Preprogramming

José Mauricio Sevilla[1]
Andrés Camilo Sevilla[1]

[1]Universidad Nacional de Colombia

9 of October of 2015

Review
- What Have We Done?
- Summary
- Special Characters
- Files and Navigation
- Manual

Introduction
- Computer Programming
- Programming Language

Compile vs Interpret
- Compiler
- Interpreter

Pre-programming
- Flow-Chart
- Pseudocode

Programming
- Examples

References

What Have We Done?

We've trying to develop some concepts and learn how to use some tools, that were making sense after some examples.

It's important to us, keep in mind our goal, and try to use the concepts we're trying to build.

Let's see what we have learnt.

Summary

At first, we saw some differences between the GUI (**G**raphical **U**ser **I**nterface) and CLI (**C**ommand **L**ine **I**nterface).

The first one, is natural for us, we've been dealing with it since ever, but start to use the command user interface, isn't natural at all, but it is for programming.

Special Characters

There are some characters that have a special importance, that's why we are interested in studying them:

Character	Description	Character	Description
'\'	Escape character	'/'	Directory separator
'.'	Current directory	'..'	Parent directory
'~'	Home directory	'*'	Bonus character[1]
'?'	Bonus character [2]	'[ ]'	Range of values
'|'	"Pipe"	'>'	Redirect output
'>>'	Redirect output	'<'	Redirect input
';'	Command separator[3]	'&& '	Command separator[4]
'&'	Run in background	'#'	Comment

[1] 0 or more characters.

[2] A single character.

[3] Execute multiple commands on a single line

[4] Runs the second command if the first one finish without errors

Files and Navigation

Manage data is a very important thing, so we need to learn how to move in folders and some instructions that will help us.

Command	Description
`pwd`	"Print Working directory"
`cd`	"Change Directory"
`ls`	"List"
`mv`	"Move"[5]
`cp`	"Copy"
`rm`	"Remove"
`mkdir`	"Make Directory"
`rmdir`	"Remove Directory"

[5] It's used to rename files too.

Manual

Most of programs/commands have a manual page where we can find a description of their use and options. for example, if we type `man ls`, obtain:

```
jmsevillam@jmsevillam-desktop: ~
LS(1)                          User Commands                          LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List  information  about  the FILEs (the current directory by default).
       Sort entries alphabetically if none of -cftuvSUX nor --sort  is  speci-
       fied.

       Mandatory  arguments  to  long  options are mandatory for short options
       too.

       -a, --all
              do not ignore entries starting with .

       -A, --almost-all
              do not list implied . and ..

       --author
 Manual page ls(1) line 1 (press h for help or q to quit)
```

Figure 1: `man ls` - Manual of the command `ls`

As an advice is recommended that the first thing we've got to do when learn a new command is read its manual, at least its description and some possible arguments.

Often the manual has lots of information that may get things complicated, so over the time we have to find our favorites commands for the utilities we need.

Introduction (Programming)

Definition 1

Programming (or computer programming)is the process of give some instructions to the computer and obtain results from those.

The idea of programming is deeply attached to the concept of organization; Basically is organize the ideas and translate them to the computer.

Is clear that the computer *does not* "understand" our language, here is where we notice the need of "*translate*" our language into the machine one.

Programming Language

So, naturally arise the concept of "*Programming Language*" as follows:

Definition 2

A programming language is a set of grammatical rules for instructing a computer to perform specific tasks.

Computer is a machine able to do many calculations per second, and we have to learn how to take advantage of that, so, what we have to learn is how to "*say*" the things we want to the PC.

Examples

- Fortran
- Java
- C/C++
- Python

As our personal goal we have understand what's the difference between some of those languages, how do they work, and which one we should use.

There are lots of ways to classify the languages, the idea is how much similar are to our language, those that are "*similar*" are called *High-level Languages* and those that are more like the one the machine understand are called *Machine Languages* or *Low-level Languages*.

The machine language is an idea we are not going to develop any longer, so are going to get focused on two particular languages, `C/C++` and `python`.

Compile vs Interpret

Regardless of what language we choose, at some point we have to convert our program into a machine language, so the computer could understand it, and there are two ways to do that:

1. Compile
   - `C/C++`
2. Interpret
   - `Python`

So, we write down the instructions into a text file, and depending of the kind of language we compile or interpret our *code* so we can *run* it.

Compiler

A compiler is a program that takes a source code written in a high-level programming language and transforms it into an lower-level language (Often having a binary form known as "*Object Code*"), creating a executable program.

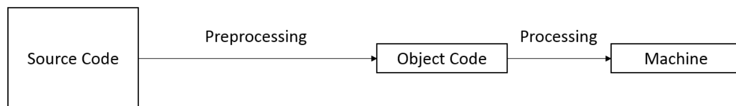

Figure 2: Compiler process

Interpreter

An interpreter is a program that directly executes the instructions without the need of previously compiling them into a machine language.

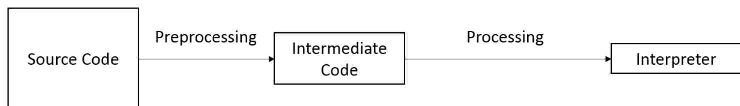

Figure 3: Interpreter process

Compiler	Interpreter
Scans the entire program and translates it as a whole.	Translate program one statement at a time.
It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.	It takes less amount of time to analyze the source code but the overall execution time is slower.
Generates intermediate object code which further requires linking, hence requires more memory.	No intermediate object code is generated, hence are memory efficient.
It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.	Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.

Table 1: Compiler vs Interpreter

Pre-programming

Almost as important as programming is, the pre-programming is a crucial part of any project that includes programming.

Before we star to program we need to have keep in mind our goal, and try to divide it in little tasks.

Is more efficient write a program in the simplest way (even if it isn't all we need from it), and once it works, include more of those little tasks.

There a lots of strategies or methods to do that, we are going to study two of them, not so deeply, but at least, we are going to mention them.

Flow Chart

The *Flow Chart* or *Flow Diagram* is a graphical way to organize ideas or process as instance, programming. Different symbols are used and every one has a different meaning like: start/end, processing, decision, input/output.

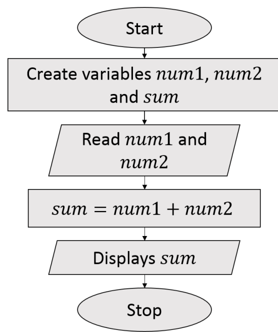

Figure 4: Flowchart for a sum

This is an example of a simple Flowchart representing a simple program, in this way we can understand in a deep way the problem we want to program.

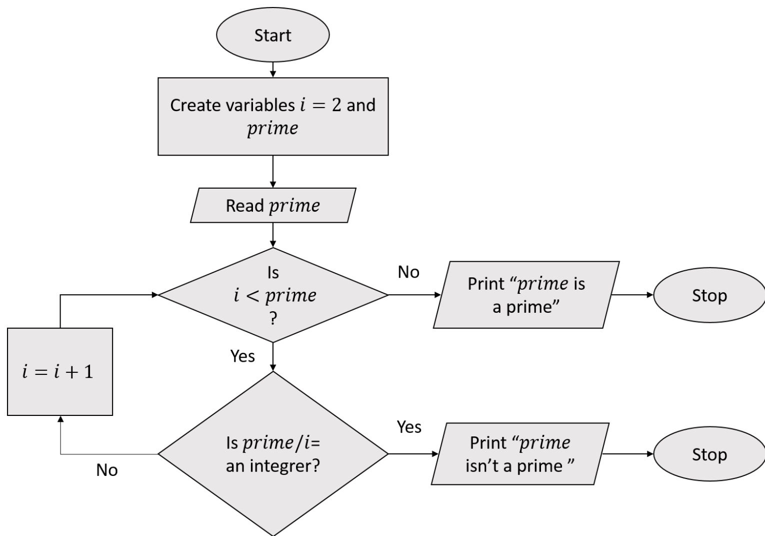

Figure 5: Flowchart to verify if a number is prime

Pseudocode

The pseudocode is in the middle between our language and the programming language, so, the idea is trying to write down what we want to our program to do, and then write the same but changing things making it more similar to a program language, for example:

i an index from 0 to 10

`C/C++`:

i starts as 0, until $i = 10$, increasing 1 at a time

`python`:

i goes from 0 to 10

As an advice, is recommended using pseudocode before programming, at first it can take some time, but as a complete work, it's better due to it makes programming easier and comfortable to correct if any error occurs.

Programming

At first, we need to understand some things, the parts of every program, syntax, kind of variables and others that we will show in the next section with some examples. Let us start with a C++ code:

```
//comments
#include<``libraries''>

.

int main(){

.

return 0;}

.
```

Adding libraries

Function definition

Main definition

Program body

End of program body

Additional functions

Examples

```
#include <iostream>
using namespace std;
int main(){
cout<<'Hello World'<<endl;
return 0;
}
```

or

```
#include<iostream>
int main(){
std::cout<<'Hello World'<<std::endl;
}
```

Examples

```c
#include <stdio.h>
/* Block
 * comment */
int N=10;
int main(){
    int i;
    // Line comment.
    printf("Hello world!");

    for (i = 0; i < N; i++){
        printf("int: %d", i);
    }
return 0;
}
```

Example

We could keep showing lot's of examples, but, let's make our program.
Let's make a program to sum two numbers like in figure (4)

So, first, we need to use a text editor to write the code, but, which should we use?
There's a lot of them, some use the GUI and others the CLI, for example:

- `emacs`
- `gedit`
- `vi`
- `vim`
- `eclipse`
- `geany`

How to Write

So,the first thing we've got to do is open a text editor, would be easy open it from the GUI, but we can do it from shell too, that's done as follows: To write a code, we could use a graphical interface by writing on the prompt:

```
program filename
```

For example, if we want to open (create) a file named `data.dat` by using `gedit`

```
gedit dada.dat
```

which opens the file `data` in a new window using the GUI, but, although the shell is still opened, we can not use it until we close `gedit`. We can solve it, by using the character `&` after the file name.

If we want to open a text editor that uses the shell, we can use `vim`, and it's open as shown before but with `vim` instead `gedit`.

> If we open `vim` and try to write on the file what do we get?

`vi` and `vim` use some commands to use it, to close it, let's press `Esc` and then type:

```
:q
```

Let's check out the manual!!

References

PROGRAMIZ.COM
http://www.programiz.com/article/difference-compiler-interpreter

C4LEARN.COM
http://www.c4learn.com/c-programming/compiler-vs-interpreter/

CERN
https://twiki.cern.ch/twiki/pub/Main/CollaborativeProjects/Shell.pdf, 2015