UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

COMPUTER ENGINEERING

COMPILERS

Group: 5

Teacher: M.C. RENÉ ADRIÁN DÁVILA PÉREZ
Delivery date: June 8, 2025

Compiler

Team: 8

Account Number	Last Name	Middle Name	First Name(s)
320334489	Jiménez	Elizalde	Josué
320067354	Medina	Guzmán	Santiago
320054831	Tavera	Castillo	David Emmanuel
320218666	Tenorio	Martínez	Jesús Alejandro
117004023	Salazar	Rubí	Héctor Manuel

Semester 2025-2

Contents

1 Introduction **3**
- 1.1 Problem Statement . . . 3
- 1.2 Motivation . . . 3
- 1.3 Objectives . . . 3

2 Theoretical framework **4**
- 2.1 Compiler Components and Execution Architecture . . . 4

3 Development **6**
- 3.1 Lexer . . . 6
- 3.2 Parser . . . 6
- 3.3 Semantic . . . 7
- 3.4 Assembler . . . 8
- 3.5 Server . . . 9

4 Results **10**
- 4.1 Tests . . . 10
- 4.2 Use case . . . 13

5 Conclusions **14**

6 References **14**

1 Introduction

1.1 Problem Statement

We were asked to create a compiler. This tool should be able to read a source code file, analyze it through different compilation stages, and execute it. The compiler must support basic constructs such as if statements, loops, function definitions, class declarations, and arithmetic/logical operations. It should include an assembler for a simplified low-level language and additionally expose all functionalities via a web server.

1.2 Motivation

We were motivated to build a compiler using `Python` because of its readability, flexibility, and robust support for string processing and regular expressions. Designing a custom language and its corresponding compiler allowed us to explore every major stage of language processing from lexical analysis to interpretation. Adding an assembler and a web API extended the project's educational value, simulating both high-level and low-level program execution and enabling external integration through a user interface or frontend.

1.3 Objectives

Some of the objectives are:

- Create a lexical analyzer that uses regular expressions to tokenize source code.
- Produce a parser capable of generating an *abstract syntax tree* (AST) from tokens.
- Perform semantic analysis to validate the program structure and variable/function usage.
- Implement an interpreter to execute the AST and manage program state.
- Simulate a simple assembler with support for arithmetic, logic, jumps, and register management.
- Develop a web server with endpoints to handle code analysis requests and respond with tokens, AST, semantic errors, or execution results.

2 Theoretical framework

2.1 Compiler Components and Execution Architecture

A compiler is a software system that transforms high-level source code into a form that can be executed or interpreted by a machine. In our implementation, we divided the compiler into three classical phases—*lexical analysis*, *syntax analysis*, and *semantic analysis*—followed by a runtime *execution server* that interprets or runs the processed code. Each phase is implemented using `Python` and designed to mirror standard compiler construction principles, providing a modular structure and enabling efficient testing and debugging.

Lexical Analysis

The *lexical analyzer*, or *lexer*, is responsible for converting a raw input string into a structured list of tokens. These tokens represent the smallest meaningful elements of the language, such as keywords, identifiers, operators and punctuation. In our system, the *lexer* uses regular expressions defined via `Python`'s `re` module to match specific patterns. The implementation includes a dictionary that maps token types to their corresponding patterns, such as:

- `NUMBER`: pattern `\d+`
- `ID`: pattern `[a-zA-Z_][a-zA-Z0-9_]*`
- Arithmetic operators: `+`, `-`, `*`, `/`
- Logical operators: `&&`, `||`
- Keywords: `if`, `else`, `while`, etc.

The *lexer* processes the input using a loop that performs *longest-match scanning* and ignores irrelevant whitespace or comments. Errors are raised when unexpected characters are found.

Syntax Analysis

Syntax analysis, or *parsing*, takes the list of tokens from the lexer and builds a hierarchical representation known as the *Abstract Syntax Tree (AST)*. This phase is governed by a *context-free grammar (CFG)* that defines the valid structures of the programming language, including expressions, statements, conditionals, and control flow.

The parser in our implementation is a *recursive descent parser*, where each non-terminal rule in the grammar is implemented as a separate parsing function. For example:

- `expression()` handles arithmetic or logical expressions.
- `statement()` processes `if`, `while`, or assignment constructs.
- `block()` handles nested statements in compound form.

This parsing strategy allows clear rule-to-function correspondence and facilitates error reporting when the token stream does not match expected grammar productions.

Semantic Analysis

Semantic analysis is the phase where the AST is checked for *contextual correctness*. While syntax ensures correct structure, semantics ensures correct meaning. Our implementation performs:

- **Type checking**: Ensures operations are applied to compatible types (e.g., integers in arithmetic operations).
- **Undeclared variable detection**: Ensures all identifiers are declared before they are used.
- **Scope checking**: Maintains and checks symbol tables to ensure variables are accessed within valid scope.

The *semantic analyzer* traverses the AST recursively, often using a *visitor pattern*, while maintaining a symbol table implemented as a dictionary that maps variable names to their types or values. It also detects semantic errors such as redefinitions or operations on undeclared variables.

Execution Server and Runtime Behavior

Instead of generating machine code or intermediate representation, our system includes a *server component* that evaluates and executes the AST at runtime. This server simulates the behavior of an interpreter by traversing the AST and executing each node's operation in order. Key features of the execution server include:

- **Handling of control flow**: `if` statements and `while` loops are executed by evaluating their conditions and controlling the flow of statement execution.
- **Symbol table management**: A runtime environment tracks variable values and scopes.
- **Error handling**: Runtime errors such as division by zero or undefined variables are caught and reported with meaningful messages.

3 Development

3.1 Lexer

In the first part of the code in `TOKEN_SPECIFICATION`, we define the tokens using regular expressions.

Each line defines a token type with its own regular expression. Are detected:

- Keywords (`def`, `import`, `if`, `while`, `return`).
- String literals (`"text"`, including f-strings).
- Numeric constants (`123`, `3.14`).
- Identifiers (`var_name`).
- Operators (`+`, `-`, `*`, `==`, `!=`, etc).
- Punctuation marks (`()`, `[]`, {}).
- Whitespaces (just to ignore them later).

A unique regular expression unified is created, where each group has a name, this is useful to be able to classify easily each token in the analysis.

As a first step, verify that the user has provided a file as an argument, if not, handle errors. If a file was provided as an argument, save its name, open it for reading its content and if not, handle the error. To avoid problems with accents or special characters `utf-8` is used. If the file was able to reading, `lexer()` function is called with the file content and print the results: the tokens classification and the total amount.

Define the `lexer()` function to analyze the code, then an empty set is created, there is where we are going to put each token.

Then every token is found with help of `re.finditer()`, ignore whitespaces, count the total amount of tokens and save them in the set. Basically, the lexer function counts each token as it finds it. `re.finditer()` searches for all `regex` matches in the source code. It ignores whitespaces and classifies each token by its type (keywords, identifiers, etc). Return the token list and the total amount.

3.2 Parser

Now, a *syntactic analyzer* (`parser`) is implemented for a programming language. Its purpose is to take as input a list of tokens, previously generated by a *lexical analyzer* (`lexer`), and

convert it into an *Abstract Syntax Tree* (AST) that structurally represents the source code. This tree is essential for subsequent stages such as interpretation or code generation.

The `Token` class represents the most basic elements of the language, containing a type and a value. From these tokens, AST nodes are built to model the hierarchical structure of the program. Among the defined nodes are `ProgramNode` for the root node, `FunctionNode` and `ClassNode` for functions and classes, `IfNode`, `WhileNode`, `ForNode`, and `TryNode` for control structures, as well as `AssignmentNode` and `ReturnNode` for specific statements. There are also nodes to represent function and method calls, data structures (lists, dictionaries, tuples), and unary and binary operators.

The `Parser` class is the central component. It contains methods that allow traversing the sequence of tokens, detecting syntactic patterns, and building the corresponding nodes. For example, `parse_stmt()` identifies and analyzes different types of statements, while `parse_expr()` and `parse_term()` handle expressions. Helper functions such as `cur()`, `look()`, and `consume()` are used to access tokens and advance through the input.

This parser supports a wide range of language constructs, including function and class declarations, `import` statements with aliases, control structures (`if`, `for`, `while`, `try` / `except` / `else` / `finally`), assignments, mathematical and logical expressions, function and method calls, and data structures such as lists, dictionaries, and tuples. It also supports unary operations such as `not` or `-`, and binary operations like `and`, `or`, `in`, `+`, `-`, among others.

3.3 Semantic

This code constitutes an implementation of a complete *interpreter*, which includes both *semantic analysis* and the execution of a program represented by an *Abstract Syntax Tree* (AST). It is designed to work with the nodes generated and imported from the parser module. The architecture follows a visitor pattern approach, where a visitor traverses the AST and executes the corresponding actions based on the type of each node.

First, the `NodeVisitor` class is defined, serving as a base infrastructure for traversing nodes. Its visit method performs a kind of double dispatch by invoking the appropriate `visit_TipoDeNodo` method for each type of AST node. If a specific method is not found, it falls back to `generic_visit`, which generically traverses the attributes of the node.

Semantic analysis is implemented in the `SemanticAnalyzer` class. This class validates the correct use of *identifiers*, manages variable scope through a stack of environments (scopes), and detects common errors such as the use of undeclared variables, calls to non-existent functions, or misplaced statements like `break` or `return`. It also keeps track of defined functions and classes to ensure there are no duplicates and that all used symbols are properly declared. Additionally, it handles the context of loops and functions to validate

the use of `break` and `return`, and analyzes `try` blocks along with their `except`, `else`, and `finally` handlers.

The interpreter is defined in the `Interpreter` class, which inherits from `NodeVisitor`. This component directly executes the AST nodes after semantic analysis. It implements binary and unary operators using functions from the `operator` library, and supports user-defined functions, classes (though in a very basic form), control structures (`if`, `while`, `for`, `try` / `except`), function and method calls, and data structures such as lists, tuples, and dictionaries. To simulate `return`, a special exception (`_ReturnSignal`) is used. The state of global variables is stored in a dictionary called `self.globals`, allowing the execution environment to persist across function calls.

The `visit_FunctionNode` method stores a function as a "callback" in the global environment. Each call creates a temporary local environment, evaluates the function body, and then restores the previous environment. Classes are implemented as dictionaries of methods, without full support for attributes or inheritance. Methods and operators that are not explicitly defined raise errors with clear messages.

Finally, the high-level function `link_and_run` orchestrates the entire process: it first performs *semantic analysis* on the AST, and if no errors are found, it invokes the interpreter to execute the program. It returns a dictionary that includes the generated output (e.g., from `print` calls) and the final state of the global variables.

This modular design allows the parser and interpreter to be used independently or in an integrated manner, and it is easily extensible. New language constructs could be added, along with more advanced object handling, support for decorators, `lambda` functions, list comprehensions, or runtime optimizations. Overall, the code implements an educational flow of analysis and execution for a custom programming language, and serves as a foundation for building compilers, interpreters, or interactive educational environments.

3.4 Assembler

The `SimpleAssembler` class implements a *simple assembler* and *interpreter* that simulates a reduced set of assembly instructions and registers. This *assembler* allows defining instructions with labels, arithmetic and logical operations, conditional and unconditional jumps, printing values, and program termination.

The `__init__` constructor initializes the set of labels (`self.labels`), the list of instructions (`self.instructions`), the generated output (`self.output`), and defines how many arguments each instruction expects through `self.arg_counts`. Additionally, it sets the set of valid registers (`self.valid_registers`) along with their bit sizes via `self.register_sizes`. Then, it initializes all registers to zero in the `self.registers` dictionary.

The `parse` method receives the source code as a string and splits it into lines, ignoring

comments and empty lines. Lines containing a label (ending with :) are stored in the labels dictionary with their corresponding position in the instruction list. The other lines are stored as instructions to be executed.

The `validate_instruction` method verifies the validity of an instruction before its execution. It checks that the number of arguments is as expected, that the registers used are valid, and that in the case of the `DIV` instruction, division by zero is not attempted. It also validates that the source and destination registers have the same size for operations that require it.

Program execution takes place in the `execute` method. It sequentially iterates through the stored instructions, parsing and executing each one according to its type. It supports instructions such as `MOV` (assignment), arithmetic operations (`ADD`, `SUB`, `MUL`, `DIV`), logical operations (`AND`, `OR`, `NOT`), comparison (`CMP`), conditional jumps (`JNE`) and unconditional jumps (`JMP`), printing (`PRINT`), and termination (`HALT`). When a jump label is encountered, the program counter (`pc`) is adjusted to the corresponding position to alter the execution flow.

Finally, the `run` method allows executing the entire process: it first parses the code with `parse`, then executes it with `execute`, and returns a dictionary with the final state of the registers and the generated output.

3.5 Server

This section implements a web server using `Flask` that provides an `API` for parsing source code. The server is configured to allow `CORS` requests, facilitating its use from frontend applications.

The server exposes a single main entry point, the `/analyze` route, which receives `POST` requests with a `JSON` body containing the source code (`code`) and the desired analysis mode (`mode`). Depending on the specified mode, different processing stages are applied.

Before analysis, the *lexer* behavior is defined in the `lexer` function, which uses `Python`'s `tokenize` module to split the source code into tokens. A custom mapping (`TOKEN_MAP`) is applied to translate `Python` token types to the types expected by the custom `parser`, with operators (`OP`) and punctuation handled separately. Keywords are recognized using the `keyword` module. Relevant tokens are stored in a list for the `parser`, while a structured summary is returned for the user interface, including token types, their unique values, counts, and the total number of tokens found.

If the analysis mode is '`asm`', the code is treated directly as assembly. In this case, the `SimpleAssembler` simulator, defined in another module, is executed, and the result is returned with the register values and generated output. This completely bypasses the lexical and syntactic analysis stages.

For the other modes, the `lexer` is applied first. In '`lex`' mode, only the *lexical analysis* is returned, including token statistics. In '`full`' mode, the *Abstract Syntax Tree* (AST) is constructed using a custom `Parser`. If there are syntax errors, the exception is caught and an error message is returned.

In '`sem`' mode, in addition to constructing the AST, a full *semantic analysis* is performed. This is carried out through the `link_and_run` function imported from the semantic module, which validates the code and executes it if no errors are found. The AST is returned converted into a `JSON-serializable` format, along with a list of semantic errors (if any), and the program output.

To convert AST objects into `JSON-compatible` structures, the helper function `to_dict` is defined, which recursively traverses lists, tuples, objects with attributes (`__dict__`), and primitive types.

Finally, an if `__name__ == '__main__'` block is included to run the server locally on port 5000 with debug mode enabled, allowing for faster development.

4 Results

4.1 Tests

Figure 1: Compiler Analysis.

The tool is called **Compiler Analysis** and allows performing analysis at different levels:

- Lexical only
- Lexical + Syntax
- Lexical + Syntax + Semantic
- Assembler mode (not selected in the image)

In the image, the *Lexical* option is selected. The *lexical analysis* has classified the tokens in the code.

Figure 2: Abstract Syntax Tree.

If the "`Lexical+Syntax`" option is selected, the Abstract Syntax Tree (AST) generated by the compiler for the previously provided `Python` code is displayed. If the "`Lexical + Syntax + Semantic`" option is selected, a section is shown indicating whether your code has semantic errors.

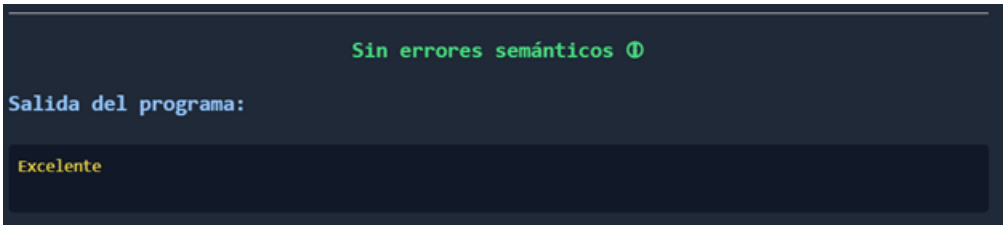

Figure 3: Program output.

If the last option is selected, the "`Assembler Mode`" is enabled. In this section, a small program written in simulated assembly is executed. The code loads the value `5` into register `A` and the value `10` into register `B`. Then, it compares both registers using the instruction `CMP A, B`. Since the values are different (`5` $\neq$ `10`), the condition of the conditional jump `JNE DIF` is met, causing the program's flow to jump to the `DIF` label. Within this section, the instruction `PRINT B` is executed, which prints the content of register `B` — that is, the number `10`. The instructions following `JNE` in the original flow (`PRINT A` and `JMP END`) are not executed because the conditional jump skipped that part of the code.

Finally, the output from the assembler correctly shows the values in the registers (`A = 5` and `B = 10`), and the output produced is the number `10`, indicating that the control flow simulation works correctly. In summary, the assembler implemented in the tool properly handles basic instructions, comparisons, and conditional jumps, which demonstrates that it is a functional representation of a simple assembly language.

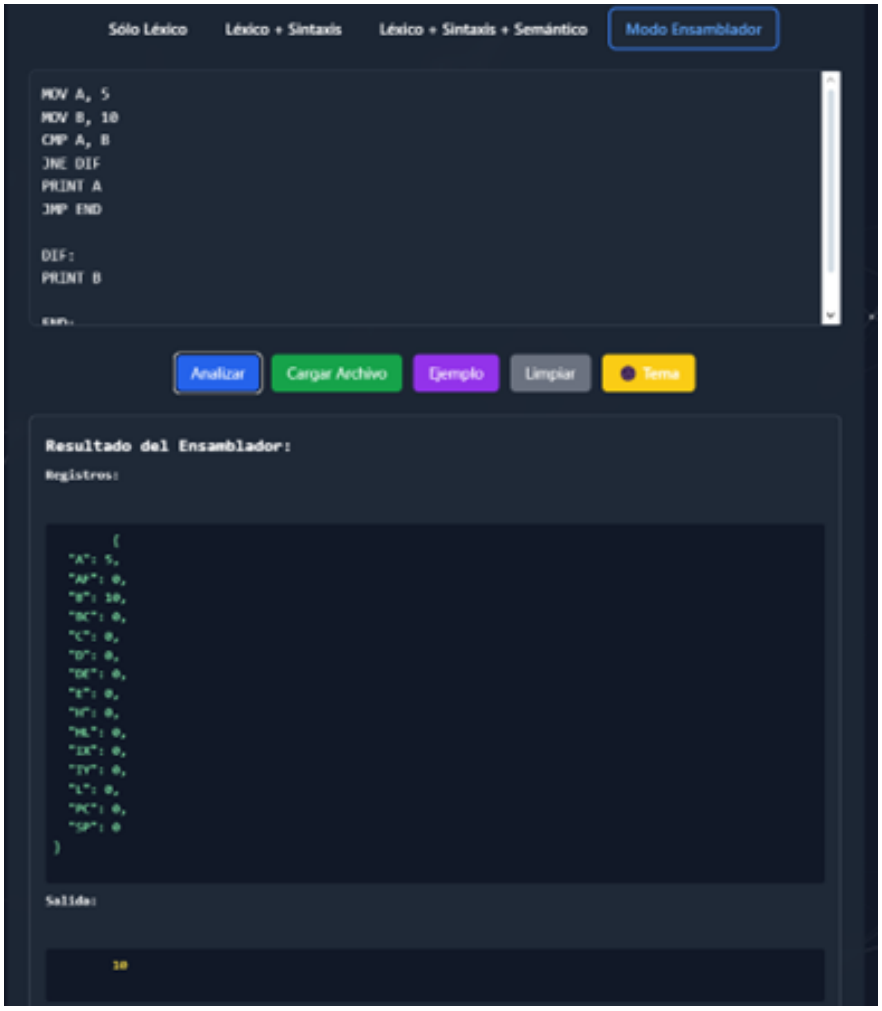

Figure 4: Assembler Mode.

4.2 Use case

Compiler Education

This is one of the primary applications. Students can write code and immediately see how it is broken down into tokens (*lexical analysis*), how it is organized into structures (*syntactic analysis*), and how its meaning is interpreted (*semantic analysis*). It also allows them to see how the code is translated into a lower-level language (*assembler*), facilitating a comprehensive understanding of the entire compilation process.

5 Conclusions

The implementation of the *lexer* and its subsequent analysis illustrate how theoretical concepts from formal languages and automaton can be effectively applied to solve practical problems in code processing. By defining tokens through regular expressions and organizing their recognition systematically, a precise and efficient classification of source code is achieved, which is essential for later stages of the compiler. This process highlights the importance of a solid foundation in computational theory for designing tools that translate text into meaningful structures, enabling deeper analysis and correct program execution. Furthermore, the ability to represent and handle different token types demonstrates the flexibility and power of *lexical analysis* as a crucial first step in programming language processing.

6 References

[1] TREMBLAY, Jean-Paul, y SORENSON, Paul. *The Theory and Practice of Compiler Writing*. McGraw-Hill, 1985.

[2] AHO, Alfred, SETHI, Ravi, y ULLMAN, Jeffrey D. *Compiladores: Principios, técnicas y herramientas*. Addison-Wesley Iberoamericana, 2000.

[3] AHO, Alfred V., LAM, Monica S., SETHI, Ravi, y ULLMAN, Jeffrey D. *Compilers: Principles, Techniques, and Tools*. 2ª ed., Pearson Education, 2007.

[4] WAITE, W. M., y GOOS, G. *Compiler Construction*. Springer Science & Business Media, 2012.

[5] WIRTH, Niklaus. *Compiler Construction*. Vol. 1, Addison-Wesley, 1996.

[6] LOUDEN, Kenneth C. *Compiler Construction: Principles and Practice*. PWS Publishing Co., 1997.

[7] STEELE, Peter W., y TOMEK, Ivan. *Z80 Assembly Language Programming*. Computer Science Press, Inc., 1987.