UNIVERSIDAD POLITÉCNICA

DE YUCATÁN

ALGORITHMS FUNDAMENTALS

UNIT I

RESEARCH

PRESENTED BY

JULIO ERNESTO DZUL DZIB

DATA 1°A

PROFESSOR

ING. LUIS GERARDO CÁMARA SALINAS

JANUARY, TUESDAY 11TH

STAGES OF PROGRAM COMPILATION AND LEVELS OF PROGRAMMING

Programming nowadays it is easier to understand for people; even though it might look hard or overwhelming for starters, it is way easier than how it was decades ago.

Normally we code in a programming language such as python, c, c++, java, etc. which are high-level programming languages. Levels of programming basically mean how abstract and easier to understand is a language compared to its hardware, the lower the level is, the less understandable for humans is, and will be more understandable for computers and optimized.

Low level programming languages are machine language and assembly language.

- Machine language is the fundamental language of a processor, written in binary (1 and 0 that represents high and low voltage, working with logical gates.). It is barely understandable for humans, only machines understand it.
- Assembly language is more user-friendly, and, instead of working with binary, works with symbolic language to represent binary code. It has some commands, such as mov, add, sub, etc. and some logic operators.

None of these low-level languages are portable as it is hardware specific. Only the memory of the CPU the program was programmed for keeps it.

High-level machine languages are less optimized, but way easier to understand for the user. In these we have more options than in low level languages, we have easy to write variables, conditions, types of data, libraries and more. Often these have one or multiple IDE's that make it even way easier for us, getting definitions and completing functions.

The thing is that machines do not understand high-level languages, so these languages, when compiled, translate the source code that we write into machine code so that the processor will follow the instructions.

Each high-level language has its own compilator, however, all compilators have the same goal. Compilation is a multistage process, evaluating the code, transforming it to assembly code and then to machine code, as it follows:

Preprocessing

The first stage of compilation is preprocessing. The compiler will run preprocessor command. This does different tasks such as a

- **Lexical analysis**, that will load the libraries code, remove all comments and unnecessary spaces, and change keywords, constants, and identifiers by 'tokens', which are symbolic strings to identify what the elements are.

- **Symbol table construction**, that will create a table to store variables, constants, and arrays
- **Syntax analysis**, that will check if tokens match the syntax of the language and will throw error messages if they don't match.
- **Semantic analysis**, that will check variables declaration and data types, as well as operations.

Compilation

Not to confuse it with the process name, this is the second stage of compilation. There, the preprocessed code will be transformed into assembly code, that will match the operator system and processor architecture. This step is why even high-level languages aren't portable unless they are in the same OS, with few exceptions like Java.

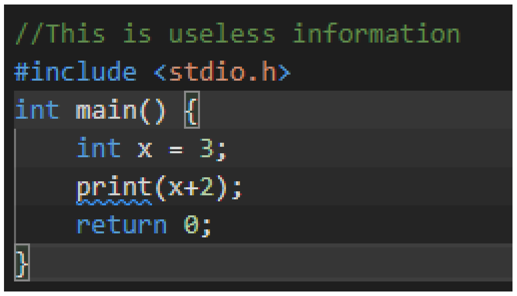
Assembly

The third stage of the process, an assembler it will transform the assembly code to object code (Binary, machine code, but in no particular order or with connections missing)

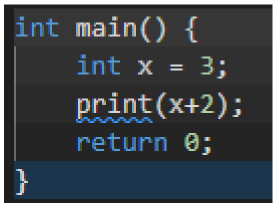
Linking

Linking is the last stage of the compilation. To create an executable file, all the different objects will have to be linked; rearranged and filled in if necessary, so that the will call the correct functions in the others.

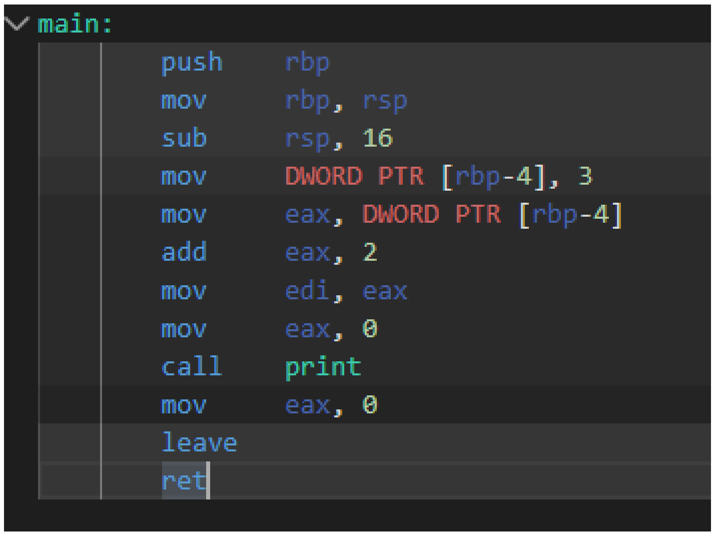
One example for the stages could be the following code in c.



```
//This is useless information
#include <stdio.h>
int main() {
    int x = 3;
    print(x+2);
    return 0;
}
```



```
int main() {
    int x = 3;
    print(x+2);
    return 0;
}
```



```
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], 3
        mov     eax, DWORD PTR [rbp-4]
        add     eax, 2
        mov     edi, eax
        mov     eax, 0
        call    print
        mov     eax, 0
        leave
        ret
```

```
554889E54883EC10C745FC030000008B45FC83C00289C7B800000000E800000000B800000000C9C3
```

(Sadly, I couldn't find an assembler to transform the code in binary, only in hex so that it won't use so much space)

REFERENCES

BBC. (2021). *Compilers, interpreters and assemblers*. Retrieved from BBC Bitesize: https://www.bbc.co.uk/bitesize/guides/zmthsrd/revision/3

Duncan, P. O. (2017). *How do computers read code?* Retrieved from Youtube: https://www.youtube.com/watch?v=QXjU9qTsYCc

HowTo. (2015). *Understanding C program Compilation Process*. Retrieved from Youtube: https://www.youtube.com/watch?v=VDslRumKvRA

JavaTPoint. (2021). *What is a programming language?* Retrieved from JavaTPoint: https://www.javatpoint.com/classification-of-programming-languages

Luks, C. (2015). *The Four Stages of Compiling a C Program*. Retrieved from calleluks: https://www.calleluks.com/the-four-stages-of-compiling-a-c-program/