Universidad Nacional Autónoma de México

Ingeniería en Computación

Compiladores

Compiler

Student:
320172483
423123463
320241271
320118937
320101179

Group:
5
Semester:
2026-I

México, CDMX. December 2025

Contents

1 Introduction

This section presents the Problem Statement, the Motivation for developing the analysis tools, and the specific Objectives sought through the implementation of the Compiler's **Front-end** (Lexical, Syntactic, and Semantic Analysis) and **Back-end** (Intermediate Code Generation and Execution).

- **Problem Statement.**

  The process of translating a high-level program (such as the C language) into executable code involves several crucial stages, including **Syntactic and Semantic Analysis**, and the subsequent **Intermediate Code Generation**. While a Lexical Analyzer (Lexer) converts the source code into a linear sequence of tokens, it cannot verify the grammatical structure, nor can it ensure that the defined operations have a valid meaning [1, 2].

  The core problem of this project is: *Given a sequence of tokens from a sub-language of C, how can we verify its structural correctness (syntax), detect elementary type and assignment errors (semantics), and, upon success, translate the valid program into a machine-independent representation (**Three-Address Code - TAC**) for eventual execution.*

  For a construct like `int x = 10;`, the task is to use an analysis mechanism to confirm it complies with the variable declaration rule, validate its semantics (type compatibility), and then generate the corresponding intermediate code for this operation, as evidenced by the successful compilation to Assembly and execution in the project results [1, 2].

- **Motivation.**

  The implementation of a complete compiler pipeline—from analysis to code generation—is fundamental for deeply understanding the operation of modern compilers. The primary motivation for this project is to put into practice core compiler theory concepts, such as:

  - Context-free grammars and the use of stacks for **Bottom-Up** analysis.
  - The application of **Syntax-Directed Translation (SDT)** rules for semantic verification (like type checking).
  - The bridge between analysis and execution via the generation of an **Intermediate Representation** (TAC), which is then compiled to assembly for a virtual machine, demonstrating a full compilation cycle [1, 2].

  We developed a diagnostic tool that not only validates syntax and provides crucial feedback on semantic errors (e.g., checking literal assignment validity in

`try_reduce`), but also executes the valid code, as demonstrated by the successful compilation and running of the `while` loop example in the results.

Ultimately, successful analysis and intermediate code generation are mandatory prerequisites for building a fully functional and extensible compiler [1].

- **Objectives.**
  - Design and implement a parser that applies a set of grammatical rules and a **Bottom-Up reduction mechanism** (using an Analysis Stack and reduction functions like `try_reduce`) to verify the syntax of the input code.
  - Incorporate a semantic verification mechanism (**Syntax-Directed Translation**) to validate type and assignment compatibility, ensuring syntactically correct constructions are also semantically valid.
  - Upon successful syntactic and semantic analysis, generate **Three-Address Code (TAC)** as an intermediate representation of the program.
  - Develop the **Back-end** to compile the TAC into machine code for a Virtual Machine and execute it, demonstrating a complete end-to-end compilation process.
  - Generate clear output explicitly distinguishing between syntactic failures, semantic failures (e.g., `Parsing Success! SDT error...`), and successful execution, as demonstrated in the project's results [1, 2].

2 Theoretical Framework

A compiler is a complex system responsible for translating programs written in a high level programming language into a target machine language that can be executed by a computer. Its construction requires the study of formal languages, automata, grammars and computer architecture. In order to make it, it has some phases such as lexical analysis, syntax analysis, semantic analysis, intermediate code generation, machine independent optimizations, machine dependent optimizations, and code generation [1].

2.1 Lexical Analysis

This is the first step in the translation process. The lexical analyzer reads the input characters and groups them into meaningful sequences called tokens such as identifiers, keywords, operators, and literals. The scanner deletes irrelevant elements like whitespace and comments, producing a clean sequence of language symbols for the parser [1].

Lexical analyzers are frequently implemented using finite automata derived from regular expressions. Regular expressions are ideal for token specification, enabling automatic construction of scanners using tools or algorithms based on deterministic finite automata [1, 3].

2.2 Syntax Analysis (Parsing)

The purpose of syntax analysis is to determine whether the sequence of tokens conforms to the grammatical structure of the programming language. The parser focuses on the structure of the program; this structure is defined by context-free grammar, which specifies the acceptable patterns and hierarchies in the language [1].

To perform this analysis, compilers use two types of parsers:

Top-down parsing: Top-down parsing starts from the grammar's start symbol and predicts which production rules to apply, expanding nonterminals until the input is matched. It constructs the parse tree from the root toward the leaves by choosing rules based on the next input tokens [1].

Bottom-up parsing: Bottom-up parsing begins with the input tokens and repeatedly reduces them to higher-level grammar constructs until the start symbol is produced. It builds the parse tree from the leaves upward by identifying handles and applying reductions in reverse order of derivation [1, 2].

2.3 Semantic Analysis

This phase is responsible for ensuring that a syntactically correct program also makes logical and meaningful sense according to the rules of the language. Type checking is at the heart of this phase because it enforces how values, expressions, and variables can legally interact [1].

To perform these tasks, the compiler uses a symbol table, which is a data structure that tracks identifiers, their declaration types, scopes, memory locations and additional attributes. When the compiler encounters an expression, it consults the symbol table to confirm that both operands are numeric types and that the operator is valid for them; if types are incompatible, the compiler reports a semantic error [1].

2.4 Intermediate Representations

This is a crucial form of the program used by the compiler to bridge the gap between the high-level source language and low-level machine code. The main purpose is to provide a uniform, simplified and machine-independent representation of the program so that later phases can operate efficiently [1].

An effective IR captures the most important semantics of the program while removing unnecessary syntactic detail. There are several common forms of IR such as Abstract Syntax Trees, Three-Address Code, or Static Single Assignment [1, 2].

2.5 Code Optimization

Code optimization is dedicated to improving the efficiency of the intermediate representation without changing the program's behavior, aiming to produce faster machine code. During this phase, the compiler analyzes the program's data flow and control flow to detect redundancies, dead code, constant expressions and more. This is achieved by restructuring instructions, simplifying expressions and reorganizing loops and memory accesses [1, 2].

2.6 Code Generation

This part is responsible for translating the optimized intermediate representation into actual machine code or assembly that can run on a specific hardware architecture. In this phase, the compiler maps IR operations to concrete instructions while respecting the target machine's constraints, such as available registers, instruction formats, addressing modes and calling conventions. It performs critical tasks like instruction selection, register allocation and instruction scheduling [1, 2].

The goal is to produce correct, efficient and optimized machine code that preserves the program's semantics while leveraging the hardware capabilities as effectively as possible [1, 2].

2.7 Error Handling and Diagnostics

A compiler must detect and report lexical, syntactic and semantic errors with useful diagnostics. Some strategies are panic mode, phrase-level recovery and error productions. It emphasizes the importance of continuing analysis to report multiple errors per compilation [1, 3].

2.8 Run Time Systems and Memory Management

Run-time environment design is closely related to compiler decisions on calling conventions, lifetime analysis and memory access patterns [1, 2].

3 Results

3.1 Program Execution

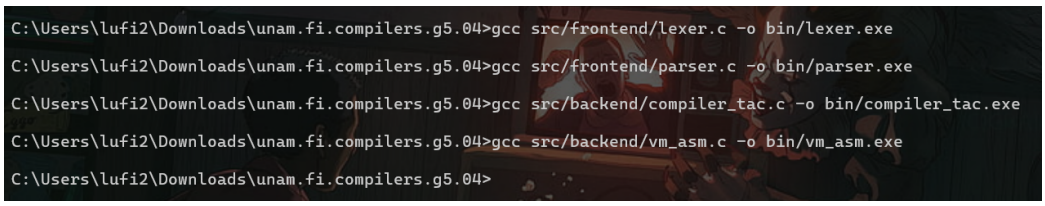
Before running the project, you need to compile the four main modules.Open your terminal in the project's root folder and run the following commands in order:

3.2 Compiling the Frontend (Analysis)

```
gcc src/frontend/lexer.c -o bin/lexer.exe
gcc src/frontend/parser.c -o bin/parser.exe
```

3.3 Compiling the Backend (Syntax)

```
gcc src/backend/compiler_tac.c -o bin/compiler_tac.exe
gcc src/backend/vm_asm.c -o bin/vm_asm.exe
```



(a) Compiling the Backend

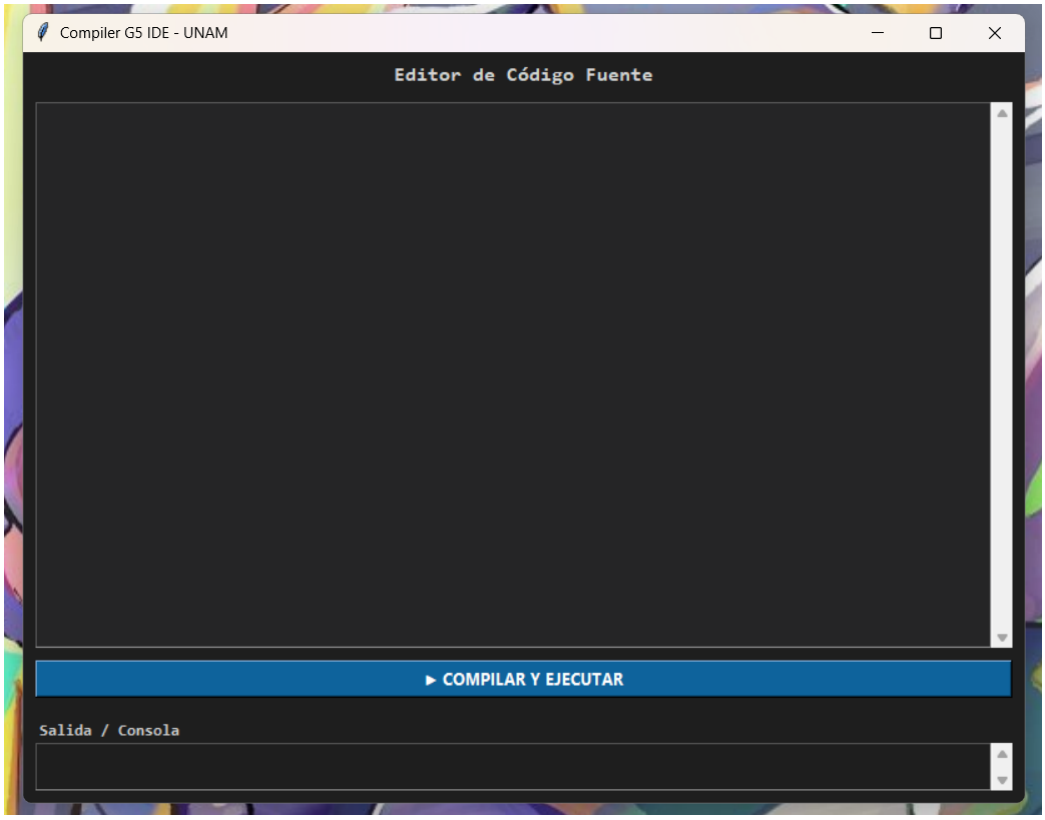
Note: Ensure that the folder `bin` exists. If not, create it with:

```
mkdir bin
```

3.4 Graphical User Interface

A visual IDE-style environment that allows you to write code, compile it, and view the output in an integrated console. Requires Python to be installed.

```
python ide.py
```



(a) Graphical User Interface

3.5 Option B: Command Line

Run the code from the terminal if you don't want to do it in the interface.

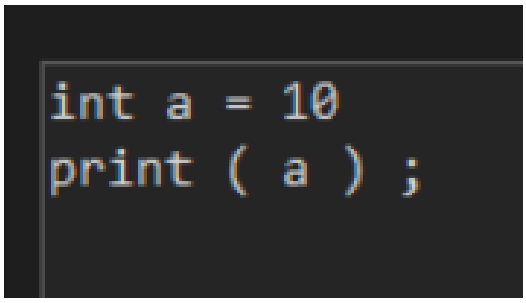
Example:

```
C:\unam.fi.compilers.g5.04>run.bat examples\espacios.txt
```

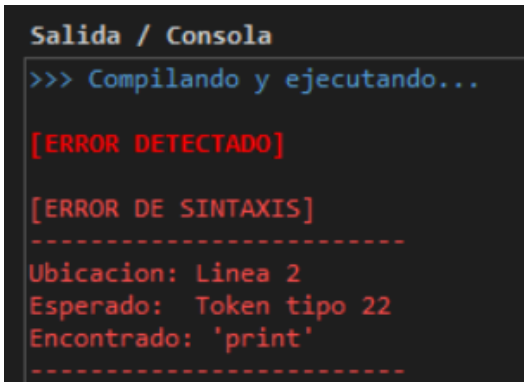
Use the `run.bat` script to process a text file directly. This script connects all stages of the compiler automatically.

```
run.bat examples\archivo_prueba.txt
```

Program implementation

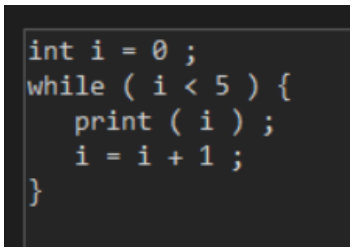


(a) Example with error

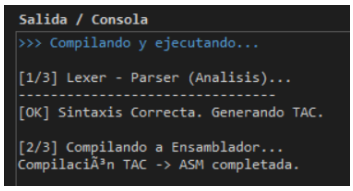


(b) Results example with error

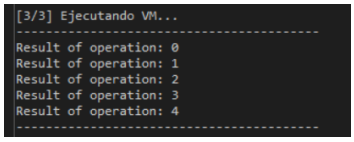
Figure 3: Example with error and its result



(a) Example While

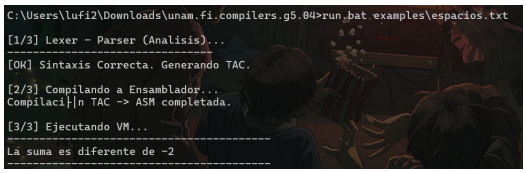


(b) Results example While part 1

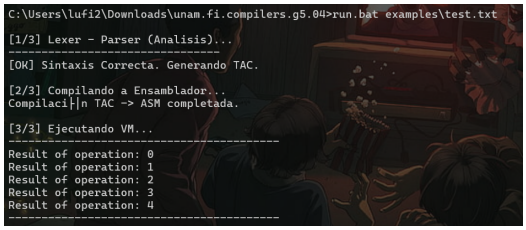


(c) Results example While part 2

Figure 4: Example with excercise While



(a) Example 1



(b) Example 2

Figure 5: Examples in CMD

4 Conclusion

We can conclude that the development of the Syntactic Analyzer and the Syntax-Directed Translation (SDT) Rules was essential for integrating the structural and meaning analysis stages of the compiler. Although the implementation focuses on a reduced subset of the C language, the project demonstrates a practical understanding of a *Bottom-Up* analysis mechanism based on the explicit use of a stack, as seen in the `parser.c` source code. The key to the project lies in the implementation of semantic verification, where the logic of the `try_reduce` function is extended to discriminate between valid constant assignments (numbers) and invalid assignments (literals or strings), fulfilling the requirement to provide outputs that clearly distinguish between syntactic and semantic errors, as requested by the specifications. This work lays the foundation for future compiler expansions, including code generation and the handling of more complex data structures.

References

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison-Wesley, 2006.

[2] A. W. Appel and M. Ginsburg, *Modern Compiler Implementation in C*. Cambridge University Press, 1997.

[3] S. Gonzalez, "Compiladores", class notes, Facultad de Ingeniería UNAM, Ciudad de México, México, Agosto, 2025.

[4] S. Gonzalez, "Compiladores1", class notes, Facultad de Ingeniería UNAM, Ciudad de México, México, Agosto, 2025.