Universidad Nacional Autónoma de México

Faculad de Ingeniería

Computer Engineering

Compilers

Compiler

Team: 04

Barco Núñez Claudia Citlali 422067621

Caballero Antunez Jesús Yael 320231364

Jacinto Robledo Valeria Berenice 320057973

Ospino Mérida Emilio Sebastián 319573471

Group: 05

Semester 2025-2

México, CDMX. June 2025

Contents

1 Introduction

This project was developed as part of the Compilers course at the School of Engineering, National Autonomous University of Mexico. Its primary goal was to design and implement a fully functional compiler, built from scratch, that integrates all fundamental compilation stages: lexical analysis, syntactic analysis, semantic analysis, and assembly code generation.

The compiler was implemented using the Python programming language, leveraging the **Lark** library to construct a LALR(1) parser. A graphical user interface was also developed using **Tkinter**, allowing users to interact with the compiler in an intuitive and practical way. The input language designed by the team is imperative, with syntax inspired by C and Go, supporting variable declarations, arithmetic and logical expressions, conditional statements, and code blocks.

As the final output, the compiler generates NASM-style assembly code targeting the x86 architecture, which can be assembled and linked using external tools such as *nasm* and *ld* to produce an executable binary. The project also includes basic semantic validations such as variable declaration checks, as well as lexical and syntactic error handling.

1.1 Goals

To develop an educational compiler capable of analyzing, validating, and translating programs written in an imperative language defined by the team, generating executable assembly code and supported by a graphical user interface.

1. Design a formal grammar that defines the syntax of the source language.
2. Implement a lexical analyzer to identify valid tokens of the language.
3. Build a syntactic analyzer that produces an Abstract Syntax Tree (AST).
4. Develop a semantic analyzer module to validate variable declarations and usage.
5. Implement an assembly code generator compatible with NASM for x86 architecture.
6. Create a graphical user interface for writing, compiling, and displaying the results of code analysis and translation.
7. Integrate lexical, syntactic, and semantic error handling to provide immediate user feedback.

2 Theoretical Framework

The development of a compiler requires a solid understanding and application of fundamental concepts from formal language theory, automata theory, and computer architecture. Each compilation stage (lexical, syntactic, semantic analysis, and code generation)

is based on theoretical principles that support its correct implementation.

One of the key pillars is the use of **context-free grammars** (CFGs), which describe the syntactic structure of a programming language. These grammars are interpreted by a **parser**, which builds hierarchical trees (ASTs) from token streams produced by the lexer, which in turn relies on regular expressions and deterministic finite automata to segment the source code into lexical units.

The **semantic analysis** layer provides logical validation, ensuring, for example, that identifiers are properly declared and used. This is supported by structures such as symbol tables and builds upon principles from type theory and scope management.

The **code generation** phase translates the abstract representations into low-level instructions. In this project, *NASM assembly code* was generated for the x86 architecture, which required applying knowledge from computer architecture, including the use of registers, memory management, and processor-level arithmetic and logical operations.

Compiler, interpreter, and Assembler

In the context of software construction, it's essential to distinguish between a compiler, an interpreter, and an assembler, as they serve different purposes and operate in distinct ways.

- A **compiler** takes a high-level language program and translates it entirely into machine code or a lower-level language (such as assembly), producing an independent executable file.
- An **interpreter** executes source code line by line, performing lexical, syntactic, and semantic analysis at runtime, without generating an independent executable.
- An **assembler** translates assembly language instructions (a human-readable low-level language) into binary machine code directly executable by the processor.

This project functions as a compiler, as it performs all analysis stages and produces a NASM-style assembly file, which can later be compiled into an executable using external tools (*nasm*, *ld*).

LALR(1) Parsers and Their use

The compiler uses a **LALR(1) parser** (Look-Ahead Left-to-Right with 1 token of lookahead), which is an efficient method for syntactic analysis of context-free grammars. This kind of parser can handle complex language structures and is widely used in real-world tools like Yacc and Bison.

By using the Lark library, which implements LALR parsing, the grammar could be defined declaratively and parsed automatically, reducing development time and minimizing parsing errors. [1]

One-Pass Compilers

A **one-pass compiler** is designed to process the source code in a single sequential pass, performing all phases of analysis and code generation without storing extensive intermediate representations. While they may have some limitations compared to multi-pass compilers, they are faster and more efficient in terms of compilation time.

The compiler developed in this project follows a similar one-pass approach: the source code is read once, then lexically and syntactically analyzed, semantically validated, and immediately translated to assembly in a single execution cycle, without requiring multiple traversals of the AST.

3 Development

Requirements

The python packages that are required for the execution of the program are the following:

- lark
- tkinter
- llvmlite

The development process followed an incremental approach, starting with the definition of the language grammar and parser construction. The semantic analyzer and assembly code generator were later integrated, followed by the creation of a graphical interface to enhance user interaction. Each stage of the compiler was designed with clear separation of concerns, enabling easier testing, maintenance, and potential future extensions.

3.1 Language Desing

The language desinged for this compiler is inspired by languages such as C, and Go. It presents a simpl syntax focused on basic control structures, declarations, expressions, and function calls.

Keys elements of the language:

- Variable declarations using the **var** keyword.
- Assigment statements using operators such as =, +=, etc.
- Arithmetic, logical, and comparison expressions.
- Conditional statements with **if** and **else**.
- Code blocks delimited by curly braces {}.
- Function calls using C-like syntax **name(args)**.

The grammar was implemented using Lark syntax, and it was designed to automatically produce Abstract Syntax Trees (AST) using transformation rules. Its syntax is simple and easy to analyze.

3.2 Lexical analizer

The lexical analyzer's main function is to read the source code and break it into lexical units (tokens) that are processed by the parser. It was implemented using regular expressions, identifying the following token categories:

- Identifiers: variable or function names or functions that follow a valid pattern.
- Numbers: integers and floating-point values, including negatives.
- Literals: strings and characters.
- Delimiters: parentheses, braces, brackets, semicolons, commas.
- Comments:single-line and multi-line.
- Operators: arithmetic (+, -, *), comparison (==, !=), assignment (=, +=, etc.), and logical (&&, ||).

Implementation: The lexical analyzer was not manually implemented; instead, Lark automatically handles tokenization by interpreting the declared regular expressions and grammar rules.

3.3 Syntactic analizer

Syntactic analysis is the second fundamental stage in the compilation process, following lexical analysis. Its main purpose is to verify that the sequence of tokens generated by the lexer conforms to the grammatical rules of the designed language. For this purpose, a **LALR(1)** parser was implemented using the Lark library, which supports declarative definition of context-free grammars. [2]

The parser receives a stream of tokens as input and produces an Abstract Syntax Tree (AST) as output, representing the hierarchical structure of the program. This structure is essential for semantic validation and later code generation. The AST is constructed through a custom transformer that maps each grammar rule to a specific node type in the tree, based on the corresponding statement or expression.

The system correctly handles variable declarations, arithmetic and logical expressions, assignments, brace-delimited blocks, conditionals with optional **else** clauses, and function calls with arguments. It also integrates error-handling mechanisms through specific exceptions, enabling accurate identification and reporting of structural issues in the source code.

This modular, tree-based approach allows for clear separation between compiler components and improves the maintainability of the project, making future grammar extensions more manageable.

3.4 Semantic analizer

Semantic analysis is a critical stage in the compilation process, as it validates the logical consistency of the source code beyond its grammatical structure. This module was implemented in a modular way, by traversing the previously generated Abstract Syntax Tree (AST).

In this project, the semantic analyzer performs three critical validations:

1. **Correct use of variables:** It verifies that all variables used in the program have been previously declared, preventing references to undefined identifiers.
2. **No duplicate declarations:** It ensures that no variable is declared more than once within the same scope, avoiding naming conflicts in the symbol table.
3. **Expression consistency:** It checks that all operations used in arithmetic, logical, or conditional expressions are defined within the grammar, and that the operands make semantic sense in that context.

Additionally, as the Abstract Syntax Tree (AST) is traversed, the analyzer annotates the data type associated with each constant, variable, or expression. While strict type checking is not enforced at this stage, this infrastructure provides a foundation for future enhancements such as a formal type system or type inference capabilities. [1]

3.5 Intermediate Code Generation / Assembly

The assembly code generator is the module responsible for translating the Abstract Syntax Tree (AST) into assembly instructions compatible with the x86 architecture using NASM syntax. This stage converts high-level constructs such as declarations, assignments, expressions, and conditionals into low-level instructions interpretable by the processor.

The system manages an intermediate code that can be optimized and compiled by the library LLVM IR [10] by obtaining the parse tree developed by the grammatic. Due the functionalities of the library, it is possible to obtain the binary code of a diversity of architectures, in our project is especially design for x86 architecture. In order to obtain the intermediate code, there is a n implementation of a variable handler than inserts the variables into the code, a block handler that executes statements within code blocks and an evaluation of comparison operations and expressions.

The final output includes a comment indicating the name of the LLVM module, the target architecture, vendor, and operating system and a description of the memory alignment and size details for the target.

3.6 User Interface

A graphical interface using **Tkinter** was implemented to allow users to write, open, and save Tkinter**.comp** files, compile code, display errors, show the AST, and generate x86 assembly code.

Features:

- Code editor and console.
- Buttons: New, Open, Save, Compiler, Parse Tree, and Generate Assambler.
- Source selection support.
- Lexical, syntactic and semantic validation.

The design includes a main text editor area, an output console, and a set of functional buttons to perform actions such as opening files, saving content, compiling the source code, viewing the parse tree, and generating the corresponding assembly code. Additionally, the interface allows font customization for the editor.

Compilation is executed internally by integrating all system modules: the lexer and parser build the Abstract Syntax Tree (AST), the semantic module analyzes it, and finally, the assembly generator produces the output code. All error messages and results are displayed in the integrated console.

4 Results

The results were observed through test are:
The intial interface shows the options to save, open or create a new file to compile.

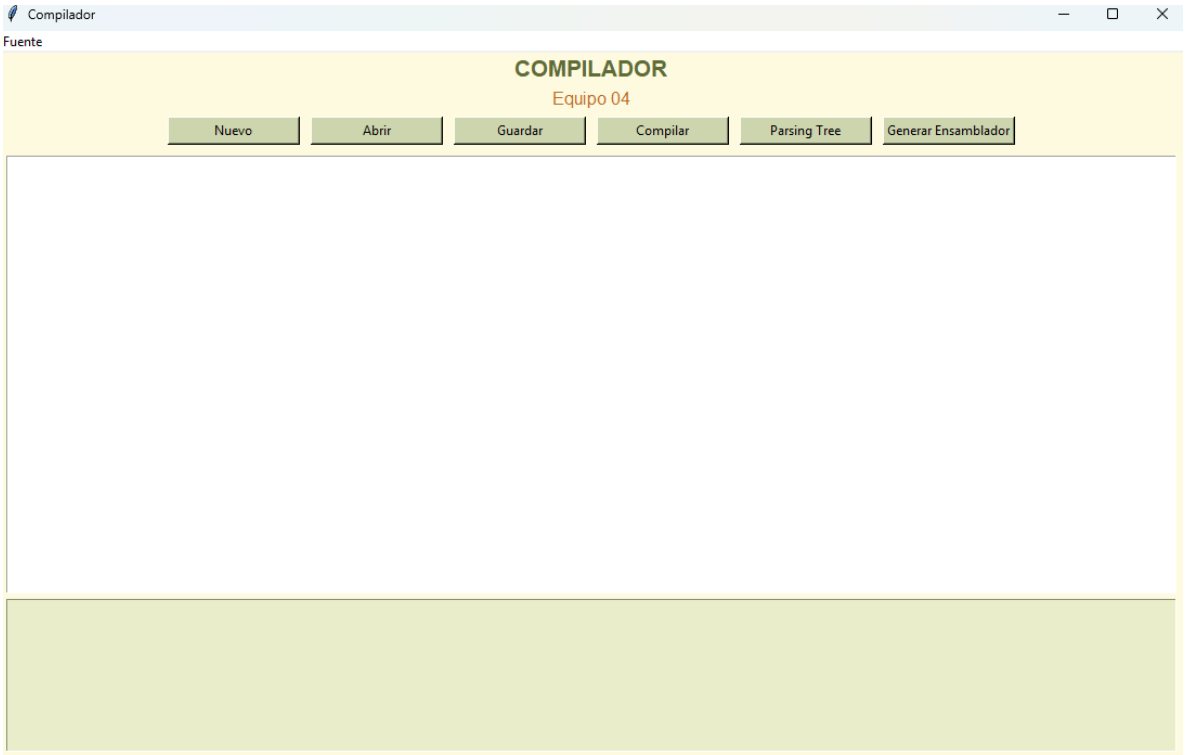

Figure 1: Initial view of the compiler's graphical interface.

The parse tree is display on the interface.

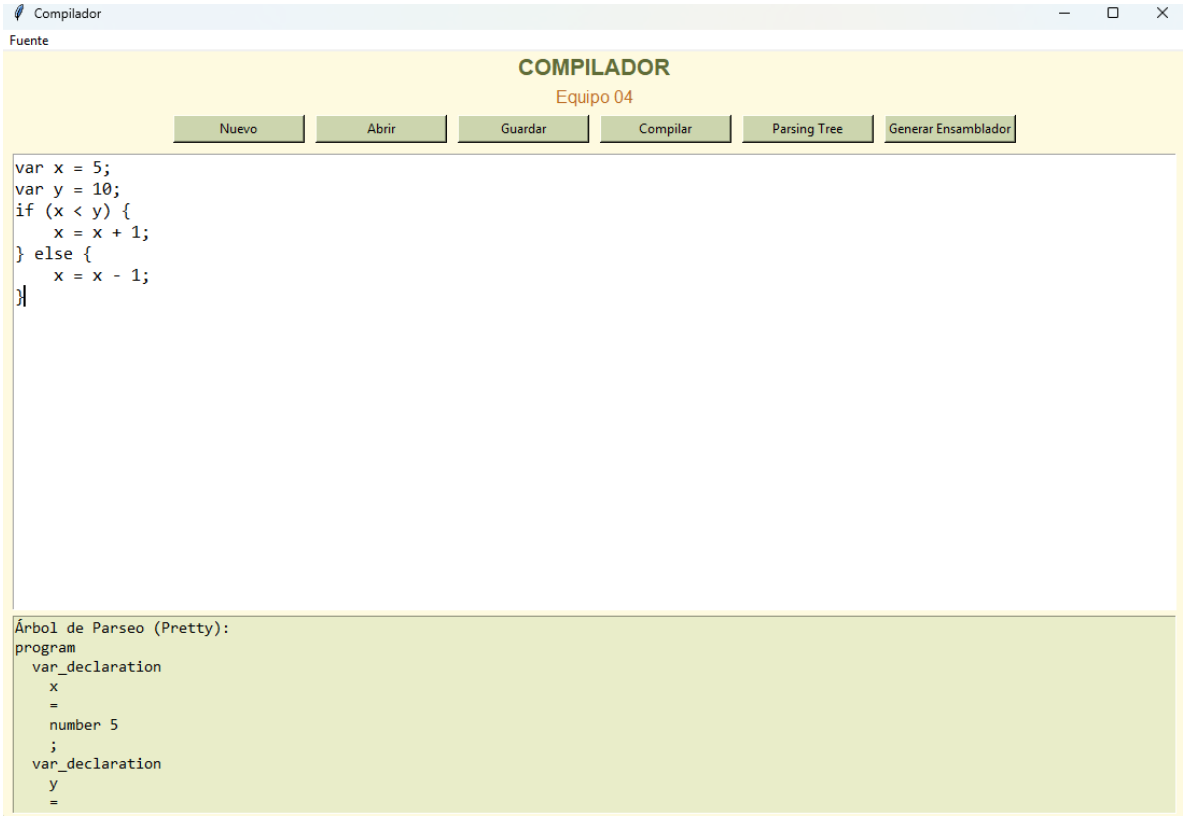

Figure 2: The console displays the generated parse tree.

The intermediate code is display on the screen.

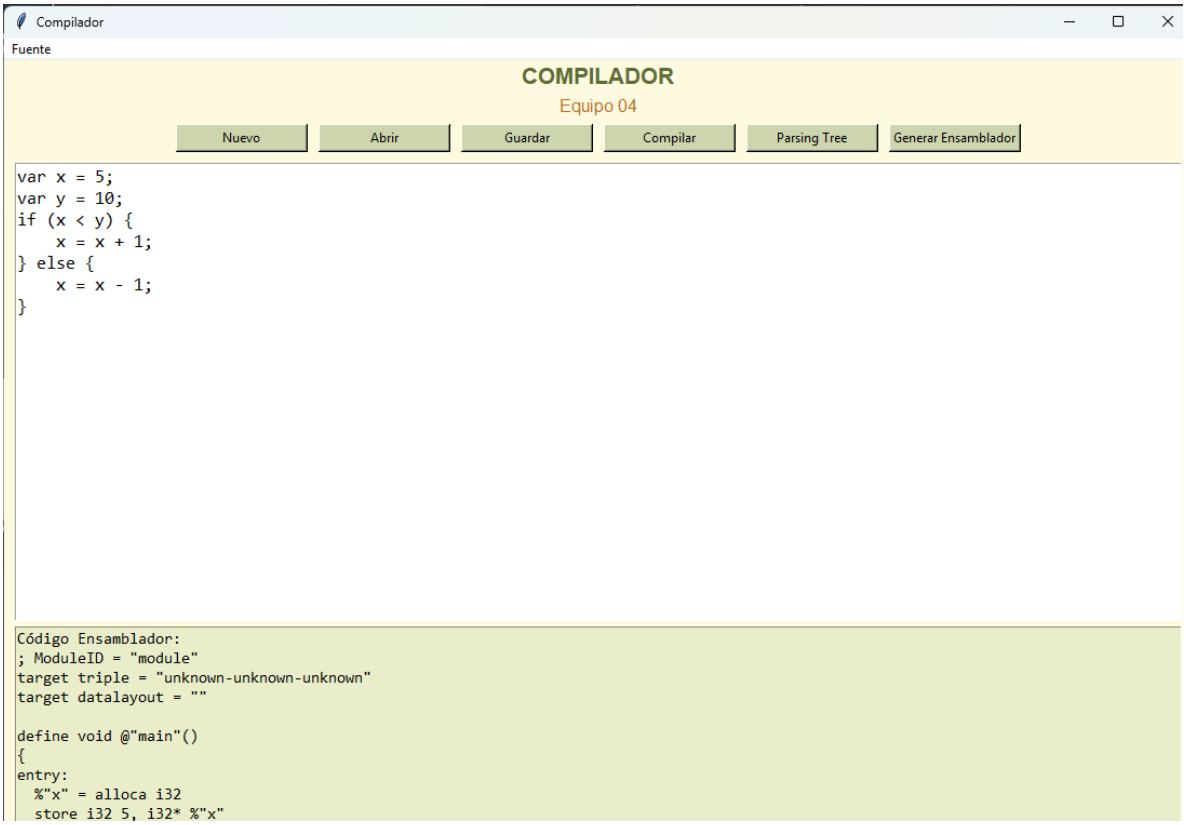

Figure 3: Intermediate code visualization in the console.

With the compilar option, both the parse tree and the intermediate code are displayed on the screen

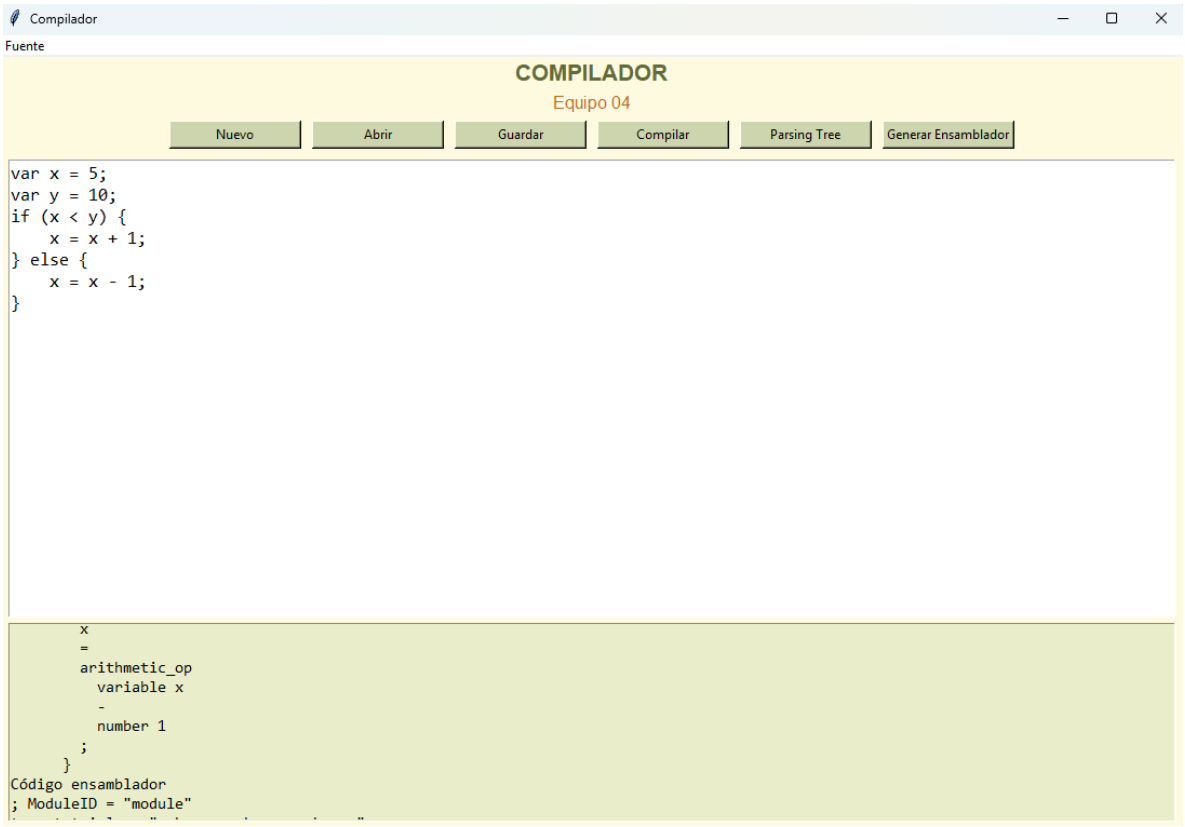

Figure 4: The console displays the resulting AST, followed by the generated intermediate code.

5 Conclusions

In conclusion, the majority of the goals of the projects were met, but there is room for improvement in the future. We acquired more experience and a deeper understanding of the mechanisms working within a compiler in an educational setting. Our simple compiler is able to compile some limited simple code.

Firstly, we were successful in the definition of a formal grammar that defines the language which will be compiled by our compiler. This language is very simple and functionally useless for any real programming job, but serves the main purpose of our work.

Second, we used a python library that enabled a simple lexical and syntactical analysis. This proved challenging to be used, since it take "automagically" many decisions in the building of the parse tree. This might be beneficial for more experienced programmers or projects of a certain nature. However, this affected our workflow, since it was sometimes difficult to understand how the analyzer was working. For future reference, it would be advised to use a different library like GNU Bison, mainly due to more extensive documentation.

On the other hand, one of the benefits of using lark is the simplification of the semantic analysis. Since the Transformer module of Lark allows for convenient annotations of the sysntax tree, the following analysis has fewer considerations to check. Our program was succesful in finding some semantical errors. Creating an assembly output that works for the selected architecture.

The project presents a GUI that creates an abstraction for the user to interact directly with the program, that goal was also successfully achieved.
Finally, some degree of error handling was implemented. If an input not within the language is presented, there will be an indication of an error. However, this indication is not extensive and specific. For future improvements, it would be interesting to further expand said capabilities.

References

[1] A. V. Ho, M. S. Lam, and R. Sethi, *Compilers: Principles, Techniques and Tools*, 2nd ed. Pearson Education, 2009.

[2] How To Use Lark - Guide — Lark documentation, Readthedocs.io, 2020. [Online]. Available: https://lark-parser.readthedocs.io/en/stable/how_to_use.html

[3] T. Pai and P. S. Aithal, A systematic literature review of lexical analyzer implementation techniques in compiler design, *International Journal of Applied Engineering and Management Letters (IJAEML)*, vol. 4, no. 2, pp. 285–301, 2020.

[4] *X86 Assembly*. Wikibooks, October 2017. https://en.wikibooks.org/wiki/X86_Assembly/Arithmetic

[5] TutorialsPoint. *Assembly Programming*. March 2025. https://www.tutorialspoint.com/assembly_programming/index.htm

[6] Gedare Bloom. *x86-64 Assembly Programming Part 1: Registers, Data Movement, and Addressing Modes*. September 2020. https://www.youtube.com/watch?v=lUbPUWtmVUU

[7] Gedare Bloom. *x86-64 Assembly Programming Part 2: Arithmetic/Logic Instructions*. September 2020. https://www.youtube.com/watch?v=uZl_6aQ-FwI

[8] Gedare Bloom. *x86-64 Assembly Programming Part 3: Control Flow Instructions*. September 2020. https://www.youtube.com/watch?v=YQQdiQLTWWE

[9] Davy Wybiral. *Intro to x86 Assembly Language* [Lista de reproducción]. YouTube, s. f. https://youtube.com/playlist?list=PLmxT2pVYo5LB5EzTPZGfFN0c2GDiSXgQe&si=4N-Sr0qVL-vTYVBe

[10] llvm/llvm-project, *The LLVM Project is a collection of modular and reusable compiler and toolchain technologies*, GitHub, 28 May 2025. https://github.com/llvm/llvm-project