



Universidad Nacional Autónoma de México

Computer Engineering

Compilers

PRACTICE

Students:

320013447

320280087

320341704

423032833

423529834

Group:

#5

Semester:

2026-I

Mexico, CDMX. August 21, 2025

Contents

1	Introduction	2
2	Theoretical Framework	2
3	Development	3
3.1	Compilers	3
3.2	Interpreters	4
3.3	Assemblers	4
4	Results	5
4.1	Compilers	5
4.2	Interpreters	6
4.3	Assemblers	8
5	Conclusions	9

1 Introduction

This report gives a comparative analysis of three key programs in programming language processing: **compilers, interpreters and assemblers**.

Problem statement: In programming, there are different translators that change human-written code into instructions that are understandable by the computer. But compilers, interpreters, and assemblers aren't the same. They differ in inputs, grammar, execution rules and outputs. The main problem is to understand those differences: how each tool processes the programs it receives, what lexical, syntax, and semantic rules they follow and what kind of output they produce.

Motivation: Comparing compilers, interpreters and assemblers is essential to really get how programming languages and software life cycles work, from source code to machine execution. This understanding not only makes it easier to learn programming techniques and build more efficient software, but is also key in things like optimization, cross-platform portability and program security.

Objectives: The main goal is to analyze and compare compilers, interpreters, and assemblers using examples: GCC and Javac (compilers), CPython and PHP Zend Engine (interpreters), NASM and GAS (assemblers). We'll look at their inputs, grammar rules (lexical, syntax, semantic), and outputs or intermediate representations, to clearly see their similarities and differences.

2 Theoretical Framework

In computer science, programming language translators are key tools to turn source code into machine-executable representations. They fall into three main categories: compilers, interpreters, and assemblers, each with its own purpose and abstraction level. [16]

Compiler: Translates mid/high-level languages (like C or Java) into intermediate code or directly into machine code. Phases usually include lexical, syntax, semantic analysis, code generation, optimization, and translation to object or executable code. Examples: GCC (C/C++ to machine code) and Javac (Java to JVM bytecode). [15]

Interpreter: Runs code directly, usually generating intermediate code (like bytecode) for a virtual machine. Examples: CPython (Python to bytecode) and Zend Engine (PHP to opcodes). [3]

Assembler: Works with low-level assembly language, close to the hardware, turning mnemonics (like `mov`, `add`, `jmp`) into binary object code. Examples: NASM (Intel syntax) and GAS (AT&T syntax, used by GCC). [7]

Grammar: Defines the lexical, syntactical, and semantic rules that determine the valid structure of programs. [16]

Intermediate code: Abstract representation of the program, used for optimization and portability before generating the final code. [8]

Object files: Intermediate binaries produced by compilers or assemblers, which require linking to form an executable. [6]

3 Development

3.1 Compilers

C (GCC): Input: Source code in C (mid-level language). Grammar: Based on the ISO C standard.

- **Lexical rules:** Uses tokens as identifiers (letters/_ + digits), keywords (**if**, **for**, **return**, ...), literals (integers, floats, strings, characters), operators (+, -, *, /, %, &&, ||, &, |, ^, «, »), and punctuators (;, { }, (), []).
- **Syntactic rules:** Functions with return types, with a main function called **main** (for executable programs). Blocks delimited with { ... }. Global declarations and multiple functions are possible. Variables declared with type.
- **Semantic rules:** Static typing with simple types (int, char, ...) and derived types (structs, pointers). Automatic conversion to expressions and explicit *cast* for conversions from higher to lower range.

The compiler performs preprocessing (#include, #define, macros), lexical and syntactical analysis, generates intermediate code (GIMPLE/RTL), applies optimizations, and finally emits object code (assembler or binary).

Java (Javac): Input: Source code in Java (an object-oriented language). Grammar: Defined by the *Java Language Specification (JLS)*. [11]

- **Lexical rules:** Tokens such as identifiers (letters/_/ \$ + digits), keywords (**class**, **if**, **for**, **return**, ...), literals (integers, floats, booleans, null, strings, characters), operators (+, -, *, /, &&, ||, **instanceof**), and separators (;, { }, (), [], .). [11]
- **Syntactic rules:** Class and method-based structures; signed methods (**type name(parameters) { ... }**). Control statements (if, for, while, switch), package declarations and imports. Entry point: **main**, not required for libraries. [11]
- **Semantic rules:** Static typing with primitive types (int, float, etc.) and reference types (string, classes). Implicit conversion to primitives (widening), explicit *cast* for narrowing. Supports inheritance and polymorphism. [11]

The compiler parses according to the JLS, performs semantic checks and generates JVM bytecode (`.class`).

3.2 Interpreters

Python (CPython): Input: Python source code. Grammar: Defined in the *Python Language Reference*. [13]

- **Lexical rules:** Literals (integers, floats, single/triple strings, bytes), operators (+, -, *, /, //, %, **, and, or, not), delimiters (:, ,, (), [], {}), Indentation is significant and defines blocks. [13]
- **Syntax rules:** Definitions with `def` and `class`, blocks with: followed by indentation, expressions and statements (`if/elif/else`, `for`, `while`, etc.), list comprehensions, sets, and dictionaries. [13]
- **Semantic rules:** Dynamic typing. Variables are typed at runtime; everything is an object. Automatic conversion to expressions. No integer overflow. [13]

CPython tokenizes and parses the code, generates bytecode (`.pyc`) and executes it on its virtual machine. `.pyc` files are stored in `__pycache__` to speed up future executions.

PHP (Zend Engine): Input: PHP (web scripting language) code. Grammar: Documented at php.net and the formal `php-langspeg` project. [12]

- **Lexical rules:** identifiers (variables with `$`), reserved words (`function`, `class`, `if`, `echo`, ...), literals (numbers, strings, heredoc/nowdoc), operators (+, -, *, /, .., ==, ===, ??, ?:, ...), delimiters (;, { }, (), [], ->). [12]
- **Syntactic rules:** functions with optional types in parameters/returns. Doesn't require `main`; execution begins in the invoked file. Blocks delimited with { } or with alternative syntax (`: ... endif;`). PHP code goes inside `<?php ... ?>`. [12]
- **Semantic rules:** Dynamic and lax typing. Automatic type conversion (*juggling*). Weak comparisons (`==`) convert types; strict comparisons (`===`) require an exact match. [12]

The interpreter parses, builds an AST and generates opcodes (`op_array`) that the Zend VM executes. You can use Opcache to store opcodes and optimize performance.

3.3 Assemblers

NASM (Netwide Assembler): Input: x86/x86-64 assembly code (`.asm`), Intel syntax. [9]

- **Lexical rules:** mnemonics (`mov`, `add`), registers (`eax`, `rbx`), directives (`section`, `global`), labels (`label:`), operands (immediate, memory [`var`]), comments (`;`). [9]
- **Syntactic rules:** sections (`.data`, `.bss`, `.text`). Instructions: mnemonic destination, source. Use of labels as operands. [9]
- **Semantic rules:** Checks for valid operands, references and mode of operation (16/32/64 bits). [9]

NASM directly generates object code (binary, COFF, ELF, etc.) or link-ready machine code.

GAS (GNU Assembler): Input: Assembly code (`.s`, `.S`), AT&T syntax by default. [5]

- **Lexical rules:** Suffixed mnemonics (`movl`, `addq`), registers (`%eax`), directives (`.data`, `.text`), operands (`$immediate`), comments (`#`, `/* */`). [5]
- **Syntactic rules:** AT&T syntax: source, destination. Use of dot directives, offset addressing (base, index, scale). [5]
- **Semantic rules:** Validates operands, architecturally correct instructions and symbols. [5]

GAS directly generates object code (usually ELF on Unix-like) or machine code. It is used by GCC as a backend assembler.

4 Results

4.1 Compilers

Compilation is the process of converting human-readable code (like Java, C, or Pascal) into machine language, which computers can understand. Since humans can't easily work with machine code, we use high-level languages and a compiler to translate them. Normally, the result is an executable that only works on a specific operating system. [14]

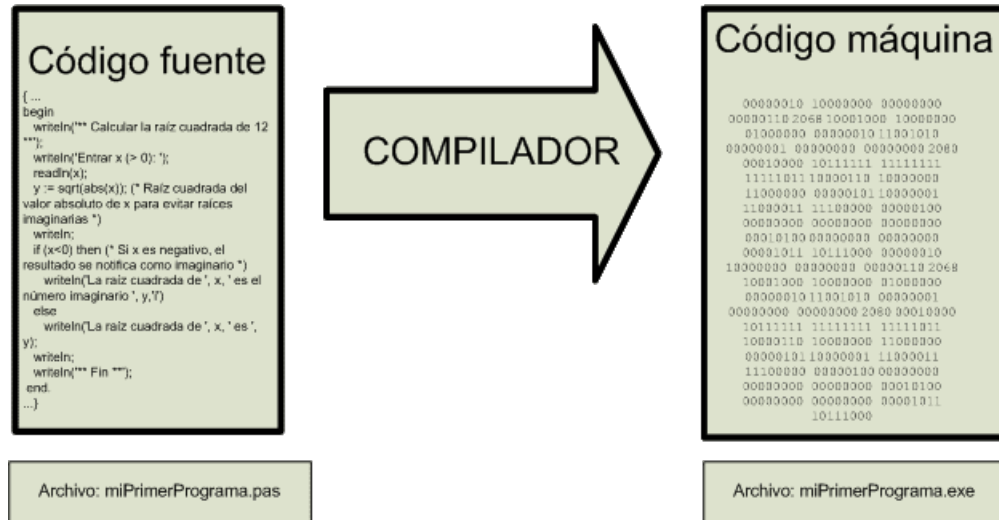


Figure 1: Compiler Example. [14]

Java introduced a key innovation: platform independence. Instead of running directly on the hardware, Java programs are compiled into an intermediate “bytecode” that runs on the Java Virtual Machine (JVM). The JVM simulates its own system, allowing the same program to run on Windows, Linux, Unix, or other platforms, as long as Java is installed. [14]

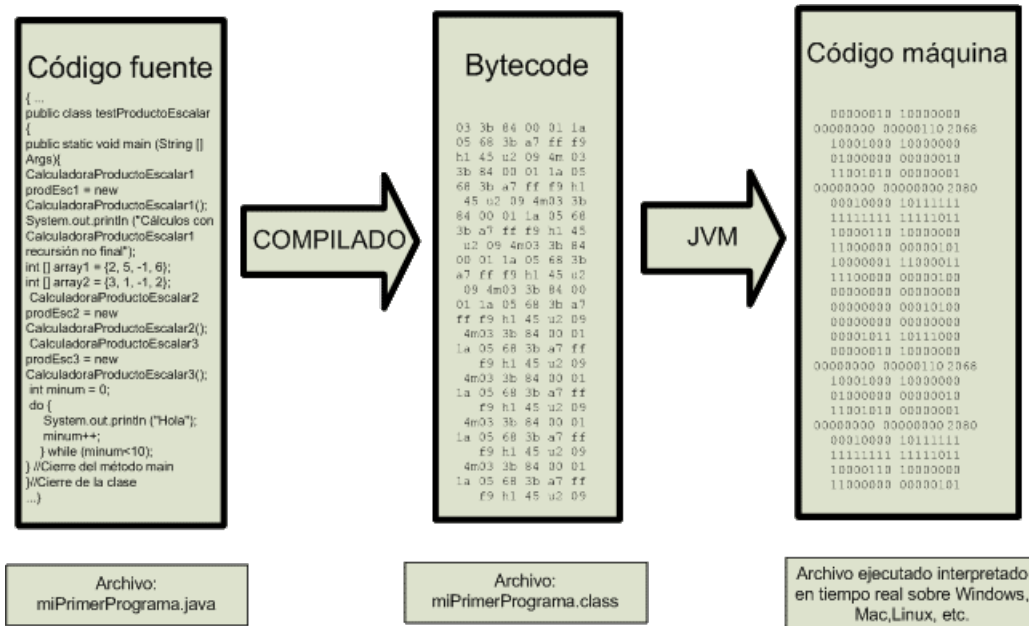


Figure 2: Compiler Example with Bytecode. [14]

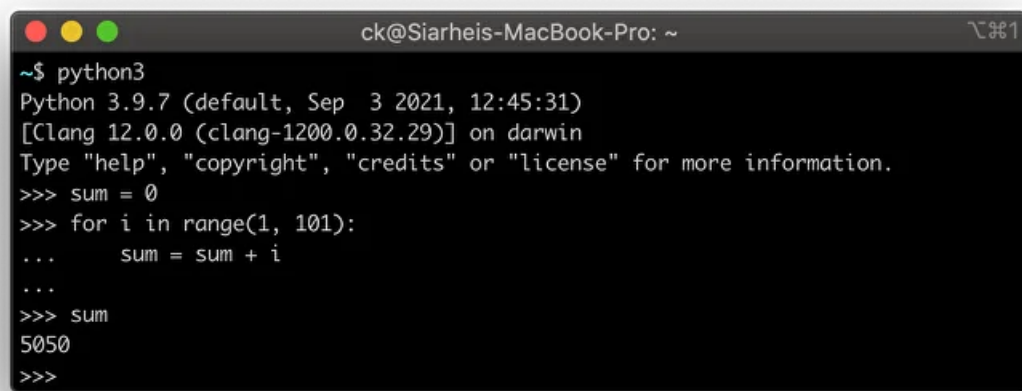
4.2 Interpreters

To better understand how an interpreter works, we will use Python as an example. The following short program calculates the sum of the numbers from 1 to 100:

```
sum = 0
for i in range(1, 101):
    sum = sum + i
print(sum) # => 5050
```

Figure 3: Python Script - Example. [2]

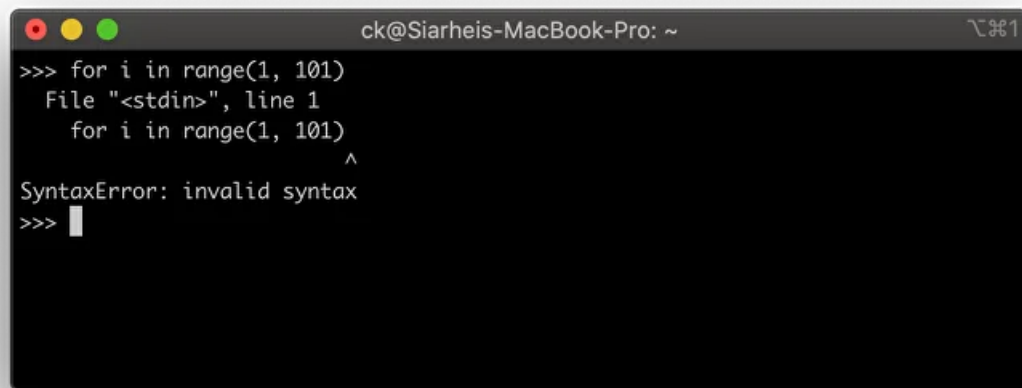
Instead of running the script from a file, we will type it directly into the Python interpreter, line by line. [2]

A screenshot of a terminal window on a MacBook Pro. The window title is "ck@Siarheis-MacBook-Pro: ~". The terminal shows the command "\$ python3" being executed, which starts the Python 3.9.7 interpreter. The interpreter displays its version and the time. Then, the user enters a multi-line Python script: "sum = 0", "for i in range(1, 101):", " sum = sum + i", and "sum". The interpreter shows the prompt ">>>" for each line. For the for loop, the prompt changes to "..." on subsequent lines. After the loop, the user enters "sum", and the interpreter displays the result "5050". The prompt returns to ">>>" at the end.

```
ck@Siarheis-MacBook-Pro: ~
~$ python3
Python 3.9.7 (default, Sep  3 2021, 12:45:31)
[Clang 12.0.0 (clang-1200.0.32.29)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> sum = 0
>>> for i in range(1, 101):
...     sum = sum + i
...
>>> sum
5050
>>>
```

Figure 4: Executing Python code interactively in the interpreter. [2]

When a command is entered, the interpreter displays the prompt `>>>`. If the command spans multiple lines (such as a `for` loop), the prompt changes to `...` until the block is finished. After completing the loop and printing the result, the interpreter shows the value 5050. [2]

A terminal window titled 'ck@Siarheis-MacBook-Pro: ~' with a dark background. It shows a Python script being executed in interactive mode. The script has a 'for' loop with a missing colon at the end. The interpreter reports a 'SyntaxError: invalid syntax' with a caret pointing to the missing colon.

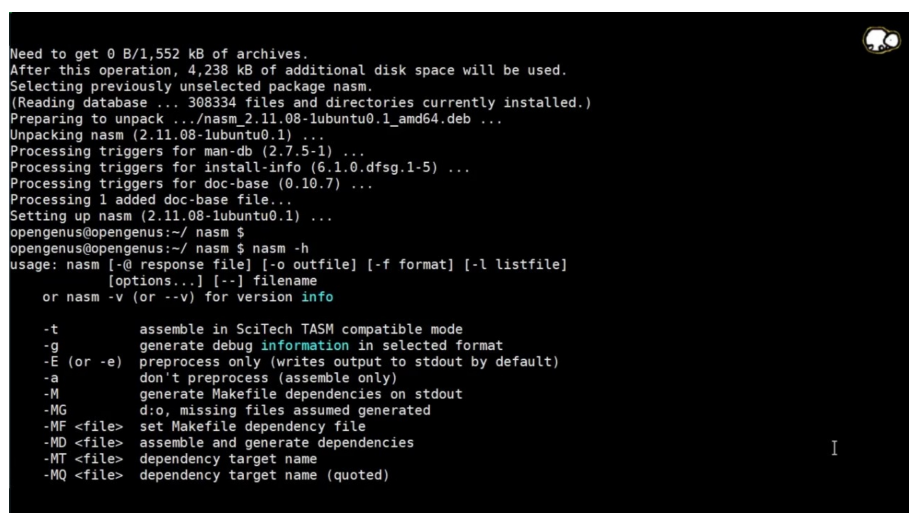
```
>>> for i in range(1, 101)
      File "<stdin>", line 1
        for i in range(1, 101)
                                ^
SyntaxError: invalid syntax
>>> █
```

Figure 5: Interpreter reporting a syntax error after a missing colon. [2]

In this case, we intentionally left out the colon at the end of the `for` statement. The interpreter immediately reports a syntax error, clearly indicating the line where the problem occurs. [2]

This direct feedback is one of the reasons why Python is considered beginner-friendly: users can experiment with code, make mistakes, and quickly learn the correct syntax. Even experienced developers find the interactive mode useful for testing small pieces of code before including them in larger projects. [2]

4.3 Assemblers

A terminal window showing the installation of NASM on Ubuntu. It starts with a message about disk space requirements, followed by the selection and unpacking of the package. Then, it shows the command to install NASM and its help output.

```
Need to get 0 B/1,552 kB of archives.
After this operation, 4,238 kB of additional disk space will be used.
Selecting previously unselected package nasm.
(Reading database ... 308334 files and directories currently installed.)
Preparing to unpack .../nasm_2.11.08-1ubuntu0.1_amd64.deb ...
Unpacking nasm (2.11.08-1ubuntu0.1) ...
Processing triggers for man-db (2.7.5-1) ...
Processing triggers for install-info (6.1.0.dfsg.1-5) ...
Processing triggers for doc-base (0.10.7) ...
Processing 1 added doc-base file...
Setting up nasm (2.11.08-1ubuntu0.1) ...
opengenus@opengenus:~/ nasm $
opengenus@opengenus:~/ nasm $ nasm -h
usage: nasm [-@ response file] [-o outfile] [-f format] [-l listfile]
           [options...] [--] filename
or nasm -v (or --v) for version info

-t         assemble in SciTech TASM compatible mode
-g         generate debug information in selected format
-E (or -e) preprocess only (writes output to stdout by default)
-a         don't preprocess (assemble only)
-M         generate Makefile dependencies on stdout
-MG        d:o, missing files assumed generated
-MF <file> set Makefile dependency file
-MD <file> assemble and generate dependencies
-MT <file> dependency target name
-MQ <file> dependency target name (quoted)
```

Figure 6: Install NASM (Netwide Assembler) on Ubuntu. [10]

The image shows a Linux terminal where an ELF 64-bit executable (‘a.out’) is analyzed.

```
$ file ./a.out
./a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, stripped
$ ls -lah ./a.out
-rwxr-xr-x 1 ii64 ii64 4.3K Nov 12 11:17 ./a.out
$ objdump -D ./a.out

./a.out:      file format elf64-x86-64

Disassembly of section .text:

0000000000401000 <.text>:
401000: 48 c7 c0 01 00 00 00    mov     $0x1,%rax
401007: 48 c7 c7 01 00 00 00    mov     $0x1,%rdi
40100e: 48 c7 c6 2a 10 40 00    mov     $0x40102a,%rsi
401015: 48 c7 c2 0d 00 00 00    mov     $0xd,%rdx
40101c: 0f 05                  syscall
40101e: 48 c7 c0 3c 00 00 00    mov     $0x3c,%rax
401025: 48 31 ff              xor     %rdi,%rdi
401028: 0f 05                  syscall
40102a: 48                    rex.W
40102b: 65 6c                gs insb (%dx),%es:(%rdi)
40102d: 6c                    insb   (%dx),%es:(%rdi)
40102e: 6f                    outsl  %ds:(%rsi),(%dx)
40102f: 2c 20                sub     $0x20,%al
401031: 77 6f                ja     0x4010a2
401033: 72 6c                jb     0x4010a1
401035: 64                    fs
401036: 0a                    .byte 0xa
```

Figure 7: Hello world in GNU Assembler (GAS). [4]

5 Conclusions

The analysis made it possible to clearly contrast the differences and similarities between compilers, interpreters and assemblers, highlighting how each plays a specific role within the program translation and execution cycle. The application of theoretical concepts discussed in class, such as grammars, intermediate representations, object files, and translation rules, was essential to give meaning and support to the results obtained.

First, it was evident that **compilers** perform a complex multi-phase translation process, which allows for code optimization and the generation of portable executables or intermediate code (as in the case of Java with its *bytecode*). In contrast, **interpreters** prioritize direct execution, which facilitates flexibility and immediate interaction, although at a cost to performance. Finally, **assemblers** demonstrated their relevance at the level closest to the hardware, being essential for the detailed control of the machine’s resources. [1]

The most relevant conclusions highlight that understanding language translators enables the establishment of relationships between source code, intermediate representations, and the final executable. Each translation tool addresses specific needs: compilers focus on optimization and portability, interpreters emphasize flexibility and simplicity, and assemblers provide direct control over the machine. Furthermore, the applied theoretical framework serves as the foundation for distinguishing the roles of

each translator and recognizing their significance in software construction and execution.

References

- [1] A. Casero. “Interpreters in programming”. In: *KeepCoding Bootcamps* (Sept. 2024). [Online]. URL: <https://keepcoding.io/blog/que-son-los-interpretes-en-programacion/>.
- [2] Codica. “What is an interpreter?” In: *Codica* (2025). [Online]. URL: <https://codica.la/guias/interpreter>.
- [3] EcuRed. “Interpreter (Computer Science)”. In: *EcuRed* (2025). [Online]. URL: [https://www.ecured.cu/Int%C3%A9rprete_\(Inform%C3%A1tica\)](https://www.ecured.cu/Int%C3%A9rprete_(Inform%C3%A1tica)).
- [4] Gist. “Hello world in GNU Assembler (GAS)”. In: *Gist* (2025). [Online]. URL: <https://gist.github.com/carlosarcamo/6833d19b726af698e62b>.
- [5] GNU. “User guide to the GNU assembler”. In: *GNU* (2025). [Online]. URL: <https://sourceware.org/binutils/docs/as/>.
- [6] Kexugit. “Compilers: What every programmer should know about compiler optimizations”. In: *Microsoft* (2025). [Online]. URL: <https://learn.microsoft.com/es-es/archive/msdn-magazine/2015/february/compilers-what-every-programmer-should-know-about-compiler-optimizations>.
- [7] Lenovo. “Assembler: Everything You Need to Know and How to Use It”. In: *Lenovo* (2025). [Online]. URL: <https://www.lenovo.com/mx/es/glosario/ensamblador/>.
- [8] R. Maldonado. “Intermediate language compilers: what they are and their function”. In: *KeepCoding Bootcamps* (Aug. 2024). [Online]. URL: <https://keepcoding.io/blog/que-son-compiladores-de-lenguaje-intermedio/>.
- [9] NASM. “NASM - The Netwide Assembler”. In: *NASM* (2025). [Online]. URL: <https://nasm.us/doc/>.
- [10] OpenGenus. “Install NASM (Netwide Assembler) on Ubuntu”. In: *OpenGenus* (2019). [Online]. URL: <https://www.youtube.com/watch?v=sYiU3d53p0w>.
- [11] Oracle. “The Java Language Specification”. In: *Oracle* (2015). [Online]. URL: <https://docs.oracle.com/javase/specs/jls/se8/html/index.html>.
- [12] PHP. “PHP Manual”. In: *PHP* (2025). [Online]. URL: <https://www.php.net/manual/en/index.php>.
- [13] Python. “Python Documentation contents”. In: *Python* (2025). [Online]. URL: <https://docs.python.org/3/contents.html>.

- [14] A. R. Y. W. Sagástegui. “The Java Virtual Machine”. In: *aprenderaprogramar* (2025). [Online]. URL: https://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=392:la-maquina-virtual-java-jvm-o-java-virtual-machine-compilador-e-interprete-bytecode-cu00611b&catid=68&Itemid=188.
- [15] J. Schneider and I. Smalley. “What is a compiler?” In: *IBM* (July 2025). [Online]. URL: <https://www.ibm.com/think/topics/compiler>.
- [16] Wikipedia. “Programming Language”. In: *Wikipedia* (July 2025). [Online]. URL: https://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n.