Universidad Nacional Autónoma de México

Ingeniería en Computacion

Nombre de Asignatura: Compiladores

PRACTICE 1

ALUMNOS:
320195019
320042645
320332825
320318931
320095531

Grupo:
5
Semestre:
2026-I

México,CDMX.21 Agosto 2025

Índice

1. Introducción

In the field of programming, one of the main challenges is to ensure that the code written by developers is understood and executed by the computer. This process is not straightforward, as human and machine languages are very different. To solve this non-trivial problem, compilers, interpreters, and assemblers are used. These are essential tools that enable the transformation of a language understandable to humans into a set of specific commands that computer hardware can assimilate and execute directly.

However, for a student or professional in training, clearly distinguishing between these three types of translators, how they work internally, what input data they process, what grammatical structures they use, and what intermediate code they generate is often an unfamiliar topic. Therefore, it is essential to study them, as this facilitates the creation of efficient and reliable software and can provide programmers with a better understanding of the inner workings of programming languages and their execution on the machine, as well as the knowledge necessary to create new programming languages.

The main objective of this research work is to analyze and compare the three main code translators: compilers, interpreters, and assemblers. This analysis will be carried out by breaking down each of them into three aspects:

- **Entry Requirements:** Determine what type of source code or instructions each translator requires to begin the process (e.g., source code in a high-level language, assembly code, scripts).
- **Main elements of grammar:** Identify and describe the key grammatical components (tokens, syntax, structures) that every translator must recognize and process correctly.
- **Generated intermediate code:** Investigate and explain the nature of the intermediate code of your translation process (e.g., object code, three-address code, bytecode, or direct execution).

The final objective is for this research to serve as a guide for comparing and understanding the unique role and characteristics of each tool.

2. Marco Teórico

A compiler is a program that translates source code written in a high-level language into machine code that can be executed directly by the computer, which makes it fundamental for the development of efficient and complex software.
In contrast, an assembler converts assembly language, a low-level and hardware-oriented language, into machine instructions; although more difficult to use, it provides precise control of system resources and is frequently applied in embedded systems or performance-critical applications.

An interpreter, however, executes the program directly from the source code line by line, without generating an independent executable. This approach generally results in slower execution, yet it offers important advantages in terms of testing, debugging, and portability. Therefore, compilers are most suitable for performance and large-scale projects, assemblers for low-level optimization and hardware interaction, and interpreters for rapid development and flexibility.[13]

3. Desarrollo

3.1. Compilers

- **javac**
  Javac is the standard compiler in Java, it is part of the Java Development Kit (JDK) and is used to compile source code written in Java into class files that can be executed by the Java Virtual Machine (JVM). [20]

  One of the main characteristics of javac is its ability to detect syntax and semantic errors in the source code during the compilation process. This helps developers identify and fix possible problems in their programs before executing them, saving time and reducing the probability of runtime errors.

  What input does it require?
  The javac command reads source files that contain module, package and type declarations written in the java programming language, and compiles them into class files that run on the JVM. This command can also process annotations in Java source files and classes. Source files must have a file name extension of .java. [21]

  The main elements of its grammar
  Javac compiler uses the grammar defined in the Java Language Specification (JLS).

  The lexical grammar of Java has as its terminal symbols the characters of the Unicode character set. It defines a set of productions that describe how sequences of Unicode characters are translated into a sequence of input elements. These input elements, with white space and comments discarded, form the terminal symbols for the syntactic grammar for Java and are called tokens. These tokens are the identifiers, keywords, literals, separators and operators of the Java programming language. [22]

  The syntactic grammar of Java has tokens defined by the lexical grammar as its terminal symbols. It defines a set of productions that describe how sequences of tokens can form syntactically correct programs.

  - **Identifiers**: unlimited length sequence of Java letters (uppercase and lowercase ASCII Latin letters, the ASCCI underscore and dollar sign) and Java digits (ASCII digits 0-9), the first of which must be a Java letter.

- **Keywords**: 50 character sequences, formed from ASCII letters are reserved for use as keywords (class, extends, import, break, etc.).
- **Literals**: representation of a value of a primitive type, the string type or the null type.
- **Separators**: () [] ; , .
- **Operators**: =, <, >, !, *,
- **White space**: ASCII space character, horizontal tab character, form feed character and line terminator characters.
- **Comments**: /*text*/, //text

The intermediate code it generates
The javac command compiles the source codes into bytecode class files. Bytecode is an intermediate representation designed to run on the JVM.These file names must have .class suffixes.

- **GCC**
  GCC is the GNU Project's integrated compiler for C, C++, Objective C, and Fortran; it can accept a source program in any of these languages and generate a binary executable program in the language of the machine on which it will run. The acronym GCC stands for "GNU Compiler Collection.Ït originally stood for "GNU C Compiler"; GCC is still used to designate a C compilation. G++ refers to a C compilation.[15]

  What input does it require?
  The GCC compiler requires as input at least one source code file written in C, C++, or another language supported by GCC, such as: gcc myprogram.c or g++ myprogram.cpp to compile source code files.

  The main elements of its grammar

  - **Statements:** All C statements except for loops and some control flow statements. [16]
  - **Operators:** All C unary, binary, and ternary operators except the comma operator. Operator precedence and associativity follow C language rules. [16]
  - **Data Types:** Most legally defined C-89 variable types, including all statements and keywords (`struct`, `union`, `enum`, `typedef`, etc.) for declaring types subject to constraints. This includes the types of kernel or application variables and parameters. [16]
  - **Type Conversions:** Implicit type conversions, as well as explicit conversions with type casts. [16]
  - **Subroutine:** The syntax for calling subroutines and passing parameters to functions. However, there are restrictions on which functions can be called. [16]

- **Variable Names:** Naming conventions for variables follow the C identifier rules. The full variable specification may include a colon if a variable class name is prefixed with the variable name. [16]
- **Header Files:** Header files can be included to explicitly declare the types of kernel global variables or function prototypes in applications and the kernel. There are some restrictions on how header files are included. [16]
- **Punctuators:** All C punctuators are supported, and their rules apply. Therefore, statements must be separated by the semicolon character (;). All C whitespace rules are followed. [16]
- **Literals:** String representations (using double quote characters `"`), character literals (using single quote characters `'`), octal and hexadecimal integers, and special characters such as the escape sequences `\n` and `\t`. [16]
- **Comments:** C-style and C++-style comments. Comments can appear either inside or outside a clause. Any line beginning with a `#` character will be ignored. Avoid using the character to indicate a comment line. [16]

The intermediate code it generates
the GNU Compiler Collection, generates several intermediate representations (IRs) during the compilation process before producing the final machine code. The main ones are:

- **Generic**: The code is transformed into an intermediate representation called GENERIC. The purpose is simply to provide a language-independent way of representing an entire function in trees.[17]

- **Gimple**: GIMPLE is a three-address representation derived from GENERIC by breaking down GENERIC expressions into tuples of no more than 3 operands (with some exceptions like function calls). GIMPLE was heavily influenced by the SIMPLE IL used by the McCAT compiler project at McGill University, though we have made some different choices. For one thing, SIMPLE doesn't support goto.

  Temporaries are introduced to hold intermediate values needed to compute complex expressions. Additionally, all the control structures used in GENERIC are lowered into conditional jumps, lexical scopes are removed and exception regions are converted into an on the side exception region tree.

  The compiler pass which converts GENERIC into GIMPLE is referred to as the 'gimplifier'. The gimplifier works recursively, generating GIMPLE tuples out of the original GENERIC expressions.[18]

- **RTL**: The last part of the compiler work is done on a low-level intermediate representation called Register Transfer Language. In this language, the

instructions to be output are described, pretty much one by one, in an algebraic form that describes what the instruction does.

RTL is inspired by Lisp lists. It has both an internal form, made up of structures that point at other structures, and a textual form that is used in the machine description and in printed debugging dumps. The textual form uses nested parentheses to indicate the pointers in the internal form.[19]

3.2. Interpreters

Python interpreter (CDPython)

What input does it require?

Receive Python source code, which is human-readable and written in files with the .py extension. [8]

The main elements of its grammar.

Python grammar is based on a notation that combines elements of Extended Backus-Naur Form (EBNF) and Parsing Expression Grammars (PEG). This notation details how different grammatical elements can be combined to form valid Python programs.[9]

- **Identation**: Python uses indentation to define blocks of code, unlike many other languages that use curly brackets or keywords. This is a fundamental part of its syntax.[9]
- **Statements**: The Python interpreter can execute various instructions, including assignment statements, control flow statements (e.g., if/elif/else, for, while), function definitions (def), class definitions (class), and import statements.[9]
- **Operators**: It is possible to perform operations on certain values and variables using certain symbols or keywords, such as arithmetic operators (+, -, *, /), comparison operators (==, !=, <, >), and logical operators (and, or, not).[9]
- **Keywords**: Python has a set of reserved words (e.g., if, else, while, for, def, class, import, True, False, None) that have special meanings and cannot be used as names for variables, classes, etc.[9]

Intermediate code

The Python interpreter (CPython) does not compile code directly into machine language, or in general terms, it does not generate intermediate code. Instead, it translates the source code into Python bytecode, which, using the Python virtual machine (PVM), the interpreter loads the machine language into the PVM, which converts the code into zeros and ones, thus printing the results. [9]

PHP interpreter

What input does it require?

The input must be PHP source code, a file with the extension .php.[10]

The main elements of its grammar.

- **Statements**: PHP scripts are built from a series of statements, which can be assignments, function calls, control structures (such as loops and conditionals), or even empty statements. Statements usually end with a semicolon (;). Multiple statements can be grouped together between curly braces to form a block.[11]
- **Variables**: Variables in PHP are declared with the dollar sign prefix `($)` and are case sensitive.[11]
- **Functions and Classes**: Its grammar includes reserved words for defining functions (function) and classes (class) for object-oriented programming. This has a specific syntax for declarations, parameters, return types, and inheritance. For example, to define a function, you must use the reserved word "function" followed by the name, with a list of arguments separated by commas inside parentheses, and then place the function definition between curly brackets.[11]
- **Keywords**: PHP uses reserved words such as if, else, while, for, function, class, extends, and echo, which are not case sensitive.[11]
- **Data types**: PHP supports several built-in data types, including null, bool, int, float, string, array, object, callable, and resource.[11]

Intermediate code

PHP does not generate intermediate code as such, but rather compiles the AST (Abstract Syntax Tree) by recursively traversing it, translating it with some optimizations, and generating OP Code. This is taken by the Zend virtual machine (Zend Engine VM) to be executed and build source code ready for use on a website. [11]

Ruby interpreter

What input does it require?

The input must be Ruby source code, a file with the extension .rb.[6]

The main elements of its grammar.

- **Variables**: It is not necessary to declare the type of a variable. The type is inferred from the value assigned to it. There are different types of variables. Local variables begin with a lowercase letter or an underscore (my_variable). Instance variables begin with @ (name). They are used for variables that belong to a specific instance of a class. Class variables begin with @@ (for example,

@@counter). They are shared among all instances of a class. Finally, global variables begin with $ ($path).[7]

- **Functions**: Methods are defined with the reserved word def and end with end. Parentheses may or may not be used when calling a function.[7]
- **Flow control structures**: There are conditionals: if, elsif, else, unless. And loops: while, until, for, and object iterators (each, map, select).[7]
- **Keywords**: Ruby uses reserved words such as class, def, end, if, else, elsif, while, for, nil, true, false, return, super, and module.[7]
- **Objects and Classes**: Everything in Ruby is an object. Classes are "templates" that define the structure and behavior of objects. An instance is an object created from a class. The initialize method is the constructor of a class.[7]

Intermediate code

As such, Ruby does not generate intermediate code because it is an interpreter. However, the execution process of this language is described below: First, the source code instructions are transformed into a set of instructions, or bytecode, by YARV, the Ruby interpreter. These intermediate instructions are used to execute the program, which is provided by its own virtual machine.[7]

3.3. Assemblers

NASM (Netwide Assembler)

What input does it require?

Assembly language source files (`.asm`).[1]

The main elements of its grammar.

- **Syntax**: Minimal and unambiguous, uses Intel syntax.
- **Instruction format**: `mnemonic operands`, e.g.: MOV EAX, 1
- **Sections**: Explicit separation: section .text section .data section .bss
- **Registers**: Written plainly (EAX, RAX), supports 16, 32, and 64-bit registers.
- **Constants and numbers**: e.g. `0xFF`, `1010b`, `377o`, `255d`.
- **Labels**: Must end with a colon, e.g. `start:`. Local labels with dot prefix, e.g. `.loop`.
- **Directives**: Start with %, e.g. `%define`, `%macro`, `%include`.
- **Macros**: Powerful, with parameters, conditionals and loops at assembly time.

- **Strictness**: No high-level constructs like `.IF`; must use explicit instructions (`CMP`, `JNE`, etc.).

[1]

Intermediate code

- Flat binary files (`.bin`, e.g. bootloaders).
- Object files: ELF (Linux), COFF (Windows), Mach-O (macOS).[1]

FASM (Flat Assembler)

What input does it require?

Assembly source files (`.asm`). Often used in OS development, demos and optimized programs.[2]

The main elements of its grammar.

- **Syntax**: Intel style, simple and self-consistent.
- **Instructions**: `mnemonic destination, source`, e.g.: MOV EAX, 1234h
- **Sections**: Supports `.text`, `.data`, but also allows flat binaries without sections.
- **Directives**: Advanced: `define`, `include`, `virtual`, `macro`.
- **Labels**: End with colon; local labels with dot prefix.
- **Special grammar**: Strict and recursive; FASM is self-hosting (written in itself).[2]

Intermediate code

- Flat binaries (`.bin`), executable directly.
- Object files (`.obj`, ELF, etc.) for linking.[2]

GNU assembler

What input does it require?

Source files passed as arguments in the command line, concatenated into a single execution.[3]

The main elements of its grammar.

- **Program structure**: program ::= statement statement ::= [label] (directive | instruction) [comment]
- **Symbols**: Case sensitive, cannot start with a digit.
- **Directives**: `.identifier [args]`, e.g. `.text`, `.data`.
- **Instructions**: `mnemonic [operands]`, e.g.: mov
- **Operands**: immediate, registers, addressing, constants.
- **Constants**: binary (`0b1010`), octal, decimal, hexadecimal (`0xFF`), floating point.
- **Comments**: `/* ... */`, `#`, or `//`.
- **Sections**: `.text`, `.data`, `.bss`.[3]

Intermediate code

- Object files (`.o`) containing assembly translation, linking information and optional debug symbols.[3]

MASM (Microsoft Macro Assembler)

What input does it require?

Assembly source code following MASM grammar and syntax. [4]

The main elements of its grammar.

- **Structure**: Organized into modules with directives, segments, procedures, macros and data.
- **Directives**: Segments, macros, structures, procedures, data declarations, types, flow control.
- **Procedures**: With parameters and register usage.
- **Macros**: With parameters.
- **Types**: Declared using `TYPEDEF`.
- **Control flow**: IF, ELSEIF, ELSE, WHILE, REPEAT.
- **Expressions**: Arithmetic and logical.
- **Registers**: All available registers grouped.
- **Instruction format**: [prefix] mnemonic operand-list
- **Prefixes**: e.g. `REP`, `LOCK`, `VEX`.[5]

Intermediate code

- Object code with machine instructions to be linked.[5]

4. Resultados

4.1. Compilers

```
D:\FI\FI\3° semestre\POO\avanceP1POO>javac HolaMundo.java

D:\FI\FI\3° semestre\POO\avanceP1POO>
```

```
20/08/2025  02:39 p. m.               422 HolaMundo.class
20/08/2025  02:37 p. m.               574 HolaMundo.java
```

The file named "HolaMundo.java" was compiled with the javac command into a class file named "HolaMundo.class".

```c
#include <stdio.h>

int main() {
    printf("Hola, mundo!\n");
    return 0;
}
```

Code in C

```
gcc ejemplo.c -o ejemplo
./ejemplo
```

Example of compilation with gcc

4.2. Interpreters

To test the **Python interpreter**, a file called holaM.py was used, contained in a folder called pruebaCompiladores.

```
    Directorio: C:\Users\paulm\OneDrive\Escritorio\pruebaCompiladores


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        21/08/2025  06:27 p. m.             14 holaM.py
```

To run the file, the command "py holaM.py" is used, obtaining the output Hello. Unlike compilers, Python does not generate an executable file; instead, the interpreter translates the code into bytecode and executes it in the Python virtual machine.

```
PS C:\Users\paulm\OneDrive\Escritorio\pruebaCompiladores> py holaM.py
Hello
PS C:\Users\paulm\OneDrive\Escritorio\pruebaCompiladores>
```

To test the **PHP interpreter**, a file called prueba.php was used, also contained in the pruebaCompiladores folder.

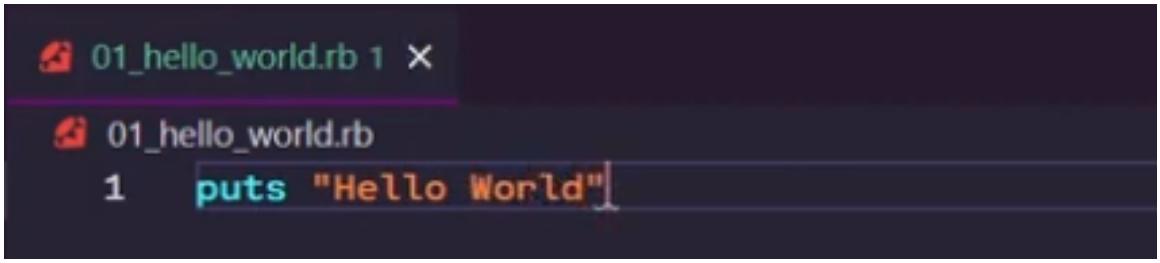
```
    Directorio: C:\Users\paulm\OneDrive\Escritorio\pruebaCompiladores


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        21/08/2025  06:27 p. m.             14 holaM.py
-a----        21/08/2025  07:11 p. m.             27 prueba.php
```

This file can be executed by typing the command "php prueba.php", which outputs "Hello PHP". With this, the interpreter processes the source code and executes it directly. PHP does not generate an executable file, but interprets the code at runtime.

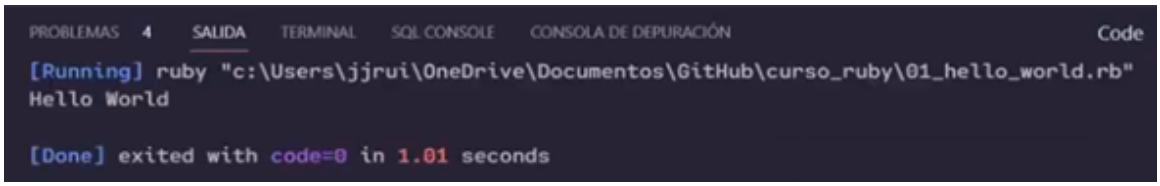
```
C:\Users\paulm>cd C:\Users\paulm\OneDrive\Escritorio\pruebaCompiladores

C:\Users\paulm\OneDrive\Escritorio\pruebaCompiladores>php prueba.php
Hello PHP
```

To test the **Ruby interpreter**, a file with the extension .rb was used, which prints a string to the console.

The execution is performed with the instruction "ruby file.rb", obtaining the output Hello World. Ruby, like Python and PHP, does not generate a binary executable file; instead, the interpreter translates the code into bytecode and executes it on its virtual machine (YARV).

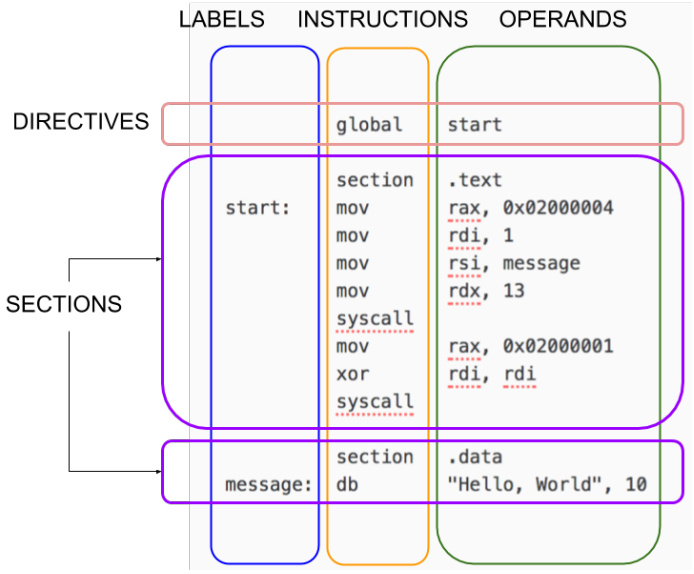


Instruction used to write the string "Hello World" in the console. Screenshot from the OpenBootcamp video [14].



Execution of the .rb file with the output of the written instruction. Screenshot from the OpenBootcamp video [14].

4.3. Assembler



General structure of NASM Assembler. The rest of the assemblers have a very similar structure. [12]

5. Conclusiones

The study of the different options for translating instructions is fundamental to understanding how programming languages can be transformed into machine instructions. Each tool has its own advantages, and its use depends on the specific needs of

the project. Compilers are the best choice when efficient, independent executables are required, making them suitable for large-scale and high-performance applications. Assemblers, on the other hand, provide detailed control over system resources. Interpreters, by directly executing the source code, offer flexibility and speed in development, making them particularly useful for testing and debugging.

Overall, these tools represent a balance between performance, control, and ease of development, which are key aspects in the creation of reliable and efficient software.

6. Referencias

[1] Team, N. A. (s. f.). NASM. https://www.nasm.us/

[2] flat assembler. [Online]. Available: https://flatassembler.net/

[3] D. Elsner and J. Fenlason, Using as the GNU Assembler, ver. 2.14. Cygnus, 1994. [Online]. Available: https://ee209-2019-spring.github.io/references/gnu-assembler.pdf

[4] TylerMSFT, "Formato de instrucciones MASM,"Microsoft Learn. [Online]. Available: https://learn.microsoft.com/es-es/cpp/assembler/masm/instruction-format?view=msvc-170

[5] TylerMSFT, "Gramática de BNF de Microsoft Macro Assembler,"Microsoft Learn. [Online]. Available: https://learn.microsoft.com/es-es/cpp/assembler/masm/masm-bnf-grammar?view=msvc170

[6] T. de Bruijn. (Aug, 2017). A look at how Ruby interprets your code. [Online]. Available: https://blog.appsignal.com/2017/08/01/ruby-magic-code-interpretation.html

[7] Ruby. (s.f.). Rdoc Documentation. [Online]. Available: https://docs.ruby-lang.org/en/3.0/lib/racc/rdoc/grammar_en_rdoc.html

[8] Geeksforgeeks. (2025, July 27). What is Python Interpreter. [Online]. Available: https://www.geeksforgeeks.org/python/what-is-python-interpreter/

[9] Python. (s.f.). Full Grammar specification. [Online]. Available: https://docs.python.org/3/reference/grammar.html

[10] M. Andrzejewski. (2020, December 29). How a PHP interpreter works. [Online]. Available: https://www.droptica.com/blog/how-php-interpreter-works/

[11] Php. (s.f.). Manual of Php. [Online]. Available: https://www.php.net/manual/es/langref.php

[12] NASM Tutorial. (s. f.). [Online]. Available en: https://cs.lmu.edu/ ray/notes/nasmtutorial/

[13] D. Watt, Brown, Programming Language Processors in Java: Compilers and Interpreters, Prentice hall, 2000

[14] OpenBootcamp. (2023, August 25). 2 Instalación - Curso Ruby - OpenBootcamp. [Online video]. Available: https://youtu.be/uL3QRuIvR-o?si=EYzDTCt1greacfGR

[15] Universidad de la República, Facultad de Ingeniería, .El compilador GCC,.Estructuras de Datos y Algoritmos – Tecnólogo en Informática, Montevideo, Uruguay, 8 p. [Online]. Available: https://www.fing.edu.uy/tecnoinf/mvd/cursos/eda/material/otros/compilador$_GCC.pdf$

[16] IBM, .Elementos de C", IBM AIX 7.3.0 Documentation, sección .Elementos de C", ProbeVue, [Online]. Available: https://www.ibm.com/docs/es/aix/7.3.0?topic=concepts-elements-c.

[17] W. Hsiangkai, "GCC GENERIC,"SlideShare, Ago. 2016. [Online]. Available: https://es.slideshare.net/slideshow/gcc-generic/651166502.

[18] Free Software Foundation, Inc., "GIMPLE,"GNU Compiler Collection (GCC) Internals, [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html

[19] Free Software Foundation, Inc., RTL Representation,"GNU Compiler Collection (GCC) Internals, [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/RTL.html

[20] ProgramaciónPro, "¿Qué es el compilador javac y cómo funciona?", ProgramaciónPro, 26-mar-2024. [Online]. Available: https://programacionpro.com/que-es-el-compilador-javac-y-como-funciona/.

[21] Oracle, "Javac – the Compiler", The Core JDK Tools, sitio web Dev.java, 14-sep-2021. [Online]. Available: https://dev.java/learn/jvm/tools/core/javac/.

[22] J. J. Gosling, B. Joy, G. L. Steele Jr., G. Bracha y A. Buckley, "Chapter 2: Grammars," en The Java® Language Specification, Java SE 7 Edition, Oracle, Feb. 28, 2013. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se7/html/jls-2.html.