Universidad Nacional Autónoma de México

Ingeniería en Computación

Compiladores

Practice

ALUMNOS:
320021312
320112179
320184624
320565661
423093137

Grupo:
5
Semestre:
2026-I

México, CDMX. Agosto 2025

Contents

1 Introduction

Understanding the internal functioning of compilers, interpreters, and assemblers is important because they are the basis for efficient and functional software development. However, there is often confusion between these three elements due to their similar purpose of translating code from one form to another.

Clarifying the differences between these tools and analyzing real-world examples allows us to appreciate the complexity of programming languages and how source code becomes executable and "understandable" for computers. This understanding is fundamental for debugging, optimization, and system-level programming.

In this work, we seek to identify the input, grammar structure, and intermediate code of compilers, interpreters, and assemblers, with examples of each case, with the objective of understanding these three concepts and learning why they're so important in software development.

2 Theoretical framework

Any program written in a high-level language is known as source code. However, computers cannot understand source code. Before it can be executed, the source code must first be translated so that the CPU can understand it. This translation into machine language is performed by three types of language processors:

- **Compiler:** A compiler reads the whole source program that has been written in a high-level language code and translates it into machine code or object code. It performs lexical analysis, syntax analysis, semantic analysis, optimization, and code generation during this process. [1]
- **Interpreter:** An interpreter directly executes instructions written in a programming language without converting them into machine code beforehand. It parses and executes the program line by line, making it more suitable for scripting languages. [1]
- **Assembler:** An assembler translates a program written in assembly language into machine code. Assembly language is a low-level language that closely maps to a computer's instruction set since its the first interface that allows humans and computers to communicate with each other. [1]

3 Development

In this section we describe the selected examples of compilers, interpreters and assemblers. For each case, we identify the required input, the main elements of its grammar, and the intermediate code it generates.

3.1 Compilers:

3.1.1 GCC (GNU Compiler Collection)

Input: GCC receives as input source code written in high-level languages such as C, C++, Objective-C, Fortran, Ada, Go, among others. The input files usually have extensions like `.c`, `.cpp`, `.f90`, etc. [2]

Main grammar elements:

- **Lexical tokens:** keywords such as `if`, `while`, `return`, identifiers, numeric constants, operators, and delimiters.
- **Grammar productions (BNF):** GCC follows the grammar defined by the language standards (for example, ISO C11 or ISO C++17).
- **Main non-terminals:** declaration, expression, statement, function-definition.
- **Abstract Syntax Tree (AST):** represents the hierarchical structure of the source code.

Intermediate code: GCC translates the source code into an *Intermediate Representation (IR)* called **GIMPLE**, which is a simplified three-address code. Then, GIMPLE is transformed into **RTL (Register Transfer Language)**, a representation closer to hardware, before generating machine code. [2]

3.1.2 Clang (part of LLVM)

Input: Clang receives source code in C, C++, and Objective-C. The input files usually have extensions such as `.c`, `.cpp`, and `.m` (for Objective-C). [3]

Main grammar elements:

- **Lexical tokens:** keywords, identifiers, literals, operators.
- **Grammar productions (EBNF):** Clang implements the grammar defined by the ISO standards of C and C++.
- **Main non-terminals:** declaration, expression, initializer, statement.
- **Abstract Syntax Tree (AST):** Clang generates an abstract syntax tree enriched with metadata, which is used for analysis and external tools (such as `clang-tidy`).

Intermediate code: Clang produces the **LLVM Intermediate Representation (LLVM IR)**, a low-level three-address code that is machine-independent. LLVM IR can be represented in:

- **Textual form** (`.ll`)
- **Binary form** (`.bc`)

3.2 Interpreters:

3.2.1 Python interpreter (CPython)

Input: The python interpreter receives python source code (.py files) as input.

Main grammar elements:

- **Lexical Tokens:** Python lexical structure is formed by comments, variables, literals, operators, delimiters and keywords. [4]
- **Python indentation:** Python uses indentation to indicate a block of code and not for readability purposes only.
- **Grammar Definition:** Python grammar is defined in a mixture of EBNF and PEG notations.
- **Interactive interpreter:** Each line of code is parsed, compiled and executed when it is read by the interpreter.

Intermediate code: The interpreter generates an Abstract Syntax Tree which then is compiled into bytecode. This bytecode is the intermediate code generated by the interpreter stored in a .pyc file. It is an abstraction of the machine code that is then translated to it and executed by the Python Virtual Machine (PVM). [5]

3.2.2 JavaScript Engine (V8)

Input: The V8 engine receives JavaScript source code as input, used in web applications or server-side scripts (Node.js).

Main grammar elements:

- **Lexical Tokens:** JavaScript lexical grammar is defined by the ECMAScript specification and includes identifiers, reserved keywords, literals (numeric, string, boolean, null, undefined), punctuators (e.g., , ;, ,), and operators. [6]
- **Syntactic Grammar:** Based on ECMAScript's context-free grammar rules, it defines statements (if, for, while), declarations, expressions, and function structures. [6]
- **Automatic Semicolon Insertion (ASI):** JavaScript grammar allows omission of semicolons; the engine automatically inserts them where needed according to strict rules. [6]
- **Prototype-based semantics:** JavaScript grammar includes rules for object creation and inheritance through prototypes. [6]

Intermediate code: The V8 engine first parses the code into an Abstract Syntax Tree (AST). From this AST, it produces Ignition bytecode, which is an intermediate representation optimized for interpretation. Frequently executed sections of bytecode are then compiled by the TurboFan optimizing compiler into highly efficient machine code, allowing for fast execution while preserving flexibility. [6]

3.3 Assemblers:

3.3.1 GNU Assembler

Input: The GNU assembler is intended to assemble the output of the GNU C compiler (gcc) for use by the linker. Its input is assembly language source code. [7]

Main grammar elements:

- **Lexical Tokens:** GNU Assembler main tokens are identifiers, mnemonics, registers, literals, values, operators, labels and comments.
- **Mnemonic based:** Its grammar follows a mnemonic based pattern, where each one represents a CPU instruction.
- **Labels:** Are used to mark a location in the code.
- **Directives:** These are instructions for the assembler itself and not the cpu, can be used to define constants and data values.

Intermediate code: The GNU Assembler does not generate intermediate code, instead it generates the machine code directly in a .o file which is then executed by the CPU.

3.3.2 NASM Assembler

Input: The NASM (Netwide Assembler) receives assembly source code files (.asm) as input, targeting the x86 and x86-64 instruction set architectures.

Main grammar elements:

- **Instruction Mnemonics:** Human-readable representations of machine instructions (e.g., MOV, ADD, JMP) that correspond to CPU operations. [8]
- **Operands:** Registers (e.g., EAX, EBX), immediate values (constants), and memory references, which form part of the instruction syntax. [8]
- **Labels:** Symbolic names that represent memory addresses or instruction locations within the program.
- **Directives:** Non-executable instructions that guide assembly, such as SECTION .data (data segment) and SECTION .text (code segment). [8]

Intermediate code: NASM translates assembly code directly into relocatable object code (.o or .obj files), which contains binary machine instructions and symbolic references. This object code is considered intermediate because it must be linked with other object files and libraries by a linker before generating the final executable. Unlike interpreters, NASM produces code that is very close to the final machine representation. [8]

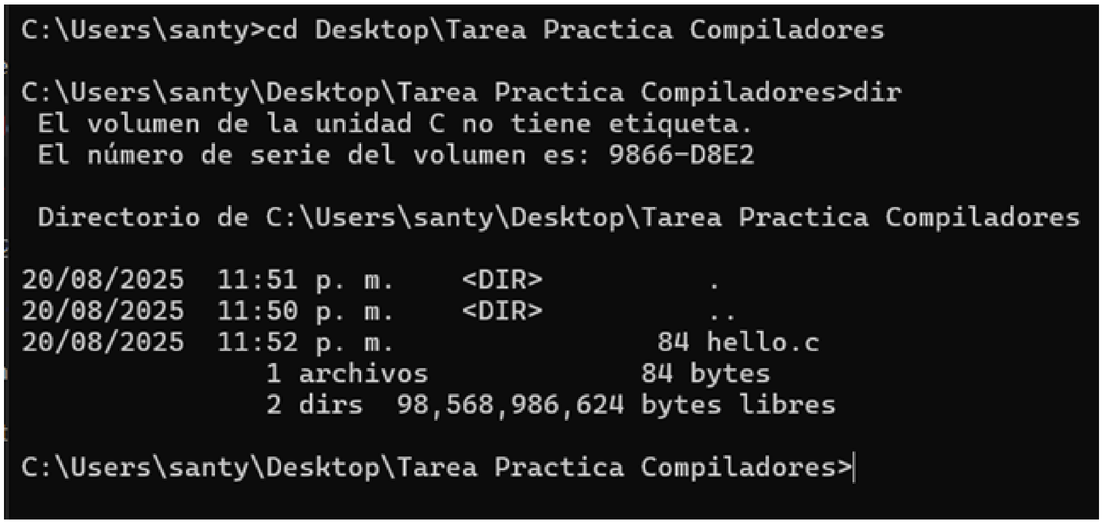
4 Results

4.1 Compilers:

4.1.1 GCC (GNU Compiler Collection)

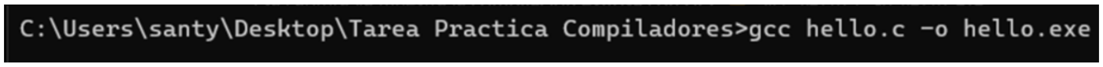
To test the **GCC** compiler, a simple C program named `hello.c` was used. The procedure is summarized below:

1. The directory containing `hello.c` was opened (Figure 1a).
2. The program was compiled with `gcc hello.c -o hello.exe`, generating the executable `hello.exe` (Figure 1b).
3. The executable was run with `hello.exe`, producing the expected output `"Hello, GCC!"` (Figure 1c).



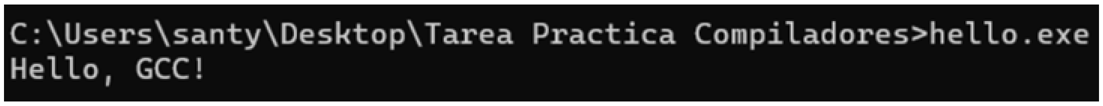
```
C:\Users\santy>cd Desktop\Tarea Practica Compiladores

C:\Users\santy\Desktop\Tarea Practica Compiladores>dir
 El volumen de la unidad C no tiene etiqueta.
 El número de serie del volumen es: 9866-D8E2

 Directorio de C:\Users\santy\Desktop\Tarea Practica Compiladores

20/08/2025  11:51 p. m.    <DIR>          .
20/08/2025  11:50 p. m.    <DIR>          ..
20/08/2025  11:52 p. m.                84 hello.c
               1 archivos             84 bytes
               2 dirs  98,568,986,624 bytes libres

C:\Users\santy\Desktop\Tarea Practica Compiladores>
```

(a) Directory showing the source file `hello.c`. (Capture elaborated by the author).



```
C:\Users\santy\Desktop\Tarea Practica Compiladores>gcc hello.c -o hello.exe
```

(b) Compilation with **GCC** using `gcc hello.c -o hello.exe`. (Capture elaborated by the author).



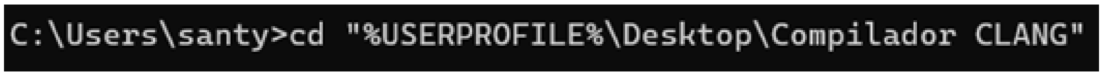
```
C:\Users\santy\Desktop\Tarea Practica Compiladores>hello.exe
Hello, GCC!
```

(c) Execution result showing `Hello, GCC!`. (Capture elaborated by the author).

Figure 1: GCC test procedure and outputs.

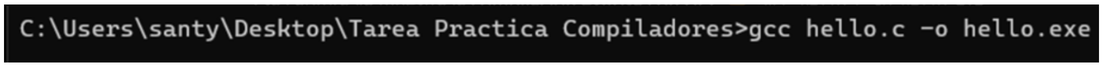
4.1.2 Clang (LLVM Compiler Infrastructure)

To test the **Clang** compiler, the program `hello_name.c` was compiled and executed as follows: (a) directory with the source file, (b) compilation command, and (c) program output.



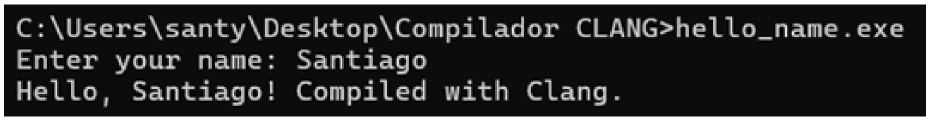
```
C:\Users\santy>cd "%USERPROFILE%\Desktop\Compilador CLANG"
```

(a) Directory with `hello_name.c`. (Capture elaborated by the author).



```
C:\Users\santy\Desktop\Tarea Practica Compiladores>gcc hello.c -o hello.exe
```

(b) Compilation with **Clang**. (Capture elaborated by the author).



```
C:\Users\santy\Desktop\Compilador CLANG>hello_name.exe
Enter your name: Santiago
Hello, Santiago! Compiled with Clang.
```

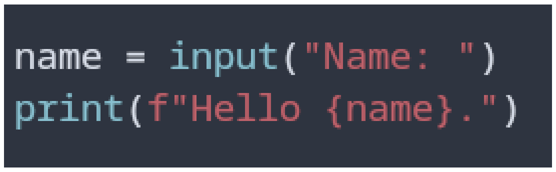
(c) Execution of `hello_name.exe`. (Capture elaborated by the author).

Figure 2: Clang test procedure and outputs.

4.2 Interpreters:

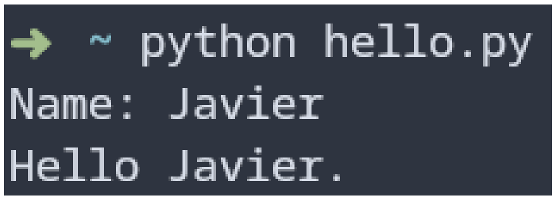
4.2.1 Python interpreter (CPython)

To test the Python interpreter the file hello.py was executed showing the following input:



```
name = input("Name: ")
print(f"Hello {name}.")
```

(a) Python interpreter input. `hello.py` file. (Capture elaborated by the author).



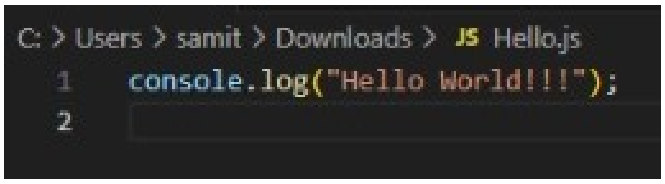
```
➜  ~ python hello.py
Name: Javier
Hello Javier.
```

(b) Execution of `hello.py`. (Capture elaborated by the author).

Figure 3: Python interpreter test procedure and outputs.

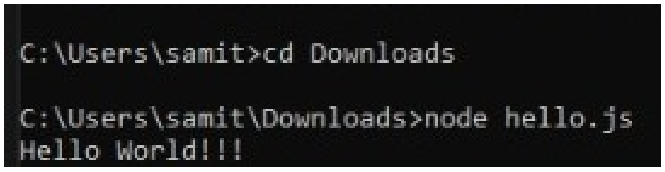
4.2.2 JavaScript Engine (V8)

To test the javaScript Engine, we used Node.js, which already has V8 by default, then created a simple file called hello.js, which has the following input:



```
C: > Users > samit > Downloads > JS Hello.js
console.log("Hello World!!!");
```

(a) JavaScript interpreter input. `hello.js` file. (Capture elaborated by the author).



```
C:\Users\samit>cd Downloads

C:\Users\samit\Downloads>node hello.js
Hello World!!!
```

(b) Execution of `hello.js`. (Capture elaborated by the author).

Figure 4: JavaScript interpreter test.

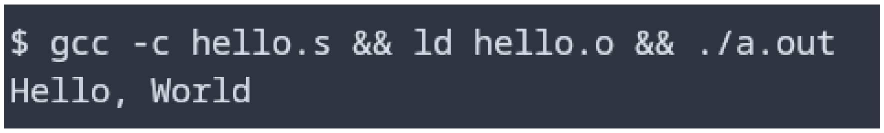
4.3 Assemblers:

4.3.1 GNU Assembler

The hello.s file will be assembled and executed using GNU Assembler.

```
_start:
        # write(1, message, 13)
        mov     $1, %rax                # system call 1 is write
        mov     $1, %rdi                # file handle 1 is stdout
        mov     $message, %rsi          # address of string to output
        mov     $13, %rdx               # number of bytes
        syscall                         # invoke operating system to do the write

        # exit(0)
        mov     $60, %rax               # system call 60 is exit
        xor     %rdi, %rdi              # we want return code 0
        syscall                         # invoke operating system to exit
message:
        .ascii  "Hello, world\n"
```

(a) GNU Assembler source code. `hello.s` file. [9]



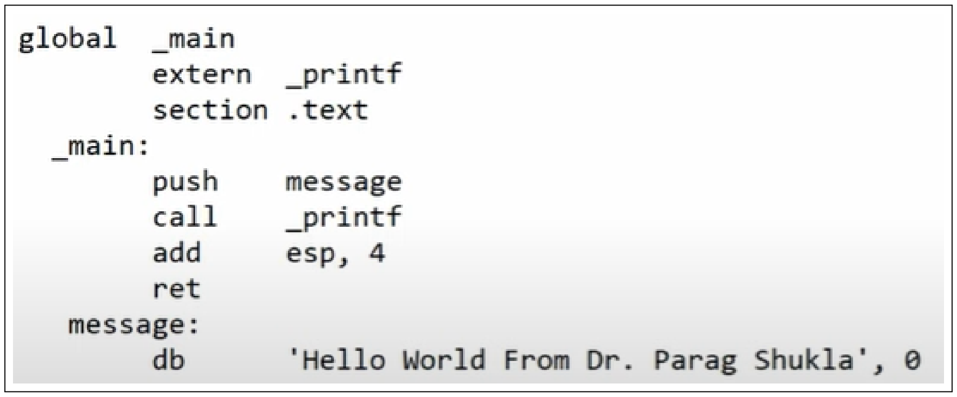
```
$ gcc -c hello.s && ld hello.o && ./a.out
Hello, World
```

(b) Execution of `hello.s`. [9]

Figure 5: GNU Assembler test procedure and outputs.

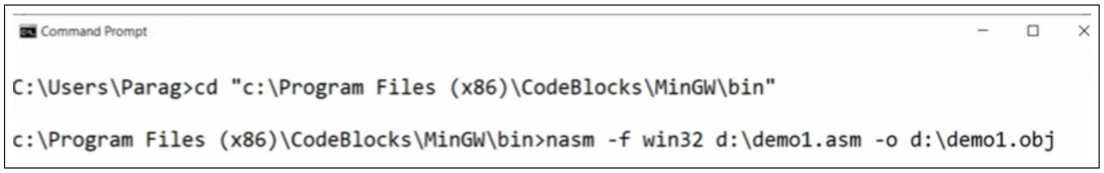
4.3.2 NASM Assembler

The demo1.asm file will be assembled using NASM and compiling using GCC. This test was obtained from a Youtube video of the channel "Dr. Parag Shukla". [10]



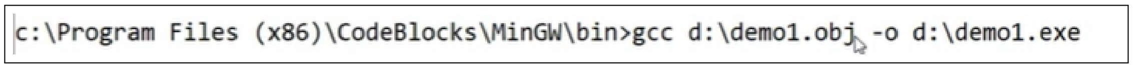
```
global  _main
        extern  _printf
        section .text
  _main:
        push    message
        call    _printf
        add     esp, 4
        ret
   message:
        db      'Hello World From Dr. Parag Shukla', 0
```

(a) NASM Assembler source code. `demo1.asm` file. [10]



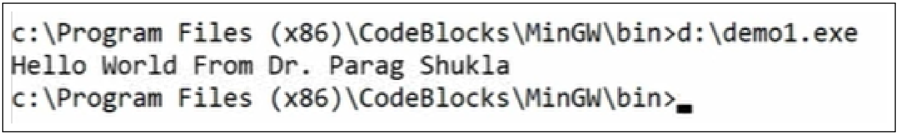
```
C:\Users\Parag>cd "c:\Program Files (x86)\CodeBlocks\MinGW\bin"

c:\Program Files (x86)\CodeBlocks\MinGW\bin>nasm -f win32 d:\demo1.asm -o d:\demo1.obj
```

(b) Creation of the object file. `demo1.obj`. [10]



```
c:\Program Files (x86)\CodeBlocks\MinGW\bin>gcc d:\demo1.obj -o d:\demo1.exe
```

(c) Creation of the exit file from the object file. [10]



```
c:\Program Files (x86)\CodeBlocks\MinGW\bin>d:\demo1.exe
Hello World From Dr. Parag Shukla
c:\Program Files (x86)\CodeBlocks\MinGW\bin>
```

(d) Execution of `demo1.exe`. [10]

Figure 6: NASM Assembler test procedure and outputs.

5 Conclusions

The practical examination of compilers, interpreters, and assemblers reinforced the theoretical distinctions between these systems by exposing their operational characteristics in real-world environments. Through the implementation and testing of specific tools such as GCC, Clang, CPython, the V8 engine and NASM, the differences between compilers, interpreters and assemblers became clearer, the principal characteristic that we noticed was the fact that interpreters did not create any extra files for the machine code like the compilers did, and the fact that each element operates at a different level of the programming language hierarchy, ranging from high-level source code to low-level machine instructions.

This exercise emphasized how theoretical concepts such as grammar parsing, lexical analysis, and code generation directly apply when interacting with actual programming tools. The observed behaviors validate the theoretical frameworks introduced in class. Understanding these differences is important for practical tasks such as performance tuning, system design, and debugging at a low level.

References

[1] GeeksforGeeks. "Language processors: Assembler, compiler and interpreter." (Jul. 2025), [Online]. Available: `https://www.geeksforgeeks.org/computer-science-fundamentals/language-processors-assembler-compiler-and-interpreter/`.

[2] Free Software Foundation. "Using the gnu compiler collection (gcc): For gcc version 13.1.0." (2023), [Online]. Available: `https://gcc.gnu.org/onlinedocs/`.

[3] LLVM Project. "Python syntax." (2024), [Online]. Available: `https://llvm.org/docs/LangRef.html`.

[4] J. Bodnar. "Python lexical structure." (2024), [Online]. Available: `https://zetcode.com/python/lexical-structure/`.

[5] W3 Schools. "Python syntax." (nd), [Online]. Available: `https://www.w3schools.com/python/python_syntax.asp`.

[6] E. International. "Ecmascript language specification." (2025), [Online]. Available: `https://tc39.es/ecma262/`.

[7] GNU Assembler. "Using as." (nd), [Online]. Available: `https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_chapter/as_toc.html`.

[8] NASM. "Nasm - the netwide assembler." (nd), [Online]. Available: `https://www.nasm.us/doc/`.

[9] L. M. University. "Gnu assembler examples." (nd), [Online]. Available: `https://cs.lmu.edu/~ray/notes/gasexamples/`.

[10] D. P. Shukla, *Download and Install NASM and Write and Test first Assembly Program on Windows - Practical Demo*, Mar. 2022. [Online]. Available: `https://www.youtube.com/watch?v=xyCQ8O9bRXU`.