University Degree in Computer Science and Engineering
(2019-2020)

Bachelor Thesis

"ESHELL"

Author: Mr. Álvaro González de la Vega

Director: Mr. Manuel A. Lea Pereira

Colmenarejo, September 2020

ABSTRACT

The purpose of this project is to create a UNIX educational shell. The use of the command line shell is very useful to do basic operations on the system. It is the element that connects the user with the kernel.

Some could argue that the man page helps you with that, but even I, who have studied software engineering, find it difficult to understand sometimes.

Another objective I want to achieve in this project is to have a command suggestion feature. The shell is very sensitive to typing errors, so I realized there should be a suggestion that tells you *maybe you wanted to say this,* so you don't need to worry trying to figure the error.

Basically, this project should help basic users to have a more positive reaction towards the command line. This could be beneficial for society in the long term because people could learn basic UNIX through this shell.

Key words: Shell, educational, UNIX

DEDICATORIA

Me gustaría dedicar este proyecto a mi familia, en especial a mis padres. Ellos con su amor y apoyo han hecho de mí el hombre que soy ahora. Su actitud trabajadora ha sido una influencia positiva en mi vida y son un modelo a seguir.

A mi novia, Sofía, gracias por estar siempre a mi lado. Tu apoyo incondicional ha sido muy importante para mí. Te quiero

Finalmente, a la Universidad Carlos III. Gracias por estos 4 años en los que he crecido tanto como persona como en conocimiento.

INDEX

FIGURE INDEX

FIGURE INDEX ON THE APPENDIX

TABLE INDEX

TABLE INDEX ON THE APPENDIX

1. INTRODUCTION

In this chapter I will comment about what was the motivation behind this project, its goals and a brief explanation on how the document is structured.

1.1 Motivation behind the project

This project is based on a project I took in on my second year of studies. The task consisted in creating a mini shell. Basically, we needed to execute the commands and fork them into pipelines.

The idea of this project appeared after I noticed how many people do not know what a command line tool is and its functionalities. It is an immensely powerful service that can manage almost everything (permissions, where to store files, removing files …) inside of a computer. The more people know how to use that tool the more the society will benefit from it, as more and more users will know how to manage their device. After some thought, one of the basic problems that this project will try to solve seems to have been found: since the shell is exceedingly difficult to use for the basic user as it is very technical, the idea of a guide used as a tutorial seemed interesting. Most of the shells have their own guide but it lacks examples. The solution proposed is to create a new guide with less theorical content and a more practical approach. This guide will consist only of the 50 most useful commands [1].

As the main objective is to make the use of the shell easier, another functionality has been added to create suggestions after a typing error by the user. This is something that none of the major shells have and it is a huge upgrade to the current situation. By doing this, the users may find the shell more approachable.

1.2 Project’s goals

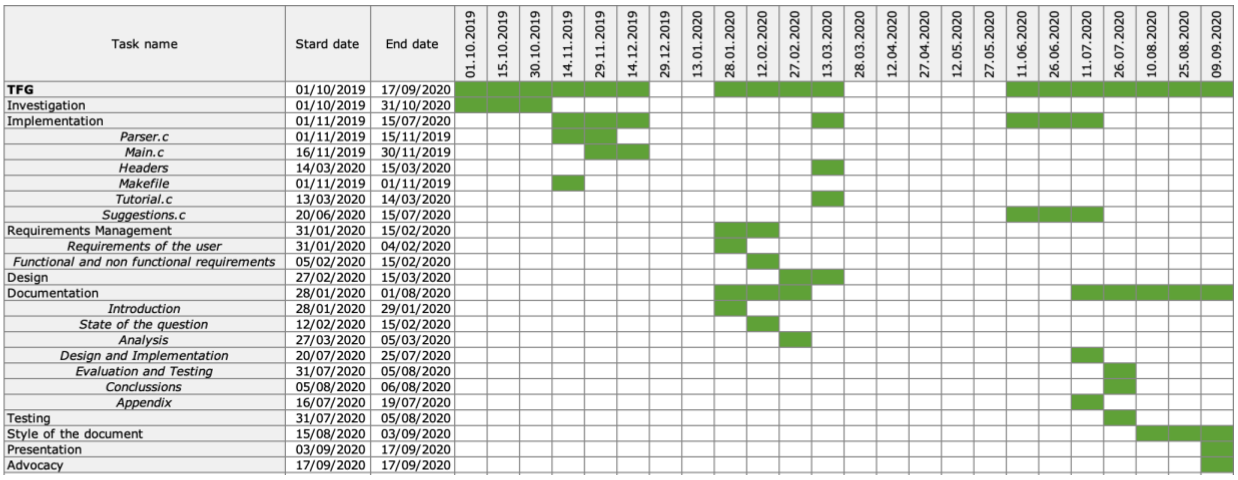
As explained, the main goal is to have increase use of shells. To do so, two things must happen: it should be more accessible to learn, and it should be easy to use. The following objectives will accomplish that:

- Parsing the input by the user to form a command.
- Forking and piping commands.
- Create a guide with practical examples of use.
- Build a command suggestion feature.

1.3 Planning

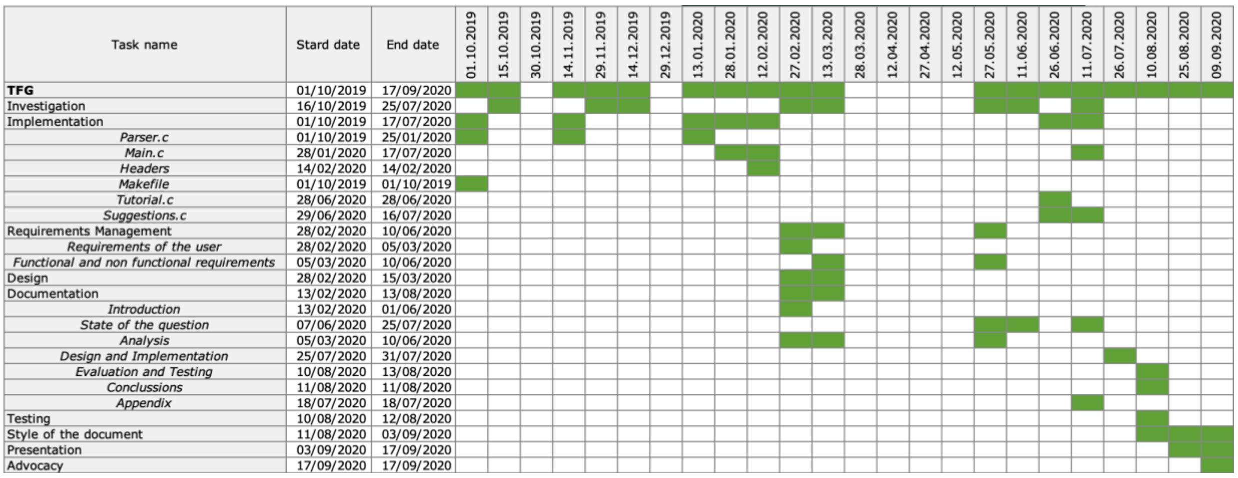
In this section the organization of tasks will be presented, as well as the duration of each. Finally we will present the total of days spent on the project.

First of all we are going to look at the initial schedule of the project and we are going to compare with the actual schedule. To do so, we are going to use Gantt diagrams.



Task name	Stard date	End date	01.10.2019	15.10.2019	30.10.2019	14.11.2019	29.11.2019	14.12.2019	29.12.2019	13.01.2020	28.01.2020	12.02.2020	27.02.2020	13.03.2020	28.03.2020	12.04.2020	27.04.2020	12.05.2020	27.05.2020	11.06.2020	26.06.2020	11.07.2020	26.07.2020	10.08.2020	25.08.2020	09.09.2020
TFG	01/10/2019	17/09/2020	🟩	🟩	🟩	🟩	🟩	🟩			🟩	🟩	🟩	🟩						🟩	🟩	🟩	🟩	🟩	🟩	🟩
Investigation	01/10/2019	31/10/2020	🟩	🟩	🟩																					
Implementation	01/11/2019	15/07/2020				🟩	🟩	🟩						🟩						🟩	🟩	🟩				
Parser.c	01/11/2019	15/11/2019				🟩	🟩																			
Main.c	16/11/2019	30/11/2019					🟩	🟩																		
Headers	14/03/2020	15/03/2020												🟩												
Makefile	01/11/2019	01/11/2019				🟩																				
Tutorial.c	13/03/2020	14/03/2020												🟩												
Suggestions.c	20/06/2020	15/07/2020																		🟩	🟩	🟩				
Requirements Management	31/01/2020	15/02/2020									🟩	🟩														
Requirements of the user	31/01/2020	04/02/2020									🟩															
Functional and non functional requirements	05/02/2020	15/02/2020										🟩														
Design	27/02/2020	15/03/2020											🟩	🟩												
Documentation	28/01/2020	01/08/2020									🟩	🟩	🟩									🟩	🟩	🟩	🟩	🟩
Introduction	28/01/2020	29/01/2020									🟩															
State of the question	12/02/2020	15/02/2020										🟩														
Analysis	27/03/2020	05/03/2020											🟩													
Design and Implementation	20/07/2020	25/07/2020																				🟩				
Evaluation and Testing	31/07/2020	05/08/2020																					🟩			
Conclussions	05/08/2020	06/08/2020																					🟩			
Appendix	16/07/2020	19/07/2020																				🟩				
Testing	31/07/2020	05/08/2020																					🟩			
Style of the document	15/08/2020	03/09/2020																						🟩	🟩	🟩
Presentation	03/09/2020	17/09/2020																								🟩
Advocacy	17/09/2020	17/09/2020																								🟩

Fig. 1.1 Initial Gantt diagram



Task name	Stard date	End date	01.10.2019	15.10.2019	30.10.2019	14.11.2019	29.11.2019	14.12.2019	29.12.2019	13.01.2020	28.01.2020	12.02.2020	27.02.2020	13.03.2020	28.03.2020	12.04.2020	27.04.2020	12.05.2020	27.05.2020	11.06.2020	26.06.2020	11.07.2020	26.07.2020	10.08.2020	25.08.2020	09.09.2020
TFG	01/10/2019	17/09/2020	🟩	🟩		🟩	🟩	🟩		🟩	🟩	🟩	🟩	🟩					🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
Investigation	16/10/2019	25/07/2020		🟩			🟩	🟩					🟩	🟩					🟩	🟩		🟩				
Implementation	01/10/2019	17/07/2020	🟩			🟩				🟩	🟩	🟩									🟩	🟩				
Parser.c	01/10/2019	25/01/2020	🟩			🟩				🟩																
Main.c	28/01/2020	17/07/2020									🟩	🟩										🟩				
Headers	14/02/2020	14/02/2020										🟩														
Makefile	01/10/2019	01/10/2019	🟩																							
Tutorial.c	28/06/2020	28/06/2020																			🟩					
Suggestions.c	29/06/2020	16/07/2020																			🟩	🟩				
Requirements Management	28/02/2020	10/06/2020											🟩	🟩					🟩							
Requirements of the user	28/02/2020	05/03/2020											🟩													
Functional and non functional requirements	05/03/2020	10/06/2020												🟩					🟩							
Design	28/02/2020	15/03/2020											🟩	🟩												
Documentation	13/02/2020	13/08/2020											🟩	🟩												
Introduction	13/02/2020	01/06/2020											🟩													
State of the question	07/06/2020	25/07/2020																	🟩	🟩		🟩				
Analysis	05/03/2020	10/06/2020											🟩	🟩					🟩							
Design and Implementation	25/07/2020	31/07/2020																					🟩			
Evaluation and Testing	10/08/2020	13/08/2020																						🟩		
Conclussions	11/08/2020	11/08/2020																						🟩		
Appendix	18/07/2020	18/07/2020																				🟩				
Testing	10/08/2020	12/08/2020																						🟩		
Style of the document	11/08/2020	03/09/2020																						🟩	🟩	🟩
Presentation	03/09/2020	17/09/2020																							🟩	🟩
Advocacy	17/09/2020	17/09/2020																								🟩

Fig. 1.2 Final Gantt diagram

The slots are periods of 15 days. There is a clear change in the outcome of the initial planning. To have a better understanding of these graphics we are going to take all the organization into tables were we will add another variable called real days, as there were slots were the work wasn't continuos (there were days were there was not any work).

As the presentation has not been done at the delivery of the document it is assumed that the initial schedule will be completed in that point.

TABLE 1.1 INITIAL PLANNING OF THE PROJECT

Task	Stard date	End date	Total days	Real days
Investigation	01/10/2019	31/10/2019	31	15
Parser.c	01/11/2019	15/11/2019	14	7
Main.c	16/11/2019	30/11/2019	14	10
Headers	14/03/2020	14/03/2020	1	0,5
Makefile	01/11/2019	01/11/2019	1	0,5
Tutorial.c	13/03/2020	14/03/2020	2	2
Suggestions.c	20/06/2020	15/07/2020	25	12
Requirements of the user	31/01/2020	04/02/2020	5	2
Functional and non functional requirements	05/02/2020	15/02/2020	10	4
Design	27/02/2020	15/03/2020	17	7
Introduction	28/01/2020	29/01/2020	2	1
State of the question	12/02/2020	15/02/2020	4	2
Analysis	27/02/2020	05/03/2020	8	4
Design and implementation	20/07/2020	25/07/2020	6	3
Evaluation and testing	31/07/2020	05/08/2020	6	2
Conclussions	05/08/2020	06/08/2020	2	1
Appendix	16/07/2020	19/07/2020	4	1
Testing	31/07/2020	05/08/2020	6	1
Style of the document	15/08/2020	03/09/2020	19	5
Presentation	03/09/2020	17/09/2020	14	3
Advocacy	17/09/2020	17/09/2020	1	1
Total of the project	01/10/2019	17/09/2020	**192**	**84**

TABLE 1.2 FINAL PLANNING OF THE PROJECT

Task	Stard date	End date	Total days	Real days
Investigation	16/10/2019	25/07/2019	45	15
Parser.c	01/10/2019	25/01/2020	42	15
Main.c	28/01/2020	17/07/2020	36	10
Headers	14/02/2020	14/02/2020	1	0,5
Makefile	01/10/2019	01/10/2019	1	0,5
Tutorial.c	28/06/2020	28/06/2020	1	1
Suggestions.c	29/06/2020	16/07/2020	18	3
Requirements of the user	28/02/2020	05/03/2020	7	2
Functional and non functional requirements	05/03/2020	10/06/2020	29	5
Design	28/02/2020	15/03/2020	17	7
Introduction	13/02/2020	01/06/2020	19	3
State of the question	07/06/2020	25/07/2020	15	10
Analysis	05/03/2020	10/06/2020	10	8
Design and implementation	25/07/2020	31/07/2020	6	4
Evaluation and testing	10/08/2020	13/08/2020	4	2
Conclussions	11/08/2020	11/08/2020	1	1
Appendix	18/07/2020	18/07/2020	1	1
Testing	10/08/2020	12/08/2020	3	2
Style of the document	11/08/2020	03/09/2020	23	2
Presentation	03/09/2020	17/09/2020	14	3
Advocacy	17/09/2020	17/09/2020	1	1
Total of the project	01/10/2019	17/09/2020	**294**	**96**

We can extract some conclussions from here. We have 102 more days than planned. That can be explained with the modifications suggested by the client, as it required to take more time than the planned.

The 12 extra days in the real days can be explained with the modifications and the COVID-19 health crisis, as it made more difficult to communicate with the client.

1.4 Budget

The budget will take into consideration the following aspects: material resources, indirect cost (transportation, electricity ..) and cost of staff.

1) Material Resources

Here we will include the software and hardware resources utilized in this project To calculate the amortization of my laptop, I am going to choose an useful life of 5 years [27]. Therefore there is a 20% of depreciation every year. As this project is going to expand over one year the 20% of the cost is going to be added in the material expenses.

TABLE 1.3 MATERIAL RESOURCES EXPENSES

CONCEPT	COST
MacBook Pro 2019	$1200€ \times 0{,}2 = 240€$
Office 365	$\frac{4{,}99€}{month} \times 12\,months = 59{,}88€$
Taxes (21%)	62,97€
TOTAL	362,85€

2) Indirect Cost

Here we will have the cost of transportation (20€/month on public transportation) and the price of Internet and Electricity at home. As I was using my room for the project, we could add the price of rent and I will say it will be 300€/month. We also need to add other expenses as the gas.

TABLE 1.4 INDIRECT COST EXPENSES

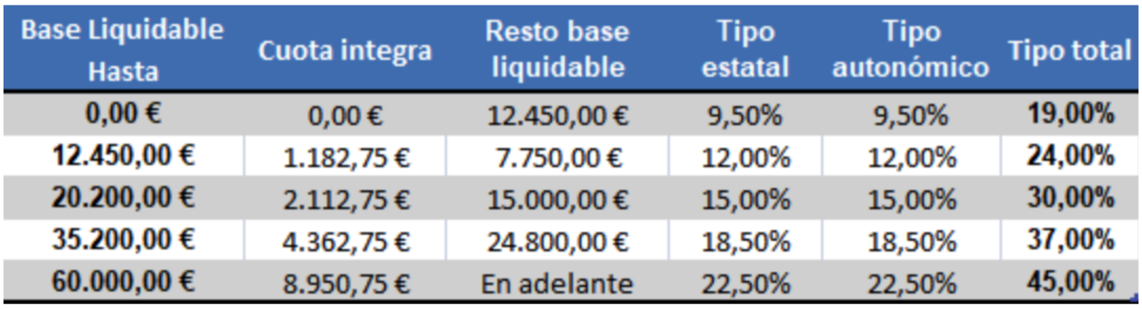
CONCEPT	COST
Travelling	$\frac{20€}{month} \times 12\,months = 240€$
Electricity	65€
Internet	$\frac{60€}{month} \times 12\,months = 720€$
Rent	$\frac{300€}{month} \times 12\,months = 3600€$
Gas	$\frac{5€}{month} \times 12\,months = 60€$
Taxes (21%)	983,85€
TOTAL	5668,85€

3) Cost of Staff

Here we will record the price of the project’s production. To do so we have taken data from LinkedIn Salary [2] to determine the salary for a: project manager and software engineer. Both roles played by me.According to the planning we are going to have 96 days of total work. There is going to be the assumption that every day of work, the amount spent on the project was of 4 hours. There will be also meetings with the client at least one every 7 days. This meetings will carry on at around 1 hour each. The price per hour is determined by this equation: $Anual\ salary \div (52\frac{weeks}{year} \times 40\frac{hours}{week})$ The average annual salary for a project manager in Madrid is 45000€ and for a software engineer 27600€. The price per hour following the previous equation is as follows:

- Project Manager: 21,63€/hour
- Software Engineer: 13,27€/hour

That is the salary, but we also need to sum up the social security and the IRPF. To calculate both we are going to need the annual salary. The percentage of retention of IRPF is calculated following the next table:



Base Liquidable Hasta	Cuota integra	Resto base liquidable	Tipo estatal	Tipo autonómico	Tipo total
0,00 €	0,00 €	12.450,00 €	9,50%	9,50%	**19,00%**
12.450,00 €	1.182,75 €	7.750,00 €	12,00%	12,00%	**24,00%**
20.200,00 €	2.112,75 €	15.000,00 €	15,00%	15,00%	**30,00%**
35.200,00 €	4.362,75 €	24.800,00 €	18,50%	18,50%	**37,00%**
60.000,00 €	8.950,75 €	En adelante	22,50%	22,50%	**45,00%**

Fig. 1.3 IRPF retention percentage [28]

According to the table we need to extract a 30% of the software engineer salary and a 37% of the project manager.

The fee of social security is going to be of 23,6% to cover normal contingencies. The extra hours will not be permited.

TABLE 1.5 STAFF EXPENSES

CONCEPT	COST
Software Engineer	$\frac{13{,}27€}{hour} \times \left(96\ days\ \times\ 4\frac{hours}{day}\right) = 5095{,}68€$
Project Manager	$\frac{21{,}63€}{hour} \times \left(96\ days\ \times\ \frac{1\ meeting}{7\ days} \times\ 1\frac{hour}{meeting}\right) = 296{,}64€$
IRPF	$0{,}3\ \times\ 5095{,}68 + 0{,}37\ \times\ 296{,}64 = 1638{,}46€$
Social security	$0{,}236\ \times\ (5095{,}68 + 296{,}64) = 1272{,}59€$
TOTAL	8301,37€

If we sum up the three factors we will obtain the total cost of the project. This adds up to 14333,07€

The budget of the whole project amounts to: **FOURTEEN THOUSAND THREE HUNDRED THIRTY-THREE EUROS AND SEVEN CENTS**

1.5 Structure of the document

This document is structured as shown in the index. The first chapter being this one, with a small description of the project and the estimated schedule and cost of the project.

In the state of question, there will be a research of already existing shells, suggestion features and guides of how to use commands as well as an explanation of the social and economic benefits from this project, including a regulatory environment review.

In the analysis the user needs are going to be stablished, and from that, the requirements will be created.

In the design and implementation, we will talk about how we are going to structure the shell how it should work and an explanation about how we implemented the shell will be given.

In evaluation and testing a battery of tests will be created and the results will be noted. The battery of tests will be based on the requirements.

Finally, there will be a discussion about the project and how it could be improved on in the future.

The document will end with the bibliography used and an appendix where the user manual for the application will be placed.

2. STATE OF THE QUESTION

2.1 State of art

First, we are going explain what a shell is. The shell is basically the user's interface to the Operating System. This tool takes commands from the user and passes them to the OS to perform. There is graphical user interfaces (GUI) and the command line interface (CLI). The latter is the more common one when we talk about shells. We interact with the shell with a terminal.

The first ever shell was created in 1971 by Ken Thompson [16]. This shell was not able to do to do scripts but introduced the concept of command interpreter for future variations [3]. This shell communicates with the kernel of UNIX-Systems.

The scripts are the standard to run commands. We are going to study the actual standard: POSIX

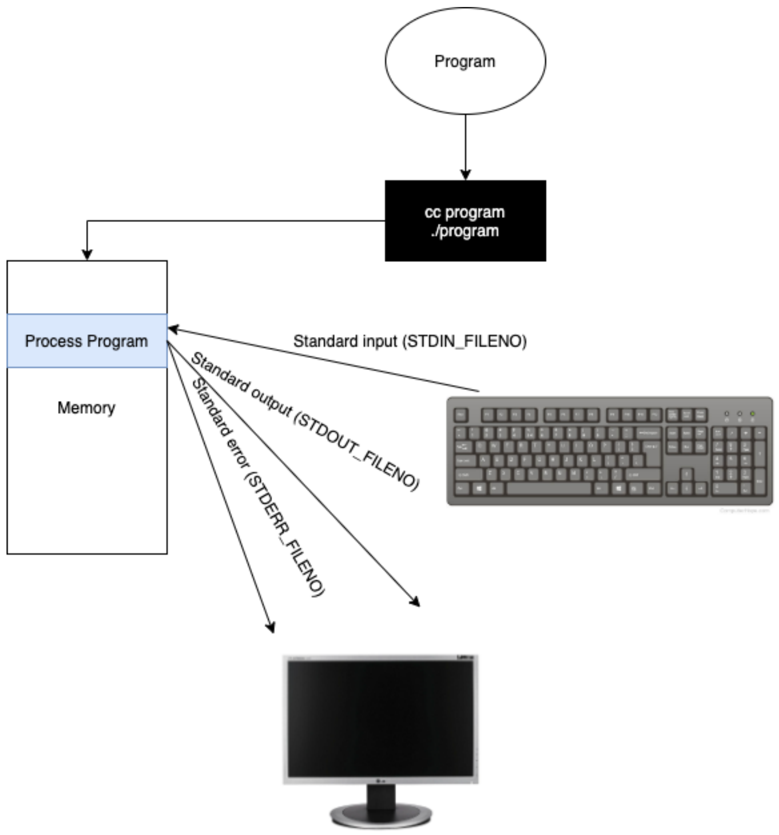
2.1.1 POSIX Standard [11]

Even though the majority of shells run in UNIX environments, there were many UNIX versions, therefore there was the need for a standardization of the scripts. POSIX was the answer to that. POSIX was published in 1988 and has a set of rules on how to do diverse actions, such as: creation of processes, signal handling, pipelines, redirection of files …

The basic language used in the scripts can be explained.

- To create a process, you need to type the following: ***cc [program]***
- To execute a process, you need to type the following: ***./[process]***
- To have the standard output of the process placed into a file, you need to type the following: ***[command] > file***
- To have the standard error of the process placed into a file, you need to type the following: ***[command] >& file***
- To have a process to read from a file, you need to type the following: ***[command] < file***
- To have a pipeline (the output of the first process is the input for the second) you need to type the following: ***[command1] | [command2]***
- To have a process running in the background, you need to type the following: ***[command] &***

A process is a program executing in the memory. When we create a process, we assign them a piece of memory. Running a process means that the instructions of the program are being executed. The standards for input, output and error are stablished, but can be changed. In UNIX, the input is the keyboard and the output and error are shown in the console.



Program
cc program
./program
Process Program
Memory
Standard input (STDIN_FILENO)
Standard output (STDOUT_FILENO)
Standard error (STDERR_FILENO)

Fig. 2.1 Explanation of the basic POSIX commands

We can divide the actual shells into two major groups: Bourne Shell and C-Shell.

2.1.2 Bourne Shell

The Bourne Shell was created by Stephen Bourne in 1978 [16], this is a major upgrade to the Thompson Shell, because it allows scripting. This shell was also the first to offer redirections, pipelines and the possibility to execute synchronous and asynchronous commands [3]. The problem with this version was that it was not user-friendly.

Other versions that were released based on the Bourne Shell are the following:

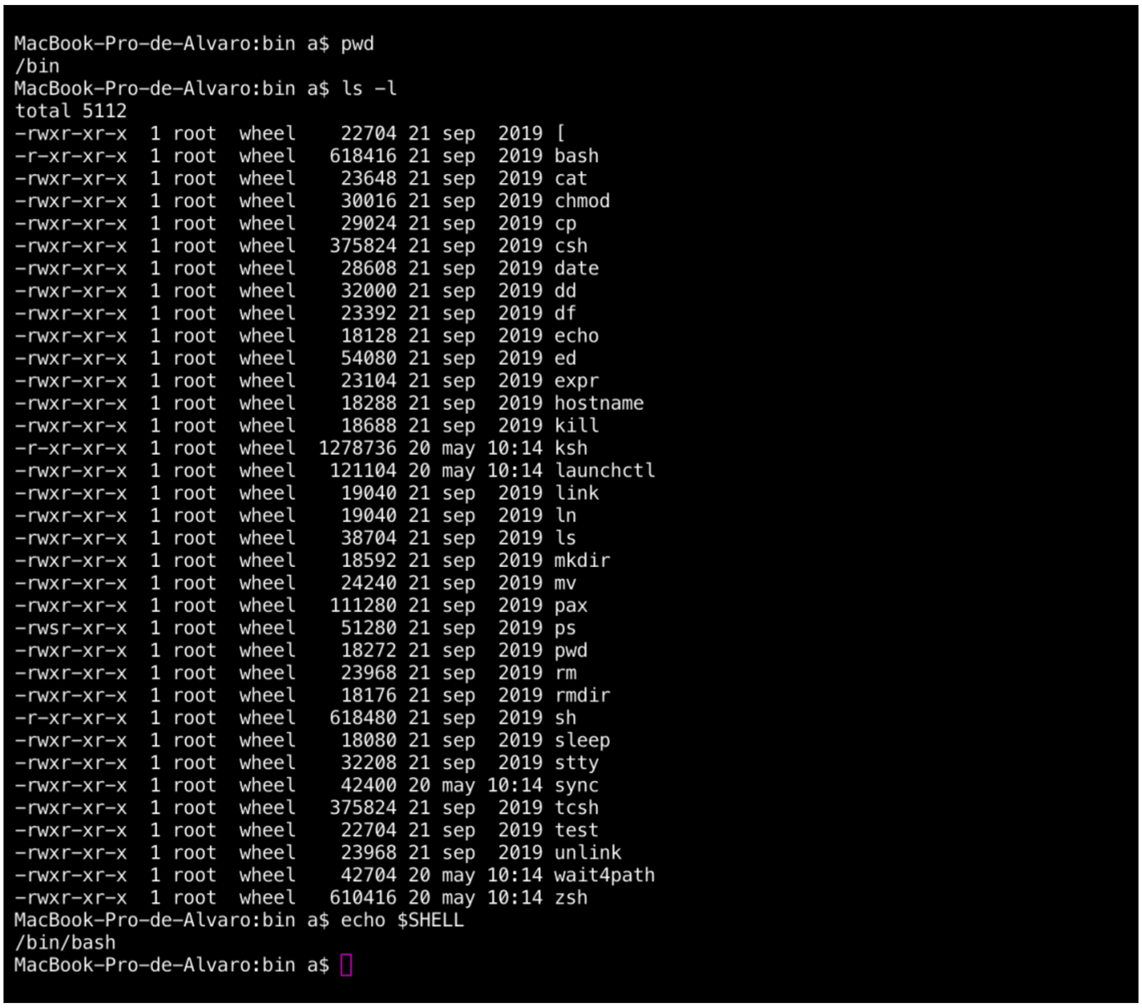
- **Korn Shell (ksh)**
- **Z Shell (zsh)**
- **Born Again Shell (bash)**

The Korn Shell was released in 1983 by David Korn [16] and even though it is basically the same shell as the sh, some extra functionalities were added. Those were command history and job control. The job control allows the shell to suspend, prioritize and start jobs (processes) [18].

The Z Shell was released in 1990 by Paul Falstad [16] and had the possibility of completion of commands by clicking on the tab key. Another functionality was the spelling error correction [19]. This last one is something similar to what we want to do in this project.

In order to work, the spelling correction has various algorithms, but there is one basic thing all have in common. We need to have a dictionary or list with the correct words. If we are talking about spelling errors, we should have to look at each word and see if there is a word that matches. If not, you should use an algorithm to produce a suggestion [21].

The third known shell is the one that is actually in most UNIX systems: The Born Again Shell commonly known as bash. This shell was released in 1989 by Brian Fox [16] and includes basic debugging and signal handling [20].



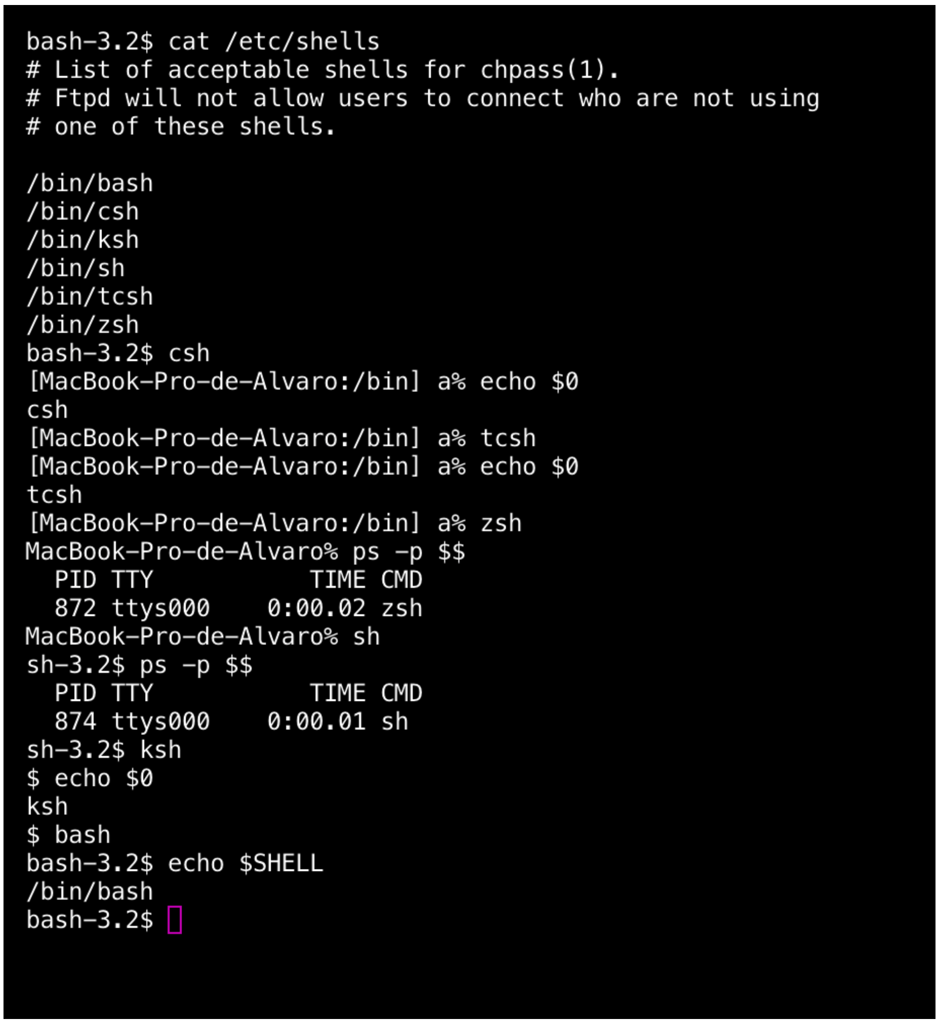
```
MacBook-Pro-de-Alvaro:bin a$ pwd
/bin
MacBook-Pro-de-Alvaro:bin a$ ls -l
total 5112
-rwxr-xr-x  1 root  wheel    22704 21 sep  2019 [
-r-xr-xr-x  1 root  wheel   618416 21 sep  2019 bash
-rwxr-xr-x  1 root  wheel    23648 21 sep  2019 cat
-rwxr-xr-x  1 root  wheel    30016 21 sep  2019 chmod
-rwxr-xr-x  1 root  wheel    29024 21 sep  2019 cp
-rwxr-xr-x  1 root  wheel   375824 21 sep  2019 csh
-rwxr-xr-x  1 root  wheel    28608 21 sep  2019 date
-rwxr-xr-x  1 root  wheel    32000 21 sep  2019 dd
-rwxr-xr-x  1 root  wheel    23392 21 sep  2019 df
-rwxr-xr-x  1 root  wheel    18128 21 sep  2019 echo
-rwxr-xr-x  1 root  wheel    54080 21 sep  2019 ed
-rwxr-xr-x  1 root  wheel    23104 21 sep  2019 expr
-rwxr-xr-x  1 root  wheel    18288 21 sep  2019 hostname
-rwxr-xr-x  1 root  wheel    18688 21 sep  2019 kill
-r-xr-xr-x  1 root  wheel  1278736 20 may 10:14 ksh
-rwxr-xr-x  1 root  wheel   121104 20 may 10:14 launchctl
-rwxr-xr-x  1 root  wheel    19040 21 sep  2019 link
-rwxr-xr-x  1 root  wheel    19040 21 sep  2019 ln
-rwxr-xr-x  1 root  wheel    38704 21 sep  2019 ls
-rwxr-xr-x  1 root  wheel    18592 21 sep  2019 mkdir
-rwxr-xr-x  1 root  wheel    24240 21 sep  2019 mv
-rwxr-xr-x  1 root  wheel   111280 21 sep  2019 pax
-rwsr-xr-x  1 root  wheel    51280 21 sep  2019 ps
-rwxr-xr-x  1 root  wheel    18272 21 sep  2019 pwd
-rwxr-xr-x  1 root  wheel    23968 21 sep  2019 rm
-rwxr-xr-x  1 root  wheel    18176 21 sep  2019 rmdir
-r-xr-xr-x  1 root  wheel   618480 21 sep  2019 sh
-rwxr-xr-x  1 root  wheel    18080 21 sep  2019 sleep
-rwxr-xr-x  1 root  wheel    32208 21 sep  2019 stty
-rwxr-xr-x  1 root  wheel    42400 20 may 10:14 sync
-rwxr-xr-x  1 root  wheel   375824 21 sep  2019 tcsh
-rwxr-xr-x  1 root  wheel    22704 21 sep  2019 test
-rwxr-xr-x  1 root  wheel    23968 21 sep  2019 unlink
-rwxr-xr-x  1 root  wheel    42704 20 may 10:14 wait4path
-rwxr-xr-x  1 root  wheel   610416 20 may 10:14 zsh
MacBook-Pro-de-Alvaro:bin a$ echo $SHELL
/bin/bash
MacBook-Pro-de-Alvaro:bin a$ 
```

Fig. 2.2 bash on a MacOS

2.1.3 C Shell

The C Shell was created by Bill Joy in 1978 [16]. The major difference between this and the Bourne Shells is the language. The C Shells try to look like C programming language. This shells also are more interactive with the user. This was the first ever shell to have command history, job control, completion of filenames (not commands) and wildcarding of names. The biggest flaw in this shell was that it could not perform scripts for various UNIX-Systems [17]. This occurred because the implementation was very chaotic, and the syntax didn't comply with the POSIX Standard.

The next generation of the C Shell was the TENEX C Shell (tcsh), released in 1983 by Ken Greer [29]. It has all the C-Shell features plus command line completion and command line editing both obtained from the TENEX source code [4].



```
bash-3.2$ cat /etc/shells
# List of acceptable shells for chpass(1).
# Ftpd will not allow users to connect who are not using
# one of these shells.

/bin/bash
/bin/csh
/bin/ksh
/bin/sh
/bin/tcsh
/bin/zsh
bash-3.2$ csh
[MacBook-Pro-de-Alvaro:/bin] a% echo $0
csh
[MacBook-Pro-de-Alvaro:/bin] a% tcsh
[MacBook-Pro-de-Alvaro:/bin] a% echo $0
tcsh
[MacBook-Pro-de-Alvaro:/bin] a% zsh
MacBook-Pro-de-Alvaro% ps -p $$
  PID TTY           TIME CMD
  872 ttys000    0:00.02 zsh
MacBook-Pro-de-Alvaro% sh
sh-3.2$ ps -p $$
  PID TTY           TIME CMD
  874 ttys000    0:00.01 sh
sh-3.2$ ksh
$ echo $0
ksh
$ bash
bash-3.2$ echo $SHELL
/bin/bash
bash-3.2$
```

Fig. 2.3 How to look what shells you have on your computer and which one is currently running

2.1.4 MAN manual

The MAN manual is the official documentation for UNIX. It was started in 1971 and had different sections to explain the functioning of UNIX-Systems. The 6 original sections were [5]:

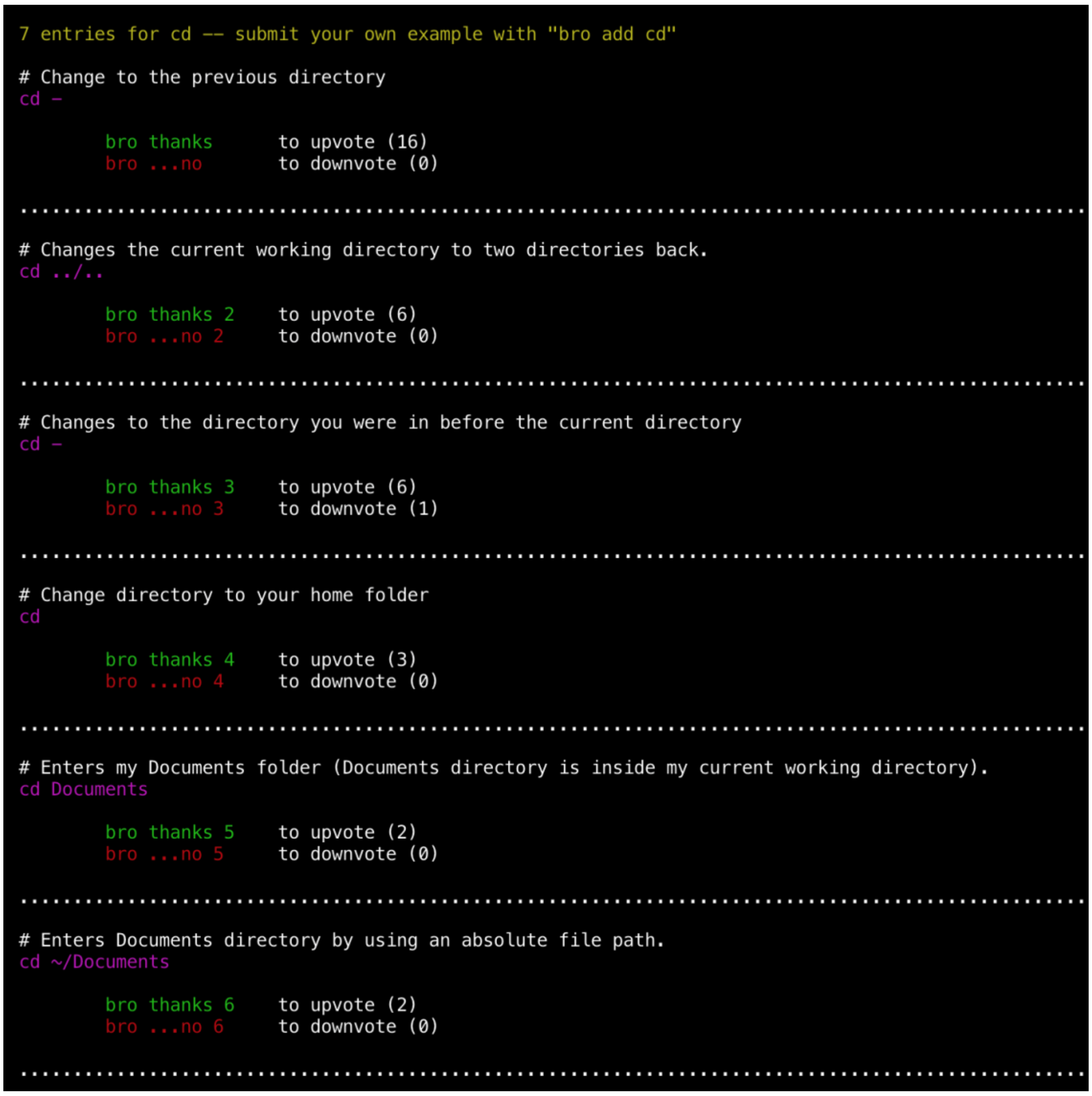
1. Commands
2. Syscalls
3. Library Functions
4. File Formats
5. Games and misc.
6. Booting and logging process

The important one for us is the first: Commands. The documentation is very extensive showing you the syntax of the command and a large explanation. One major problem is that is quite difficult to navigate through the manual [6]. Another problem is whereas we have a large amount of explanation, we do not have actual examples to understand the command. The manual itself has the following layout:

- NAME: The name of the command
- SYNOPSIS: With an explanation of how to write the command and what options are available
- DESCRIPTION: Textual overview of the command
- SEE ALSO: List of similar commands

There are alternatives to the man pages [7]:

- **Bropages:** It is a manual that includes only examples with very brief explanations. The examples are created by the users. The community can upvote or downvote the better examples.
- **Cheat:** Basically like Bropages, only this time the community doesn't participate. We have a spread sheet with examples of commands
- **TLDR:** Another manual that consists only of examples. The community can add examples.
- **Manly:** This manual explains a command fully typed. This means you need to write the full command (with flags, redirections …). It will explain every part of the command for you to understand the order you have just sent.



```
7 entries for cd -- submit your own example with "bro add cd"

# Change to the previous directory
cd -

        bro thanks      to upvote (16)
        bro ...no       to downvote (0)

....................................................................................................

# Changes the current working directory to two directories back.
cd ../..

        bro thanks 2    to upvote (6)
        bro ...no 2     to downvote (0)

....................................................................................................

# Changes to the directory you were in before the current directory
cd -

        bro thanks 3    to upvote (6)
        bro ...no 3     to downvote (1)

....................................................................................................

# Change directory to your home folder
cd

        bro thanks 4    to upvote (3)
        bro ...no 4     to downvote (0)

....................................................................................................

# Enters my Documents folder (Documents directory is inside my current working directory).
cd Documents

        bro thanks 5    to upvote (2)
        bro ...no 5     to downvote (0)

....................................................................................................

# Enters Documents directory by using an absolute file path.
cd ~/Documents

        bro thanks 6    to upvote (2)
        bro ...no 6     to downvote (0)

....................................................................................................
```

Fig. 2.4 Layout of Bropages

Ultimately, there is no manual that explains like MAN but has examples like the alternatives.

2.1.5 Spelling suggestions

Going back to what was explained previously, there has been a lot of algorithms to correct the spelling suggestions. The most common way to approach the problem of misspelling is with an edit distance.

What is an edit distance? An edit distance is the number of edit operations you need to perform in a string to match it with another [8]. This is also commonly known as the Levenshtein distance.

The formula for the Levenshtein distance is as follows:

$$lev_{a,b}(i,j) = \begin{cases} \max(i,j) & if \min(i,j) = 0 \\ min \begin{cases} lev_{a,b}(i-1,j)+1 \\ lev_{a,b}(i,j-1)+1 \\ lev_{a,b}(i-1,j-1)+1(a_i \neq b_j) \end{cases} & else \end{cases}$$

Let us explain a little bit. A and B are two strings, they can have the same length or not. i and j are the two terminal characters for the strings. The strings are therefore, 1-indexed. The last conditional $(a_i \neq b_j)$ refers that we will only add one if the string a, position i is different than the string b, position j. This would be the logical thing to do, because if the characters are equal, we should not do any edits [9].

There are three possible edits when comparing two strings:

1. **Insertion:** Adding one character
2. **Deletion:** Removing one character
3. **Substitution:** Changing one character

Knowing how this formula works, we can stablish a threshold to determine the maximum number of edits to have a word considered a misspell. The threshold is generally 2 but is not a static value.

Therefore if we find a word in our dictionary that has a number of edits lower than the threshold, we can give the user a suggestion.

One problem with this algorithm is that it gives the same amount of weight to every error. If we are typing in a QWERTY keyboard is not the same to misspell “e” with “r” than with “m”. The actual distance to the correct character should be a factor too [10].

2.1.6 Damerau-Levenshtein Distance [22].

Damerau created an algorithm very similar to what Levenshtein did. He stated that there are 4 possible edits when comparing two strings:

1. **Insertion:** Adding one character
2. **Deletion:** Removing one character
3. **Substitution:** Changing one character
4. **Swap:** Swapping two characters.

This fourth edit is something Levenshtein did not take into account and represents an upgrade in accuracy which. Reaches 80% with this new algorithm. The major downgrade is that the computing costs are higher than the ones in Levenshtein.

2.1.7 Deep Spelling

Even though the most common method to approach a problem of spelling is the edit distance, there are other ways to go around it. One of the most popular alternatives is the use of deep learning.

The deep learning is part of a branch in the Artificial Intelligence called Machine Learning. The Machine Learning tries to add intelligence to machines. The idea of intelligence is the capacity of learning something through repetition. The learning can be done by multiple algorithms, but the most common is the use of neuronal networks [23].

A neuronal network uses a set of algorithms modeled by humans to recognize patterns. This network interprets an input data and labels or clusters it [24]. With that in mind, Tal Weiss created a network to recognize words and find the misspelling that achieved 95.5% of accuracy during the testing [25]. The major downgrade of the neuronal networks is that they need large inputs of data and the right number of neurons to create knowledge. They also need lots of time to create the learning.

2.2 Socio-Economic Environment

This app was not created for economic profit. On the other hand, is a pretty cheap, and with time could achieve some profit. We do not want a monetary profit and here's why: We want schools to have this app on their software to teach shell commands. If we are going to sell to schools, we should do it at an economical price (10€/year for the software license on all the equipment). If we sell this license to 500 schools a year and counting on the fact that the software should have some minimum checking, we will begin to obtain profits on the 6th year. The cost of the project is included in the first chapter of this document.

On the other hand, this app will have a lot of benefits for education. If we sell the license to 500 schools and each school has 500 students, we could have our software being taught to 250000 persons/year. That in the short term will not be that much of a difference, but in the long term will have a lot of our students knowing the basics of the command line tool. This will translate on better knowledge of the computers, and a better understanding of its functioning.
According to the StatCounterGlobal Report [14] the most used operating systems are the indicated in the figure 2.5. The Unix-like systems are marked inside a red rectangle. They represent 61.77% of the OS that are dominating today's market.

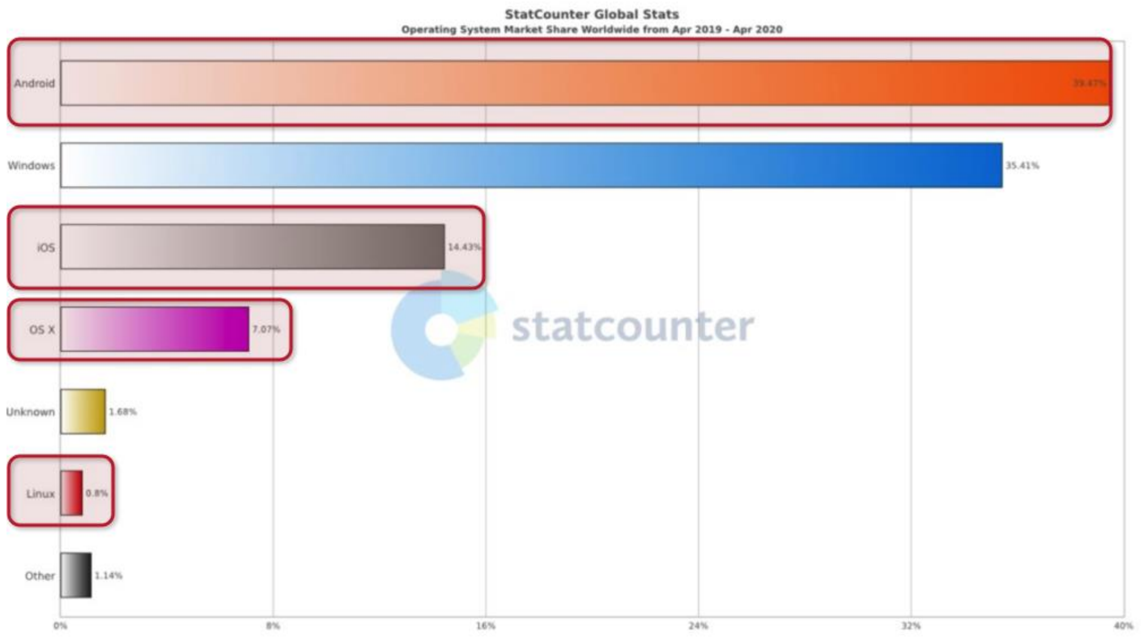


Fig. 2.5 Most used OS in 2020

This allows us to teach the most representative Operating System to the public.

As this project will be 100% digital, there is not going to be any expense that will impact on the environment. This app will be installed on personal computers and does not require any server or a database warehouse, so the impact will be zero.

This is an academic project and it will take years before it even begins to make profits but would be an upgrade on society. If basic users start to learn about operating systems with this, we will have a lot more opportunities to improve our software, as we will have more people interested in this area

2.2.1 Regulatory Environment

This project has no economic motives, so there will be an MIT license to allow users to modify and distribute the software. If this code is utilized for commercial use, a copyright notice will be needed. The source code is uploaded in a GitHub repository and will be released to the public after the advocacy of the thesis.

This shell will follow the spirit of free software that is stablished in the actual shells available to the public. These shells have a free software that is open for distribution and modification under the condition that any work created from that must have the same terms [13].

Even though this project is based on another I had at college, the source code is completely original. The libraries used for this project are part of the standard C libraries and are open source [12]

The project will not have any personal data saved, so there is no need to apply any article from either the General Data Protection Regulation or the Spanish Organic Law on Protection of Personal Data.

3. ANALYSIS

We need to stablish the requirements. The role of the user will be played by my tutor and the role of requirements engineer will be played by me. First, we are going to establish the user’s needs:

- We need to execute basic commands. How we are going to do it? Our app will be located between the user and the shell. We are going to parse the commands written by the user and execute with the command execvp().
- Have a suggestion feature when the user has a typing error for the 50 most useful commands.
- Have a tutorial for the user on the 50 most useful commands.

Why those needs? We basically have a mini-shell, that reads, parse and executes commands so there is obviously the need to run the basic commands and pipelines. Also, as this is an educational shell, we need to have a tutorial of the commands. The suggestions feature is something that is not available in every shell and it can be a differential factor as it is also the tutorial. We are going to establish the user’s requirements based on the user’s needs

3.1 User Requirements

TABLE 3.1 USER REQUIREMENTS

Name	Run a basic command
ID	UR1
Date	05/03/2020
Description	The app should execute the commands our user has written
Status	Verified

TABLE 3.2 USER REQUIREMENTS

Name	Run a command with pipelines
ID	UR2
Date	05/03/2020
Description	Our app must be able to execute a command with a pipeline. This means, that we have two commands where the output of the first is the input of the second
Status	Verified

TABLE 3.3 USER REQUIREMENTS

Name	Run a command on the background
ID	UR3
Date	04/06/2020
Description	Our app must be able to execute a command on the background. By doing this, we could be executing two processes at the same time
Status	Verified

TABLE 3.4 USER REQUIREMENTS

Name	Have a suggestion after typing error
ID	UR4
Date	05/03/2020
Description	If the user types a command with errors, the app will show a suggestion with the command correctly written. There will be only one suggestion and it would be the most significant one. This will only happen with the 50 most useful commands.
Status	Verified

TABLE 3.5 USER REQUIREMENTS

Name	Have a tutorial of the commands
ID	UR5
Date	07/03/2020
Description	The program must show users examples of the commands and explain which functionality will be filled with that command. This will only happen with the 50 most useful commands.
Status	Verified

From here I, as the requirements engineer, am going to establish the functional and non-functional requirements. The user has asked for functionalities of a basic shell and some extra functionalities like the suggestions and tutorial. As the user has specified, we should be able to execute basic commands, but this will only be through a call to the UNIX shell. So, we should be able to read and parse the commands, but our app is not going to actually execute them. The validation and verification of the requirements is specified in chapter 5 (Evaluation and testing).

3.2 Functional Requirements

TABLE 3.6 FUNCTIONAL REQUIREMENTS

Name	Read the command
ID	FR1
Date	10/03/2020
Status	Our app will have a prompt were the user will write the commands. After the user has written the command, we must read it.
Status	Verified

TABLE 3.7 FUNCTIONAL REQUIREMENTS

Name	Search for pipes
ID	FR2
Date	10/03/2020
Status	After reading the command, the app must search for the pipe symbol (“|”) and tokenize the elements (separated commands)
Status	Verified

TABLE 3.8 FUNCTIONAL REQUIREMENTS

Name	Parse the command
ID	FR3
Date	10/03/2020
Status	The app must parse every command looking for redirections (in, out and error) and the background symbol (“&”). When done, it also needs to separate every command in different words (“ls”, “-l” …)
Status	Verified

TABLE 3.9 FUNCTIONAL REQUIREMENTS

Name	Execute commands
ID	FR4
Date	10/03/2020
Status	After parsing the commands, the app will execute them with the execvp command.
Status	Verified

TABLE 3.10 FUNCTIONAL REQUIREMENTS

Name	Check error
ID	FR5
Date	10/03/2020
Status	If the command specified by the user is incorrect, the app should look to the list of the 50 most useful commands and search if it finds something similar. If not, the prompt will be popped again
Status	Verified

TABLE 3.11 FUNCTIONAL REQUIREMENTS

Name	Offer suggestion
ID	FR6
Date	04/06/2020
Status	If the app finds a command similar (within the threshold of edits) to the one the user has typed, it will show a suggestion with the new command on screen.
Status	Verified

TABLE 3.12 FUNCTIONAL REQUIREMENTS

Name	Have a tutorial for the commands
ID	FR7
Date	04/06/2020
Status	If the user wants to learn one of the commands, we will show the pertinent information, as well as some examples to practice with the user. Only for the 50 most useful commands, otherwise the app will show a message which explains that command is not listed on our guide
Status	Verified

TABLE 3.13 FUNCTIONAL REQUIREMENTS

Name	Exit the program
ID	FR8
Date	10/03/2020
Status	If the user wants to exit the program, he must do it after typing exit.
Status	Verified

3.3 Non Functional Requirements

TABLE 3.14 NON FUNCTIONAL REQUIREMENTS

Name	Execute on UNIX
ID	NFR1
Date	01/10/2019
Status	This program must be executed in a UNIX environment.
Status	Verified

TABLE 3.15 NON FUNCTIONAL REQUIREMENTS

Name	Program in C
ID	NFR2
Date	01/10/2019
Status	This program should be implemented in the C language, as it is the most interactive one with the shell.
Status	Verified

TABLE 3.16 NON FUNCTIONAL REQUIREMENTS

Name	Tutorial
ID	NFR3
Date	04/06/2020
Status	To show the tutorial of the commands the user must type: *tutorial [command]*
Status	Verified

TABLE 3.17 NON FUNCTIONAL REQUIREMENTS

TABLE XVII NON-FUNCTIONAL REQUIREMENTS	
Name	Offering suggestions
ID	NFR4
Date	04/06/2020
Description	If the user has an error executing one command and it is one of the 50 most used commands, the app will show the following on screen: *Maybe you wanted to type this: [Command corrected]*
Status	Verified

3.4 Traceability matrices

TABLE 3.18 TRACEABILITY BETWEEN THE GOALS OF THE PROJECT AND THE USER REQUIREMENTS

Requirements/Objetive	Objetive 1: Parsing the input by the user to form a command	Objetive 2: Forking and piping commands	Objetive 3: Create a guide with practical examples of use	Objetive 4: Build a command suggestion feature
UR1: The app should execute the commands our user has written	X	X		
UR2: Our app must be able to execute a command with a pipeline. This means, that we have two commands where the output of the first is the input of the second	X	X		
UR3: Our app must be able to execute a command on the background. By doing this, we could be executing two processes at the same time	X	X		
UR4: If the user types a command with errors, the app will show a suggestion with the command correctly written. There will be only one suggestion and it would be the most significant one. This will only happen with the 50 most useful commands	X			X
UR5: The program must show users examples of the commands and explain which functionality will be filled with that command. This will only happen with the 50 most useful commands	X		X	

TABLE 3.19 TRACEABILITY BETWEEN USER REQUIREMENTS AND SOFTWARE REQUIREMENTS

Software/User	UR1	UR2	UR3	UR4	UR5
FR1: Our app will have a prompt were the user will write the commands. After the user has written the command, we must read it	X	X	X	X	X
FR2: After reading the command, the app must search for the pipe symbol ("|") and tokenize the elements (separated commands)	X	X	X	X	X
FR3: The app must parse every command looking for redirections (in, out and error) and the background symbol ("&"). When done, it also needs to separate every command in different words ("ls", "-l" …)	X	X	X	X	X
FR4: After parsing the commands, the app will execute them with the execvp command.	X	X	X		
FR5: If the command specified by the user is incorrect, the app should look to the list of the 50 most useful commands and search if it finds something similar. If not, the prompt will be popped again	X	X	X	X	X
FR6: If the app finds a command similar (within the threshold of edits) to the one the user has typed, it will show a suggestion with the new command on screen				X	
FR7: If the user wants to learn one of the commands, we will show the pertinent information, as well as some examples to practice with the user. Only for the 50 most useful commands, otherwise the app will show a message which explains that command is not listed on our guide					X
FR8: If the user wants to exit the program, he must do it after typing exit.	X				
NFR1: This program must be executed in a UNIX environment.	X	X	X	X	X
NFR2: This program should be implemented in the C language, as it is the most interactive one with the shell.	X	X	X	X	X
NFR3: To show the tutorial of the commands the user must type: *tutorial [command]*					X
NFR4: If the user has an error executing one command and it is one of the 50 most used commands, the app will show the following on screen: *Maybe you wanted to type this: [Command corrected]*				X	

3.5 User cases

Now, we need to stablish the user cases, to define the design from them.

A user case is the interaction between the user and the app. When we define them, we need to stablish who is the actor (the user) and what are the pre-conditions and post-conditions

TABLE 3.20 USER CASES

Name	Execute a command (with pipelines, background …)
ID	CU1
Actor	User
Pre-conditions	1. The app is executing 2. The prompt is visible
Post-conditions	1. If the command is correctly typed, we will run it 2. If the command is not correctly typed and it is on the list, we will show a suggestion 3. If the command is incorrect and it is not part of the list, we will simply show the prompt again
Normal flow	The user will type in the command he wants and then press the Enter key

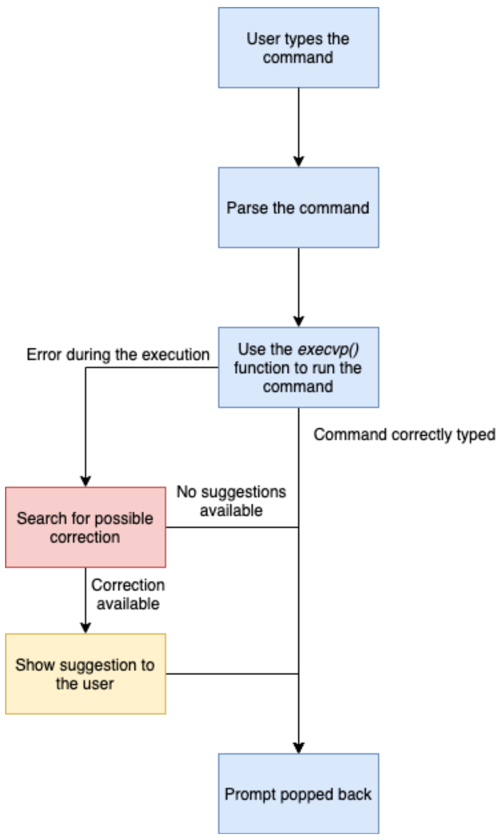

User types the command
Parse the command
Use the *execvp()* function to run the command
Error during the execution
Command correctly typed
Search for possible correction
No suggestions available
Correction available
Show suggestion to the user
Prompt popped back

Fig. 3.1 Flow-chart of CU1

TABLE 3.21 USER CASES

Name	Learn the commands
ID	CU2
Actor	User
Pre-conditions	1. The app is executing 2. The prompt is visible
Post-conditions	1. If the command the user wants to learn is on the list, we will show him an explanation as well as examples of the command 2. If it is not on the list, we will show the prompt again
Normal flow	The user will write “tutorial [command]” and then press the Enter key

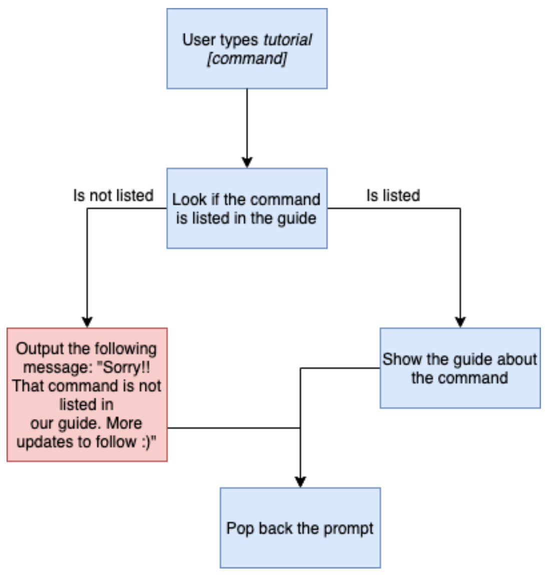

User types *tutorial [command]*
Is not listed
Look if the command is listed in the guide
Is listed
Output the following message: "Sorry!! That command is not listed in our guide. More updates to follow :)"
Show the guide about the command
Pop back the prompt

Fig. 3.2 Flow-chart of CU2

TABLE 3.22 USER CASES

Name	Exit the app
ID	CU3
Actor	User
Pre-conditions	1. The app is executing 2. The prompt is visible
Post-conditions	1. The app will finish its execution
Normal flow	The user will write “exit” and then press the Enter key

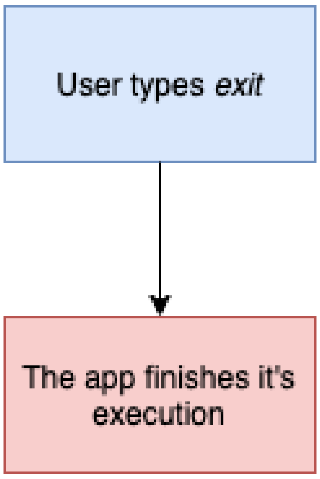


Fig. 3.3 Flow-chart of CU3

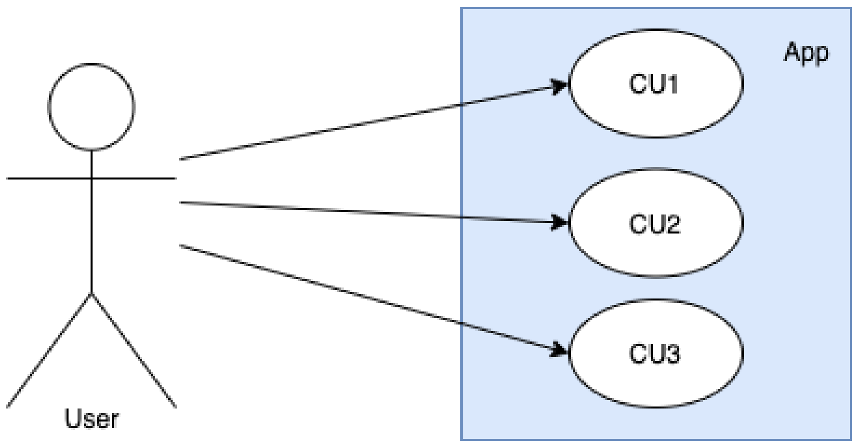


Fig. 3.4 Class diagram

4. DESIGN AND IMPLEMENTATION

After seeing the user cases we need to establish the design of our app. First, we are going to take a look at the requirements. One of the most important ones is the FR4, that determines that we are going to execute the commands with a function. That means that our app will operate above the kernel. Is possible to execute the commands with built-in functions, but it will require a huge amount of time that surpasses the allowed according to the hours of ECTS.

We can see that we want the user to type everything (commands, tutorials, exit, …). So, the first thing we have to do is have a prompt to catch the user inputs. Now we need to look at what a shell does.

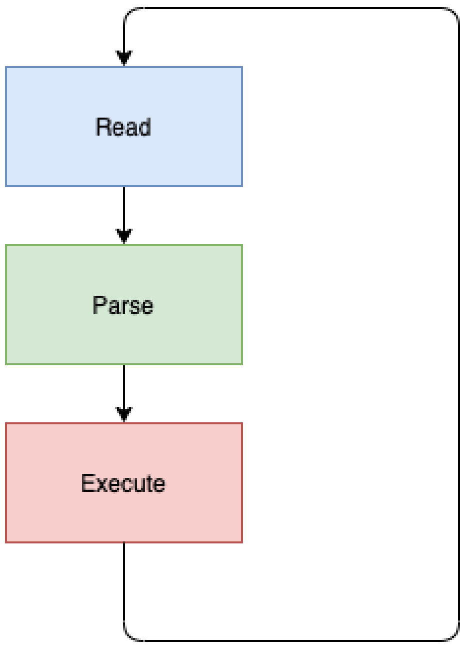


Fig. 4.1 Flow chart of a shell

Looking at the flow chart, we can clearly see that our shell needs to be running indefinitely. The best way to do that is inside an infinite loop. After we should catch the typing of the user. The prompt will handle that.

The parsing is more complicated as the POSIX standard is very specific. The different signals are as follows [11]:

- **&:** Background signal. It means that your process will be executed outside the main queue of processes. Doing this, we will be able to execute other processes while the first process is running.
- **|:** Pipeline signal. Doing this you are having two processes (one at the left of the symbol and the other at the right) being executed together. This can be explained with a diagram

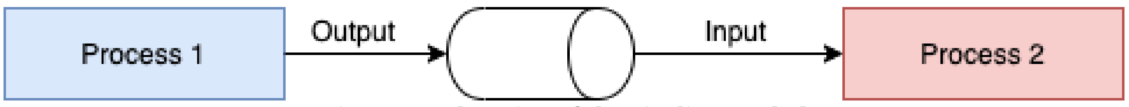


Fig. 4.2 Explanation of the pipeline symbol

Basically, you are executing one process and the result of that process is written in the pipe instead of the standard output (screen). The second process will start immediately, but instead of reading from the standard input (keyboard) it will read from the pipe.

- **>:** Redirection output signal. It changes the standard output to the one user specifies.
- **>&:** Redirection error signal. It changes the standard error output to the one the user specifies.
- **<:** Standard input signal. It changes the standard input to the one the user specifies.

Other symbols to consider are the following:

- **; and &&:** These two symbols are used to run multiple commands at once. The first one (;) executes the first command and immediately the second one, no matter what happened to the first. The second symbol only executes the second command if the first has succeeded.
- $: Parameter expansion symbol. This symbol is immediately followed by either () or {} to define some parameter or environment variable. The environment variables are values assigned inside the shell.

These two symbols have not been included in the design of the parsing. The reasons are the following:

The dollar symbol requires to have access to the environment values, which are found inside the folder **proc/PID/environ**. When we start the app, we do not know the PID of our process even though there are ways to find it.

The other one, were you can execute several commands at the same time was also discarded because the new users could benefit to the approach of one command at a time.

When we parse a command, we are going to save it inside a structure where we will have:
- Number of pipes
- Redirection files {IN, OUT, ERR}
- Background signal (yes or no)
- String of instructions to execute

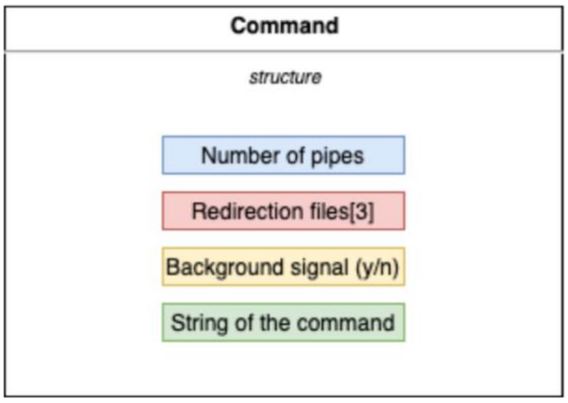
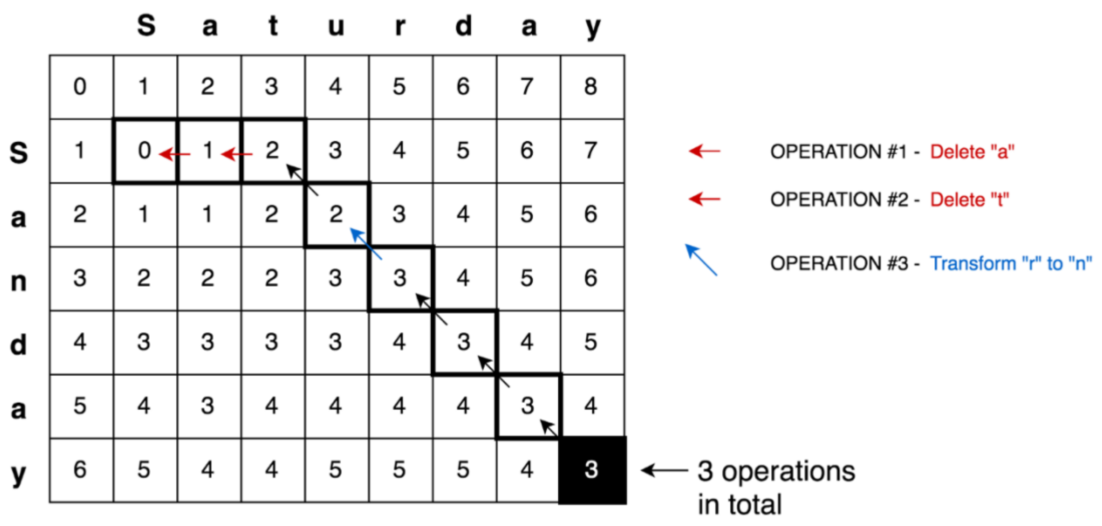


Fig. 4.3 Structure of the command after parsing

After parsing what we need to do is to execute the command. There are two built-in functions (Exit and the tutorial guide), so if we detect that the first word of the String is one of them, we will call those functions.

If not, the procedure will always be the same. Fork the process and piping it. If our PID is greater than 0 it means we are in the parent process, so we should wait to the current process to finish. If it’s 0 we will execute the command with the *execvp()* function. This is a function included in the *unistd.h* library and has a return value. If that value is different than 0 it means that there has been an error during the execution of the command, so after that we will need to look if it is a spelling error.

To compare the input by the user and the list of correct commands, we will use the Levenshtein algorithm. We are only going to do it when the difference between the length of the two strings is within 2. When the length is on the range, we will apply the Levenshtein algorithm.



		S	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
a	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

OPERATION #1 - Delete "a"
OPERATION #2 - Delete "t"
OPERATION #3 - Transform "r" to "n"
3 operations in total
Fig. 4.4 Levenshtein algorithm [15]

This algorithm works as described in the state of art. After we obtain the result (number of edits) we save that value only if it has a number of edits lower than the threshold. This threshold was stablished as 2, to stop miss-corrections. If it fits the criteria, we will save the number of edits and the string from the list as the one with less edits required. If we do find another with less edits that would be the one selected. After searching through the entire list, we print in the standard output the following message: "Maybe you wanted to type this: [COMMAND CORRECTED] [ARGUMENTS]". If there is not a String that fits the criteria on the list, the prompt will be printed.

The two built-in functions' process is very straightforward. The exit function exits the app with a value of 0 (Success). The tutorial one searches through the list the second word of the string in the command. If we find one, we will print the guide at the standard output. If not, the following message will be prompted: "Sorry!! That command is not listed in our guide. More updates to follow ☺"

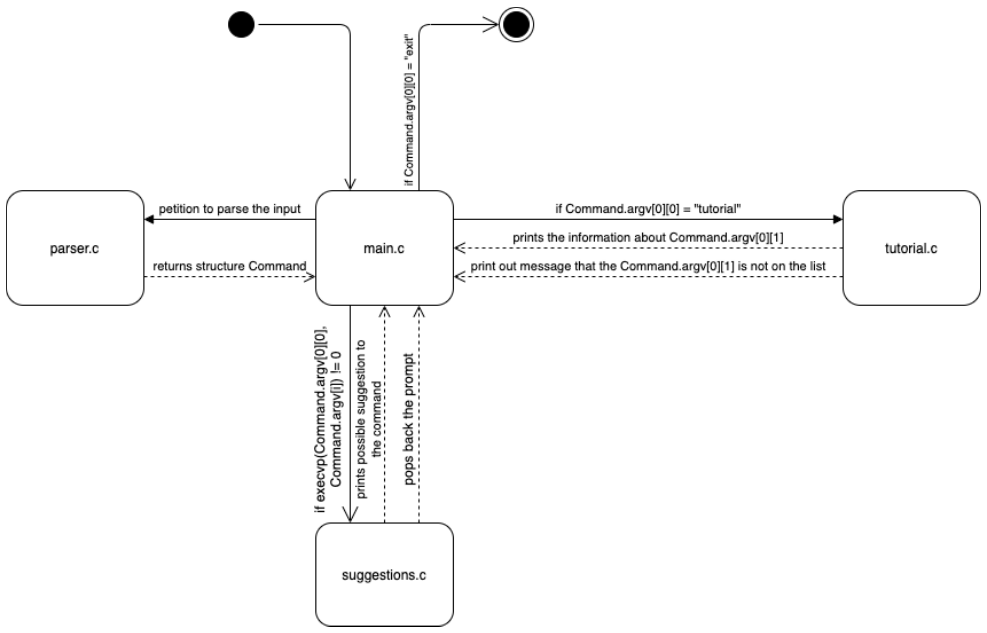

if Command.argv[0][0] = "exit"
parser.c
petition to parse the input
returns structure Command
main.c
if Command.argv[0][0] = "tutorial"
prints the information about Command.argv[0][1]
print out message that the Command.argv[0][1] is not on the list
tutorial.c
if execvp(Command.argv[0][0], Command.argv[i]) != 0
prints possible suggestion to the command
pops back the prompt
suggestions.c

Fig. 4.5 Explanation of the app

The implementation of this project includes 8 files in the source code: 4 for **development** (suggestions.c, parser.c, main.c and tutorial.c), 3 of **headers** (suggestions.h, parser.h, tutorial.h) and one **makefile** (Makefile). We are going to discuss every one of them as all of them are key for the correct functionation of the app.

4.1 Makefile

The Makefile has a script to turn the 4 development files and the 3 headers into an executable called **esh**. To create the executable, we use the standard gcc command. The flags used for the creation are the following:
-Wall: We receive every warning generated during the compilation.
-g: To generate debug information. This is very useful when the user wants to debug the program.
-o: To generate an output file. This output file will be the executable.

There is also the option to remove the executable file. It can be triggered using the command *make clean*. This removes every text file, folder and object generated, as well as the executable file.

4.2 Headers (parser.h, tutorial.h, suggestions.h)

These three files contain the functions and definitions of the development ones.

4.3 Parser.c

After catching the input by the user, we need to parse it into a command. To do it we are going to follow the sequential diagram situated below:

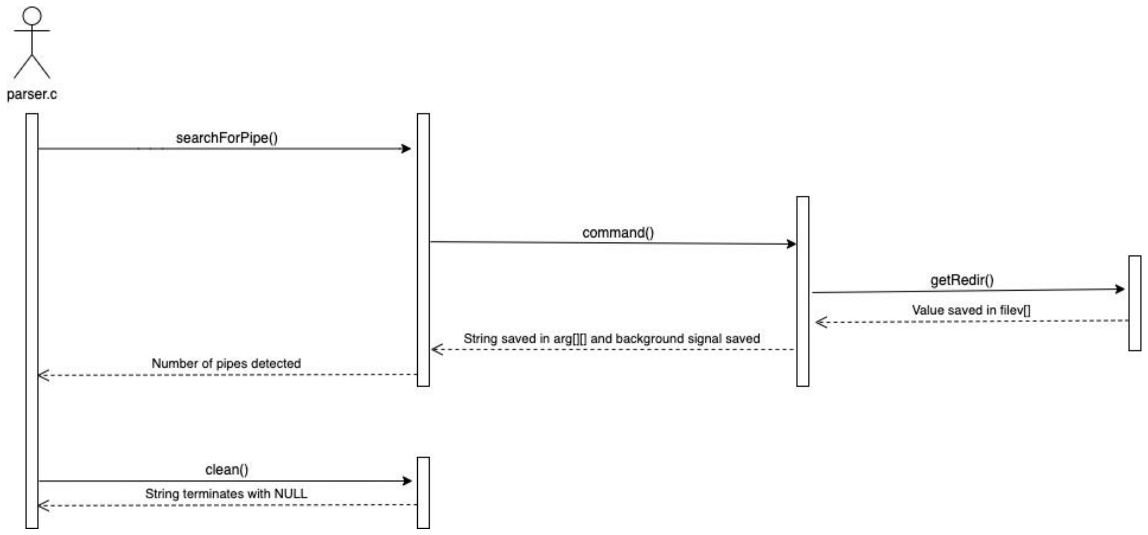


Fig. 4.6 Sequential diagram for parser.c

As we can see there are 4 main functions in this program. The first one will be *searchForPipe(),* which will tokenize the elements before and after a separator. That separator will be the pipe symbol ("|"). Immediately after, we will pass every one of the tokens to the *command()* function. This function reads the input character by character searching for other symbols (redirection and background). The input will be saved in a double char. The first part refers to the pipe, the second to the number of words.

argv[PIPES][WORDS]: If we have the following input ("ls -l -htr | grep main") the argv variable will be saved like this:

-argv[0][0] = ls
-argv[0][1] = -l
-argv[0][2] = -htr
-argv[1][0] = grep
-argv[1][1] = main

If we do find a redirection symbol, the function *getRedir()* will be called. This function basically saves the first word found after the redirection symbol in the one specified. Let us remember we have three possible redirection files (IN, OUT, ERROR), all with different symbols. If the background symbol is found, we will put one variable called **bg** to 1.

After doing that we call the *clean()* function. When the input of *command()* is saved in a double char, we assume that every time we hit a blank space we will have a new word to save. This is incorrect and we need to use the *clean()* function to put the blanks saved in arg[][]to NULL. If this is not done, the execvp function will fail.

4.4 Tutorial.c

This function searches through a list of 50 commands. If one element of the list matches the second word of the command, the guide for that command will be printed. If not, a typo message will appear.

The guide will have the following structure:

- **SYNTAXIS:** Here the app shows the how to write the command an the possible flags or aguments to add when typing it.
- **SYNOPSIS:** An explanation on what the command does
- **EXIT OF THE FUNCTION:** Shows the possible exit outcomes when calling at the command.
- **EXAMPLE FOR YOU TO TRY:** Here we will have various examples for the user to try. This examples have been selected for the user to have a better understanding of the command. There will be a brief explanation on each example.

The messages will be printed on the standard output and it is NOT possible to do redirections or piping in this particular command. There has been added hyperxtualization to highlight the different sections of the guide.

4.5 Suggestions.c

This program goes through the list to look if the user has mistyped the command. To do it, we will follow the next sequential diagram:

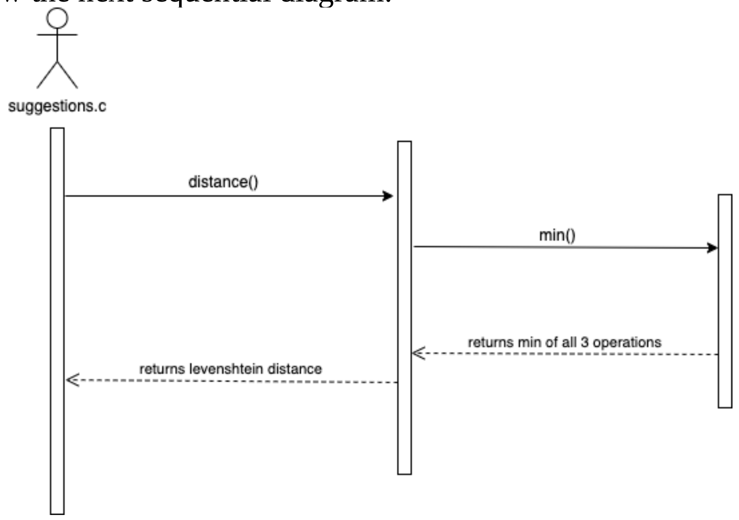


Fig. 4.7 Sequential diagram for suggestions.c

First, we will look at the length of the first word typed by the user. This word is the command. We will look at the list to see if a command matches with what the user was trying to type. As a first step, the difference in the length of the element of the list and the type of the user should not be greater than 2. By doing this we reduce the time of execution and stop miss corrections (corrections obtained by the code, but that are not a real correction). After this sort of elements, we go inside a loop where we will call the *distance()* function. Here we will execute the Levenshtein algorithm. This is a recursive function where we compare from the last elements to the first. If the last letter in both elements matches, we will re-call the function but starting at the second-to-last letter. If not, we need to obtain the result of elimination of a letter, substitution of a letter and aggregation of a letter. With these three values (obtained after calling the *distance()* function again in a recursive way) we need to choose the minimum operation. That's the job of the function *min().*

Once this is done, we obtain a number, that's the Levenshtein distance between two Strings. We stablish that the number of changes permitted to be considered a correction is 2. That is the standard for this problem. We will save the element of the list with the minimum number of changes as the correction. Then we will print the correction in the standard output. If there is no correction saved, we will exit the program.

4.6 Main.c

Now we are going to talk about the main program of the project. The first thing we do here is read the input of the user. The we will call the parser.c, which will return us the command parsed inside the structure defined in the figure 4.3. We will look inside the structure to see if the first word is "exit" or "tutorial". If it is either one of those words we will call to the built-in functions.

If not, we will go inside the function *execute(),* this function is the one that does the forking and piping. The first thing to do is to eliminate zombie processes. To do so we will do a waitpid. After that we will go inside a loop. This loop will be repeated as may times as the number of pipes detected. Inside here we will look in the read-end of the pipe (if it is the first time, we will read a value of -1) and save it as **oldfd.** Then we will re-do the pipe and fork the process. If our PID is greater than 1 it means that we are in the parent process so we will close the pipe and go back to the start of the loop. If we have a PID of 0 it means we are on the child process so we can execute the program (in the end a command is a program) without exiting the application.

The next thing to do is to read and write. If there is a read redirection file and we are on the first iteration of the loop it means that we have to read from the input file. Otherwise, we will read from the value saved at **oldfd.** After that, we need to write. If we have an output redirection file and we are on the last iteration of the loop, we will write inside the output file. Otherwise, we will write on the write-end of the pipe. The error redirection will be open all the way through as an error can be spotted at every process.

Finally, we will execute the command with the execvp() system-call function. That function has a return value, and if it is less than 0 it means that there has been an error during the execution. If that occurs, we will call to suggestions.c.

If the background symbol is off, we will wait indefinitely for the parent process to finish. Otherwise, we will not wait. When this function terminates, we go back to the beginning until the user types “exit”.

5. EVALUATION AND TESTING

In this section we are going to establish different sets of proofs to check if our app is working correctly. All the testing was done on the Carlos III Linux server, also known as **guernika.** The connection was done with the *ssh* command on my own personal laptop.

5.1 Battery of test

TABLE 5.1 VALIDATION TEST

ID	Requirements	Description	Input	Expected output
VTP-01	UR1, FR1, FR3, FR4, NFR1, NFR2	The normal use of the shell is going to be tested. As a first step, we are going to test one command	ls -l	It must show all the files and folders in our current directory
VTP-02	UR1, FR1, FR3, FR4, NFR1, NFR2	Now we are going to test the redirection.	ls -l > prueba.txt	It must save all the files and folders in our current directory inside the file “prueba.txt”. It should not show the information in the standard output
VTP-03	UR1, UR3, FR1, FR3, FR4, NFR1, NFR2	The background signal is going to be tested.	ls -l & echo hola	It must show all the files and folders of our current directory and it should not show the prompt again until the execution of the second command (echo hola). After the typing of the second command, it will show hola on the standard output and pop the prompt back.
VTP-04	UR1, UR2, FR1, FR2, FR3, FR4, NFR1, NFR2	The pipeline is going to be tested.	ls -l | grep main	It must show the file main.c in the standard output.

VTP-05	UR5, FR1, FR7, NFR1, NFR2, NFR3	The tutorial is going to be tested. We are going to search the information of the command pwd	tutorial pwd	It must show all the information of the command pwd.
VTP-06	UR5, FR1, FR7, NFR1, NFR2, NFR3	We are going to search the information of a command not listed in our guide.	tutorial rm	It must show the following message in the standard output: *Sorry!! That command is not listed in our guide. More updates will follow* ☺
VTP-07	UR4, FR1, FR3, FR5, NFR1, NFR2, NFR4	The suggestion feature is going to be tested. We are going to type a non-existent command.	qwerty	It must pop the prompt back again, with no messages whatsoever.
VTP-08	UR1, FR1, FR8, NFR1, NFR2	The exit function is going to be tested.	exit	The following message must show on the standard output : *Goodbye!* and then finish the application.

5.2 Testing the spelling corrector

Now the testing is going to be done on the suggestion feature tool. We are going to test 2 variations of each command. These variations will have no more than 2 edits (to have an available suggestion). The total percentage will give us the accuracy of this functionality.

This test is done separately from the validation test as we are going to test the capability of our feature. The result of this will determine if the requirements are validated or not. As all of the words misspelled have a possible correction, anything under a 100% of accuracy will mean that the requirement FR6 is not validated.

The list of the commands and the expected suggestions is as follows:

TABLE 5.2 TESTING THE SUGGESTION FEATURE

Input of the user	Expected suggestion
pwf	pwd
per	pwd/tar
ñs	ls/ps
fg	cd/ls/df/cp/dd/ps/ln/mv
cf	cd/df/cp
ht	cd/ls/df/dd/cp/ps/ln/mv
mkdi	mkdir

mkire	mkdir
rmdit	rmdir
rmfi	rmdir
lfblñ	lsblk
psbl	lsblk
monte	mount
mot	mount
de	df/dd
ddf	dd/df
name	uname
imame	uname
pl	ps
pln	ln/ps
kil	kill
tilo	kill
servicio	service
ervic	service
bat	batch
ath	batch
sutdon	shutdown
shutdo	shutdown
tuh	touch
ouch	touch
ail	tail
tttail	tail
pp	cp/ps
kcep	cp
bmv	mv
cv	mv/cp/cd
com	comm
coma	comm
les	ls/less
lesion	less
gato	cat
cateo	cat
hear	head
hesf	head
lk	ln/ls
leno	ln
comp	cmp/comm
clp	cmp
dg	dd/df
dder	dd
alia	alias
aliasio	alias
call	cal
col	cal
fortuna	fortune

furtun	fortune
istori	history
histori	history
yeah	yes/head
yep	yes
baner	banner
bunneri	banner
real	rev
sev	rev
get	wget
vgeta	wget
ipteibles	iptables
ipatable	iptables
treiceroute	traceroute
trazerute	traceroute
cool	curl/cal
carl	curl
fide	find
found	find
witch	which
wich	which
lookate	locate
logeate	locate
gep	grep
grefg	grep
sez	sed
siid	sed
clear	clear
cla	clear/cal
eco	echo
eo	echo/ls/cd/df/ps/ln/dd/cp
short	sort
sote	sort
suda	sudo
zud	sudo
cmod	chmod
chod	chmod
own	chown
cown	chwon
men	man
mano	man
tarta	tar
ter	tar
whats	whatis
watiz	whatis

After doing the testing the success rate is of 100%. With this sample we can insure that the feature works correctly.

5.3 Results of testing

TABLE 5.3 RESULTS OF TESTING

ID	Requirements	Result
VTP-01	UR1, FR1, FR3, FR4, NFR1, NFR2	Ok
VTP-02	UR1, FR1, FR3, FR4, NFR1, NFR2	Ok
VTP-03	UR1, UR3, FR1, FR3, FR4, NFR1, NFR2	Ok
VTP-04	UR1, UR2, FR1, FR2, FR3, FR4, NFR1, NFR2	Ok
VTP-05	UR5, FR1, FR7, NFR1, NFR2, NFR3	Ok
VTP-06	UR5, FR1, FR7, NFR1, NFR2, NFR3	Ok
VTP-07	UR4, FR1, FR3, FR4, FR5, NFR1, NFR2, NFR4	Ok
VTP-08	UR1, FR1, FR8, NFR1, NFR2	Ok
Suggestion tool	UR5, FR1, FR3, FR4, FR5, FR6, NFR1, NFR2, NFR4	100% accuracy

6. CONCLUSSIONS AND FUTURE WORK

The problem to be solved was to make the use of the shell easier for beginners. To do so, a new shell was going to be built from scratch with some extra functionalities. One of them is to create a guide of the commands. This guide needed to be more practical and to reduce the amount of text. That task has had a positive result as the number of lines is significantly lower, also hypertextualization has been added with different colours to highlight the different sections of the tutorial. On each command, we had the description of what was going to be done with an example for the user to try. This goal has been successfully passed.

Another of the goals, was the creation of a suggestion feature tool to correct possible typing errors of the user. This task has also had a good result. The algorithm used was the Levenshtein distance, but with some changes explained in chapter 4. As described in chapter 5, we tried 2 different variations of the words to check if the algorithm gives us the corrected command. The success rate was of 100%, so we can assume with some certainty that we achieved that goal.

Referring to the basic use of the shell, the execution of commands works perfectly, as well as the redirection, piping and background signal. So the goal of forking and piping is checked as well as the parsing.

With these improvements, we can insure that the interaction with the shell will be more dynamic, therefore easier to use.

To improve the work done, we could expand the number of commands used by the Levenshtein algorithm. For this project we used 50 commands, but currently there are 281555 commands available [26], so the sample size is relatively small. New entries to the guide will be needed as well as the corrections available.

Other possible improvement could be found in the description of the commands, as there could be more examples. The use of a database to store examples and the interaction with a community could be optimal. Although it will require more expenses on hardware.

The parser could be more accurate with the UNIX standard (explained in chapter 2), this standard has more syntax related that was discarded during the design of the app. The syntax is explained in chapter 4.

Finally, the correction tool could be improved. As we commented in chapter 2, the display of the keyboard is a factor when miss-typing. Another thing to account for is the phonetics. For example, the "ph" and "f" have the same phonem **[f],** but are separated inside the QWERTY keyboard.

Some context could be added as the tool will give us the first command founded on the list that fits the criteria. This means that maybe it gives you a suggestion that is also wrong. If we type *co a.txt b.txt*, we probably want to copy the file a.txt into the file b.txt. The execution of this command will fail as the correct command is *cp*. In this case, the tool gives us a suggestion to use the command *cd*, which is completely different than the *cp* one.

Implementing this changes would take at least 2 to 3 years, as the addition of that amount of commands is huge. It would require more personnel working on the project. The estimate is at least 20 persons in charge of the addition of the commands. 2 or 3 new members of staff working on the suggestion feauture and a couple of employeers working on the rest of the software. Rounding up, at least 25 members inside the project. The office will need to be bigger, therefore more rent to pay for, the cost of materials will increment and there will be more salaries to pay. Taking all of that into account and the expectancy of another 2 or 3 years of work, the cost of the project could go up to one million euros.

BIBLIOGRAPHY

[1] UbuntuPit - The 50 Most Useful Commands To Run in the Terminal: https://www.ubuntupit.com/best-linux-commands-to-run-in-the-terminal/
[2] LinkedIn Salary: https://www.linkedin.com/salary?trk=d_flagship3_nav
[3] Ken Thompson & Dennis M. Ritchie. The UNIX-Time Sharing System https://dl.acm.org/doi/pdf/10.1145/357980.358014
[4] TCSH Manual: http://www.kitebird.com/csh-tcsh-book/tcsh.pdf
[5] Man First Edition Manual: http://man.cat-v.org/unix-1st/
[6] Charles. H Franke III & Nancy J. Wahl. Authoring a hypertext UNIX help Manual: https://dl.acm.org/doi/pdf/10.1145/259526.259561
[7] osTechNix. Good Alternatives To Man Pages Every Linux User Needs To Know https://www.ostechnix.com/3-good-alternatives-man-pages-every-linux-user-know/
[8] Christopher D. Manning, Prabhakar Raghavan & Hinrich Schütze. Introduction to Information Retrieval. https://nlp.stanford.edu/IR-book/pdf/irbookonlinereading.pdf
[9] Ethan Nam. Understanding the Levenshtein Distance Equation for Beginners. https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0
[10] Peter Norvig. How to write a Spelling Corrector.
http://norvig.com/spell-correct.html
[11] The Open Group Base Specifications Issue 7, 2018 edition
https://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html
[12] GNU Operating System. What is free Software? https://www.gnu.org/philosophy/free-sw.en.html
[13] GNU Operating System. GNU General Public License https://www.gnu.org/licenses/gpl-3.0.html
[14] StatCounter GlobalStats. Operating System Market Share Worldwide. https://gs.statcounter.com/os-market-share
[15] GitHub user **anilkumarnandamuri**. Spelling mistake in readme of levenshtein-distance. https://github.com/trekhleb/javascript-algorithms/issues/311
[16] Unix Harley. The Unix Timeline for Students. http://unix.harley.com/instructors/timeline.html
[17] William Joy. An Introduction to the C-Shell. https://docs.freebsd.org/44doc/usd/04.csh/paper.html

[18] The KornShell Command and Programming Language. http://www.kornshell.com/doc/ksh93.html
[19] Paul Falstad. The Z Shell Manual. http://zsh.sourceforge.net/Doc/zsh_a4.pdf
[20] Bash Reference Manual.
https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html
[21] Google. Search 101. https://www.youtube.com/watch?v=syKY8CrHkck#t=22m03s
[22] Gregory V. Bard. Spelling-Error Tolerant, Order-Independent Pass-Phrases via the Damerau-Levenshtein Distance String-Edit Distance Metric. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.480.2329&rep=rep1&type=pdf
[23] Dot CSV. ¿Qué es el Machine Learning?¿Y Deep Learning? Un mapa conceptual https://www.youtube.com/watch?v=KytW151dpqU
[24] A.I Wiki. A Beginner's Guide to Neural Networks and Deep Learning
https://pathmind.com/wiki/neural-network
[25] Tal Weiss. Deep Spelling https://machinelearnings.co/deep-spelling-9ffef96a24f6

[26] Quora. How many Unix commands are there? Link
[27] Best Reviews. How Long Does A Mac Last Link
[28] IRPF 2020 http://www.irpf.com.es/
[29] Ken Greer. C Shell with command and filename recognition/completion.
https://groups.google.com/g/net.sources/c/BC0V7oosT8k/m/MKNdzEG_c3AJ?pli=1

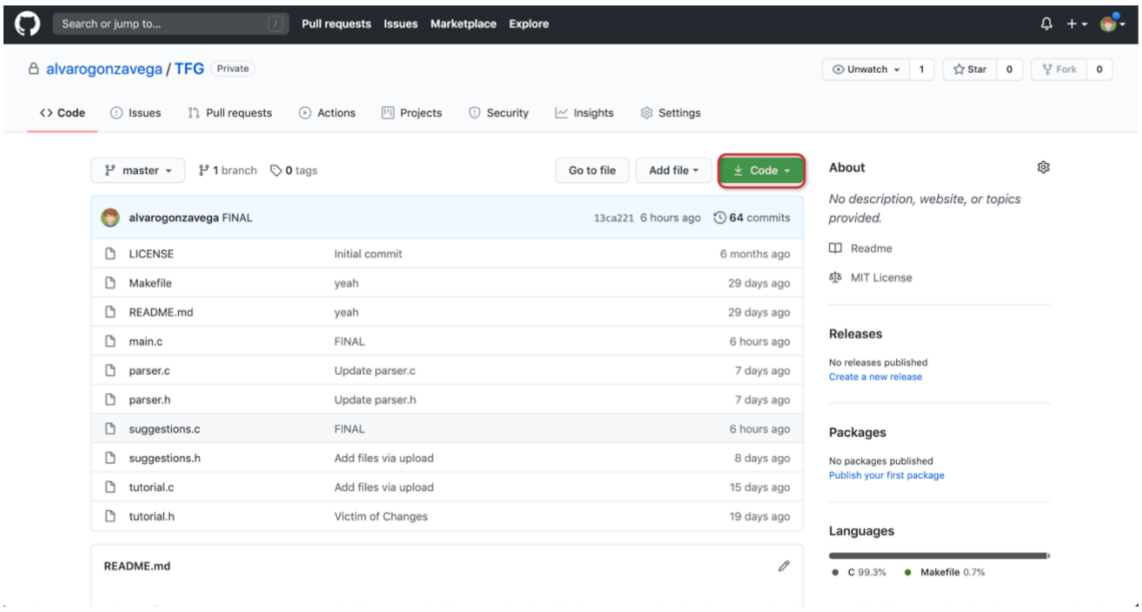
APPENDIX

In this section we are going to have a user guide on how to install and use the app.

A.1 Installation

First and foremost, you need to have a Linux OS on your device to run the app. Then you will need access to the source code (It will be open for the public after the publication of this document). The link to the code is the following:
https://github.com/alvarogonzavega/TFG

When you get there, you will need to click on the button highlighted below



alvarogonzavega FINAL 13ca221 6 hours ago 64 commits

LICENSE	Initial commit	6 months ago
Makefile	yeah	29 days ago
README.md	yeah	29 days ago
main.c	FINAL	6 hours ago
parser.c	Update parser.c	7 days ago
parser.h	Update parser.h	7 days ago
suggestions.c	FINAL	6 hours ago
suggestions.h	Add files via upload	8 days ago
tutorial.c	Add files via upload	15 days ago
tutorial.h	Victim of Changes	19 days ago

Fig. A.1 You should click on the code button

It will launch a menu where you will click the download zip button. After you unzip that file, you will access the directory and open a terminal there.

Another option is to open a terminal directly and type the following commands:

```
git clone https://github.com/alvarogonzavega/TFG
cd TFG
```

Fig. A.2 Commands to get to working directory of the project

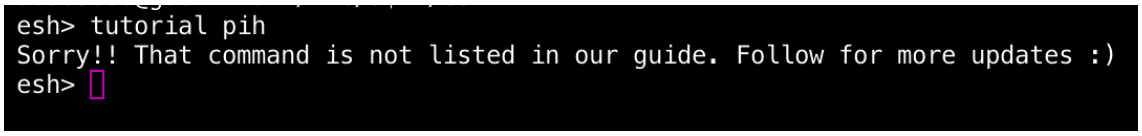
After that you will need to type these two commands:

```
make
./esh
```

Fig. A.3 Commands to launch the app

A.2 Guide of use

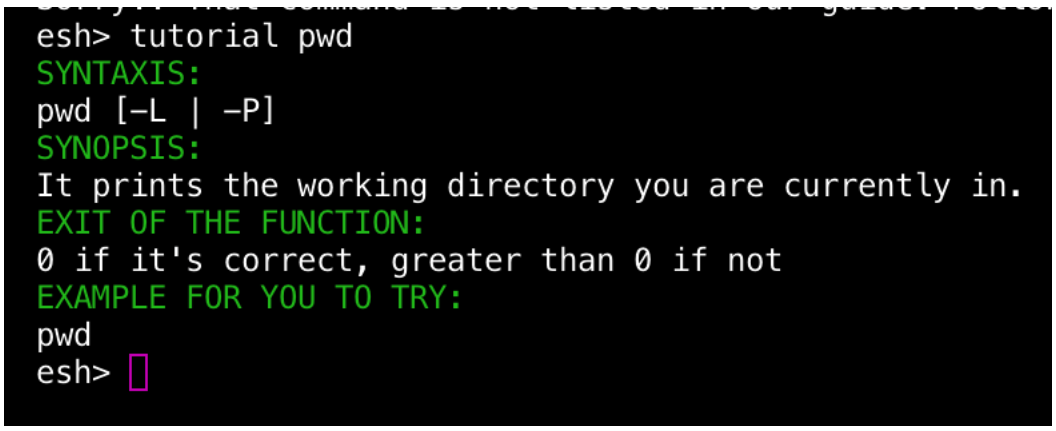
Now we have the app currently running visibly in the terminal. First, we are going to try to obtain the tutorial for a command outside the 50 most useful commands.



```
esh> tutorial pih
Sorry!! That command is not listed in our guide. Follow for more updates :)
esh> 
```

Fig. A.4 Tutorial incorrect

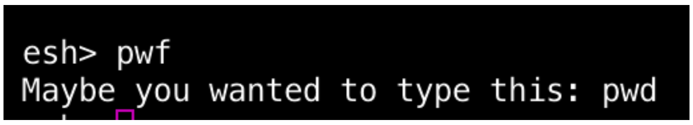
As we can see this command is not listed in the guide, so we are going to try to obtain the tutorial of a command of the pwd command.



```
esh> tutorial pwd
SYNTAXIS:
pwd [-L | -P]
SYNOPSIS:
It prints the working directory you are currently in.
EXIT OF THE FUNCTION:
0 if it's correct, greater than 0 if not
EXAMPLE FOR YOU TO TRY:
pwd
esh> 
```

Fig. A.5 Tutorial of pwd

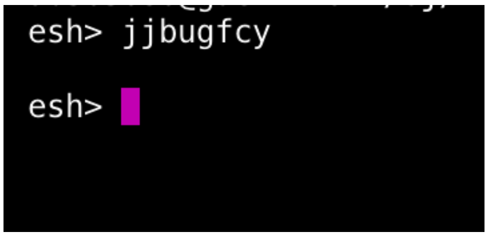
It all appears to be working so we are going to see if the suggestion function works correctly. We are going to mistype the pwd command for *pwf*



```
esh> pwf
Maybe you wanted to type this: pwd
```

Fig. A.6 Suggestion (pwf -> pwd)

Now we are going to type a non-existent command to look if our app simply pops right back the prompt to us.



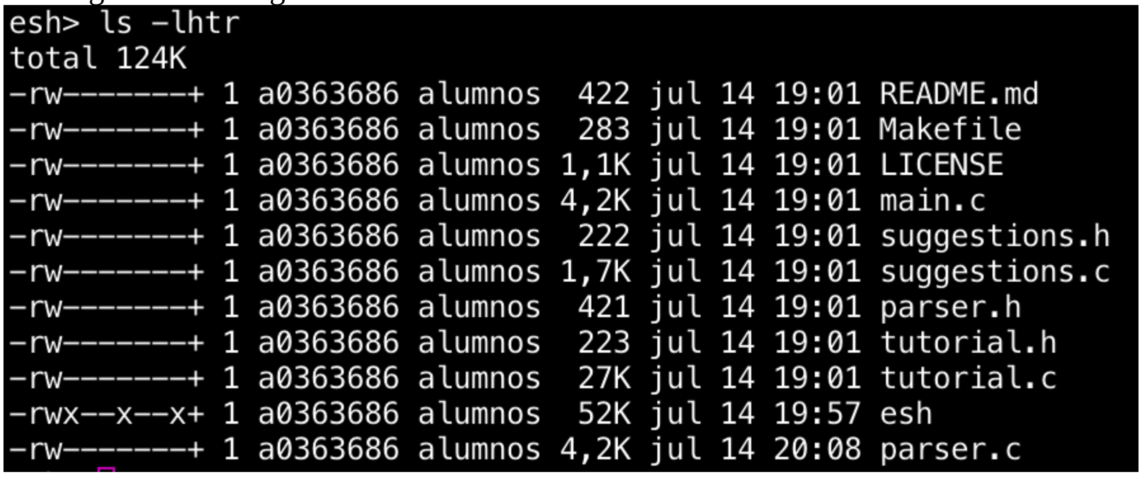
```
esh> jjbugfcy

esh> 
```

Fig. A.7 Typing non-sense on the app

The extra functionalities seem to be working so, now we are going to test the normal shell functions.

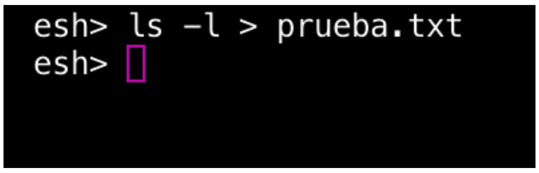
Starting with one single command



```
esh> ls -lhtr
total 124K
-rw-------+ 1 a0363686 alumnos  422 jul 14 19:01 README.md
-rw-------+ 1 a0363686 alumnos  283 jul 14 19:01 Makefile
-rw-------+ 1 a0363686 alumnos 1,1K jul 14 19:01 LICENSE
-rw-------+ 1 a0363686 alumnos 4,2K jul 14 19:01 main.c
-rw-------+ 1 a0363686 alumnos  222 jul 14 19:01 suggestions.h
-rw-------+ 1 a0363686 alumnos 1,7K jul 14 19:01 suggestions.c
-rw-------+ 1 a0363686 alumnos  421 jul 14 19:01 parser.h
-rw-------+ 1 a0363686 alumnos  223 jul 14 19:01 tutorial.h
-rw-------+ 1 a0363686 alumnos  27K jul 14 19:01 tutorial.c
-rwx--x--x+ 1 a0363686 alumnos  52K jul 14 19:57 esh
-rw-------+ 1 a0363686 alumnos 4,2K jul 14 20:08 parser.c
```

Fig. A.8 Single command

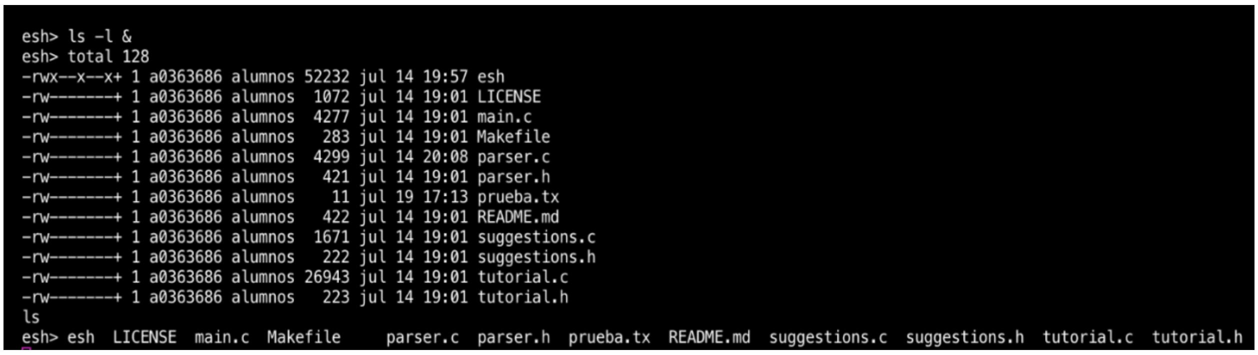
Seems to work fine, so now we are going to see if the redirection works. We are going to try the same command with an output redirection to a file called *prueba.txt*



```
esh> ls -l > prueba.txt
esh> 
```

Fig. A.9 Output redirection

We should try now the background signal to see if we can execute a second command while the first is still running.

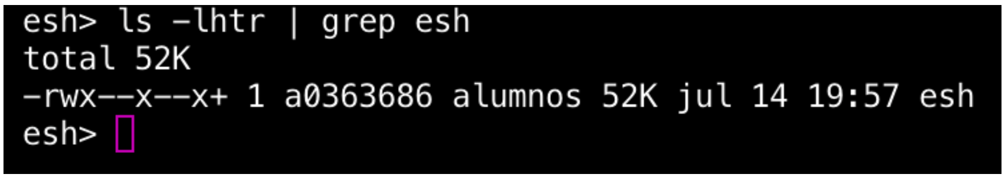


```
esh> ls -l &
esh> total 128
-rwx--x--x+ 1 a0363686 alumnos 52232 jul 14 19:57 esh
-rw-------+ 1 a0363686 alumnos  1072 jul 14 19:01 LICENSE
-rw-------+ 1 a0363686 alumnos  4277 jul 14 19:01 main.c
-rw-------+ 1 a0363686 alumnos   283 jul 14 19:01 Makefile
-rw-------+ 1 a0363686 alumnos  4299 jul 14 20:08 parser.c
-rw-------+ 1 a0363686 alumnos   421 jul 14 19:01 parser.h
-rw-------+ 1 a0363686 alumnos    11 jul 19 17:13 prueba.tx
-rw-------+ 1 a0363686 alumnos   422 jul 14 19:01 README.md
-rw-------+ 1 a0363686 alumnos  1671 jul 14 19:01 suggestions.c
-rw-------+ 1 a0363686 alumnos   222 jul 14 19:01 suggestions.h
-rw-------+ 1 a0363686 alumnos 26943 jul 14 19:01 tutorial.c
-rw-------+ 1 a0363686 alumnos   223 jul 14 19:01 tutorial.h
ls
esh> esh  LICENSE  main.c  Makefile    parser.c  parser.h  prueba.tx  README.md  suggestions.c  suggestions.h  tutorial.c  tutorial.h
```

Fig. A.10 Background signal

As the first command is ls, it finishes its execution very quickly, but it is visible that we have the opportunity to type a second command while the first one is on the background.

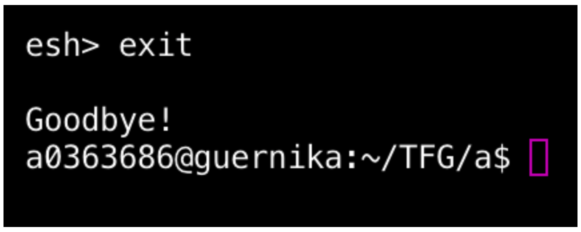
We have two more things that we need to try. The first one is the pipeline and the other one is the exit function



```
esh> ls -lhtr | grep esh
total 52K
-rwx--x--x+ 1 a0363686 alumnos 52K jul 14 19:57 esh
esh> 
```

Fig. A.11 Pipeline

It works correctly, so we are going to check the exit function.



```
esh> exit

Goodbye!
a0363686@guernika:~/TFG/a$ 
```

Fig. A.12 Exit function

If you type exit, the app will finish its execution. If you want to re-launch the app you should write this command on your terminal:

```
./esh
```

Fig. A.13 Command to re-start the app

If the user wants to change the source code, they are free to do it but when the changes are applied, they need to create another executable. The commands to do so, are the following:

```
make clean
make
./esh
```

Fig. A.14 Commands to make a new executable and launch it

A.3 Traceability matrix

TABLE A.1 TRACEABILITY BETWEEN THE EXAMPLES AND THE USER CASES

Figure / User Case	UC1	UC2	UC3
Fig A.4		X	
Fig A.5		X	
Fig A.6	X		
Fig A.7	X		
Fig A.8	X		
Fig A.9	X		
Fig A.10	X		
Fig A.11	X		
Fig A.12			X