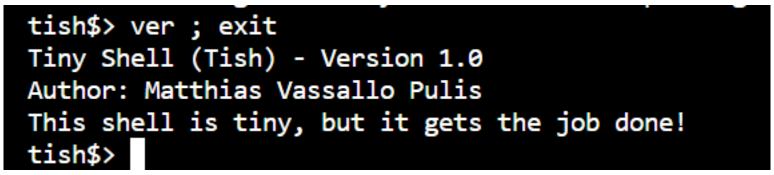
```
Current working directory: /home/matthiasvp/assignment
tish$> cwd ; cd cps1012
Current working directory: /home/matthiasvp/assignment
tish$> cwd
Current working directory: /home/matthiasvp/assignment
tish$>
```

Example 2



```
tish$> cd cps1012 ; cwd
tish$> cwd
Current working directory: /home/matthiasvp/assignment/cps1012
tish$>
```

Example 3

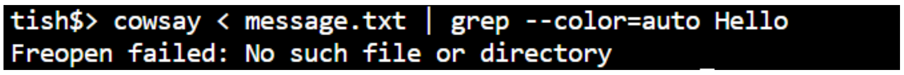


```
tish$> ver ; exit
Tiny Shell (Tish) - Version 1.0
Author: Matthias Vassallo Pulis
This shell is tiny, but it gets the job done!
tish$>
```

3.2. Use of the pipe "|" symbol

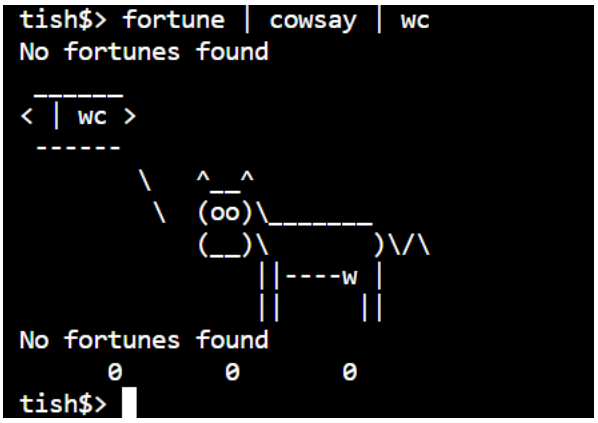
Another bug that resulted in similar issues to the last one was the pipe "|" symbol. Although it was explained in the previous section that it works, it was also mentioned that there were cases where the pipe symbol resulted in some errors, some of which are more serious than errors.

In this example, the user typed in the command **cowsay < message.txt | grep –color=auto Hello** in the shell program. Its function is to return the contents of message.txt without printing a cow. However, since this operation made use of input redirection, an "Freopen failed" error got displayed, which resulted in the program being terminated. Thus, this could also be classified as a bug relating to input redirection. Below is a screenshot that demonstrates this example.



```
tish$> cowsay < message.txt | grep --color=auto Hello
Freopen failed: No such file or directory
```

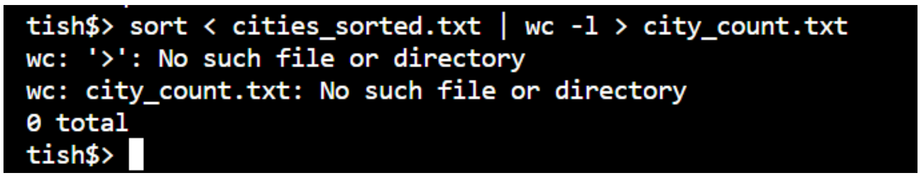
In the next example, the command **fortune | cowsay | wc** was entered in the shell program. It is supposed to return the number of lines, words and bytes for the whole operation. However, every separate operation executed on its own, with the **wc** operation returning three zeros. This of course was not expected at all. This could not be resolved in time. Below is a screenshot demonstrating the above example.



```
tish$> fortune | cowsay | wc
No fortunes found
 ______
< | wc >
 ------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
No fortunes found
      0       0       0
tish$>
```

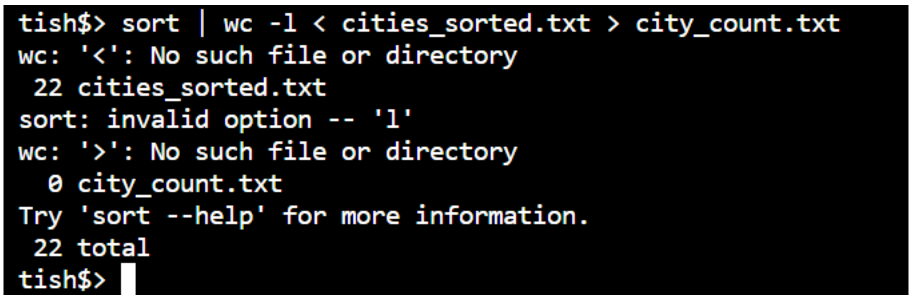
In the previous section, it was mentioned that there other commands involving the **sort** operation that resulted in errors. These two tested commands displayed below are very similar to the last successful **sort** test (page 11). Both were supposed to count the number of lines in cities_sorted.txt and store the result in city_count.txt, which was very similar to that successful test. However, these resulted in errors which were not serious enough to the point were the program was terminated. However, both commands did not execute like they were supposed to. Since the errors are related to the use of input/output redirection, these could also be classified as bugs relating to input and output redirection. Below are screenshots of the two commands demonstrating the above that resulted in errors.

Command 1



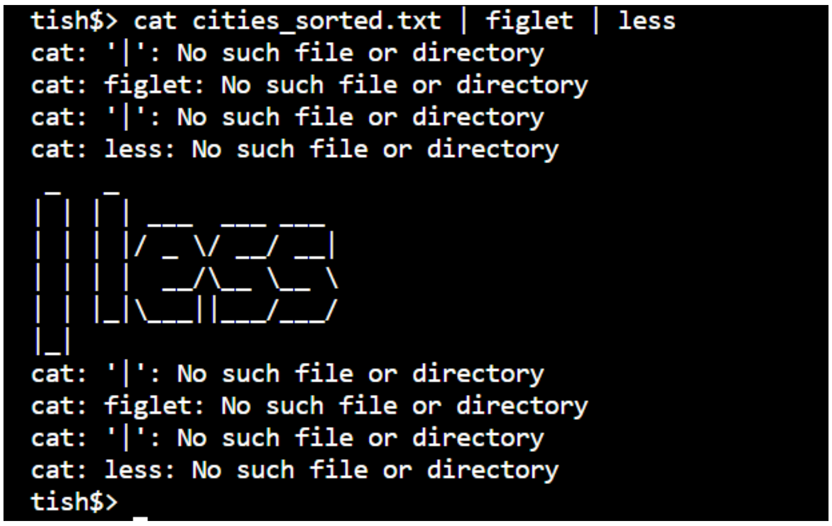
```
tish$> sort < cities_sorted.txt | wc -l > city_count.txt
wc: '>': No such file or directory
wc: city_count.txt: No such file or directory
0 total
tish$>
```

Command 2



```
tish$> sort | wc -l < cities_sorted.txt > city_count.txt
wc: '<': No such file or directory
 22 cities_sorted.txt
sort: invalid option -- 'l'
wc: '>': No such file or directory
  0 city_count.txt
Try 'sort --help' for more information.
 22 total
tish$>
```

In this following example: the command **cat cities_sorted.txt | figlet | less** was entered in the shell program. Its objective is to draw the contents of cities_sorted.txt using the **figlet** package which was installed when setting up the cps1012 directory. However, the “| less” part of the command was drawn in **figlet** instead of accessing the cities_sorted.txt. Several errors also got displayed in the shell due to the use of the “|” symbol. This is an error which could not be resolved on time. The screenshot below demonstrates the above example.

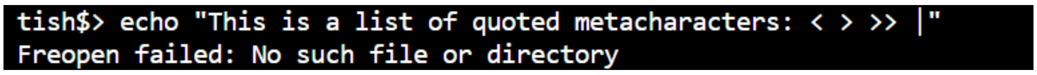


```
tish$> cat cities_sorted.txt | figlet | less
cat: '|': No such file or directory
cat: figlet: No such file or directory
cat: '|': No such file or directory
cat: less: No such file or directory
 _   _
| | | | ___  ___ ___
| | | |/ _ \/ __/ __|
| | | |  __/\__ \__ \
| | |_|\___||___/___/
|_|
cat: '|': No such file or directory
cat: figlet: No such file or directory
cat: '|': No such file or directory
cat: less: No such file or directory
tish$> _
```

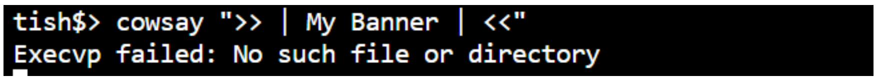
3.3. Advanced Scanning Bugs

This sub-section highlights the various bugs relating to Advanced Scanning, which was the functionality implemented in Task 4.

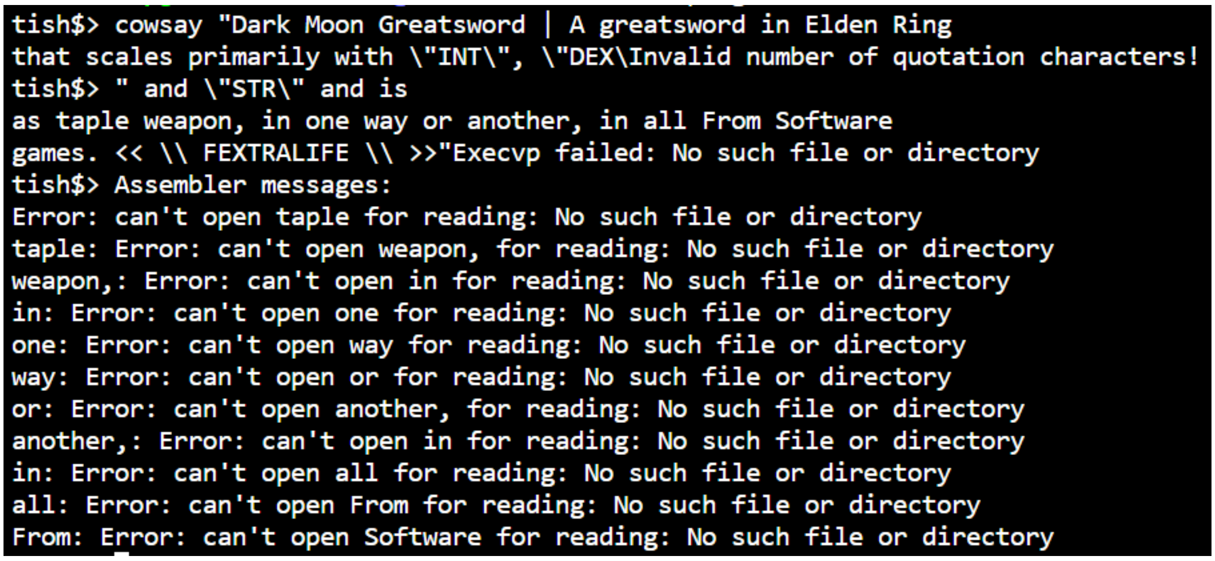
In this example, the commands **echo "This is a list of quoted metacharacters : < > >> |"** and **echo "Expanded list: \" \\ < > >> | ;"** were entered in the shell program. While the latter operation worked as expected, which was explained in the previous section (page 12), the first operation resulted in an unexpected error, relating to input/output redirection. This resulted in the program being terminated. This unfortunately could not be solved on time. The following screenshot shows the result of the first **echo** operation.


```
tish$> echo "This is a list of quoted metacharacters: < > >> |"
Freopen failed: No such file or directory
```

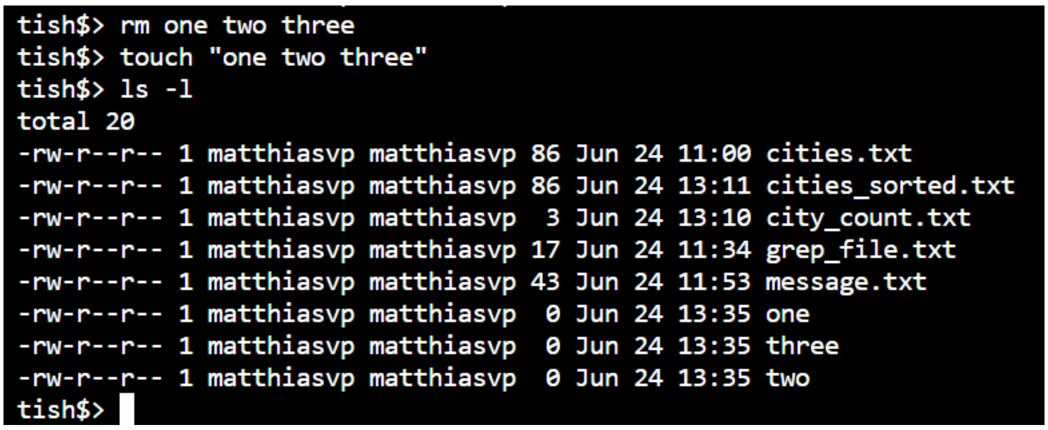
In the next example, the command **cowsay “>> | My Banner | <<”** was entered. The goal of this operation is to print the phrase in between the double quotation characters and print a cow. However, this resulted in an error, again due to the use of input/output redirection. This was serious enough to the point were the program ended up freezing until it was terminated by the user. The following screenshot demonstrates this example:


```
tish$> cowsay ">> | My Banner | <<"
Execvp failed: No such file or directory
```

In the following example, the command **cowsay "Dark Moon Greatsword | A greatsword in Elden Ring that scales primarily with \"INT\", \"DEX\" and \"STR\" and is as taple weapon, in one way or another, in all From Software games. << \\ FEXTRALIFE \\ >>"** was entered. Just like the previous two examples, this resulted in errors. However, instead of a single error, a bunch of errors were displayed, mainly due to the fact that the phrase is a very long one. The causes for these errors are the same as previously. Fortunately, the program did not terminate or freeze. The screenshot below highlights the above example.


```
tish$> cowsay "Dark Moon Greatsword | A greatsword in Elden Ring
that scales primarily with \"INT\", \"DEX\Invalid number of quotation characters!
tish$> " and \"STR\" and is
as taple weapon, in one way or another, in all From Software
games. << \\ FEXTRALIFE \\ >>"Execvp failed: No such file or directory
tish$> Assembler messages:
Error: can't open taple for reading: No such file or directory
taple: Error: can't open weapon, for reading: No such file or directory
weapon,: Error: can't open in for reading: No such file or directory
in: Error: can't open one for reading: No such file or directory
one: Error: can't open way for reading: No such file or directory
way: Error: can't open or for reading: No such file or directory
or: Error: can't open another, for reading: No such file or directory
another,: Error: can't open in for reading: No such file or directory
in: Error: can't open all for reading: No such file or directory
all: Error: can't open From for reading: No such file or directory
From: Error: can't open Software for reading: No such file or directory
```

In the previous section, it was mentioned that there was a certain bug with a **touch** operation, for example with this command: **touch "one two three"**, which performed the same operation as the same command without the double quotation characters (page 12). As mentioned previously, this is supposed to create a single file called "one two three". However, the shell program created three files, with the same names as above, instead of just one. The below screenshot demonstrates this. The three separate files were removed using the **rm** command before attempting this to avoid any possibility of not having new files created.



```
tish$> rm one two three
tish$> touch "one two three"
tish$> ls -l
total 20
-rw-r--r-- 1 matthiasvp matthiasvp 86 Jun 24 11:00 cities.txt
-rw-r--r-- 1 matthiasvp matthiasvp 86 Jun 24 13:11 cities_sorted.txt
-rw-r--r-- 1 matthiasvp matthiasvp  3 Jun 24 13:10 city_count.txt
-rw-r--r-- 1 matthiasvp matthiasvp 17 Jun 24 11:34 grep_file.txt
-rw-r--r-- 1 matthiasvp matthiasvp 43 Jun 24 11:53 message.txt
-rw-r--r-- 1 matthiasvp matthiasvp  0 Jun 24 13:35 one
-rw-r--r-- 1 matthiasvp matthiasvp  0 Jun 24 13:35 three
-rw-r--r-- 1 matthiasvp matthiasvp  0 Jun 24 13:35 two
tish$>
```