Universidad Nacional Autónoma de México

Computer Engineering

Compilers

LEXICAL ANALYZER

Students:
320206102
316255819
423031180
320117174
320340312

Group:
05

Semester:
2026 - I

Mexico City, September 2025

Index

1 Introduction

In computing, engineers and programmers face complex problems that require precise solutions through algorithms described in programming languages. For this reason, the fundamental task of compilers and interpreters is to correctly translate the source code into machine language in order to carry out the actions required by the hardware.

This process begins with a lexical analyzer or *Lexer*, which is responsible for extracting and classifying the different types of *tokens* retrieved from the high-level language (HLL), so they can later be processed and translated into machine language. Therefore, it is important that the *Lexer* obtains the *tokens* without any missing information (e.g., incorrect names, missing characters) and classifies them correctly (e.g., numbers within constants, correct identification of keywords according to the language).

Moreover, the correct functioning of the *Lexer* ensures that subsequent phases of the compiler, such as syntactic and semantic analysis, can be carried out without additional errors. A failure in this first stage could propagate through the rest of the process, leading to incorrect translations or even preventing the program from being compiled at all. In this sense, the *Lexer* acts as the first quality filter of the source code, making sure that every elementary component is properly recognized and stored.

With the implementation of the *Lexer*, we expect to achieve the following objectives:

- Extract the source code from a character string and directly from a text file.
- Fully analyze the HLL and extract all contained *tokens*.
- Avoid any loss or distortion of the recovered information.
- Keep an accurate count of the recovered *tokens*.
- Correctly classify the *tokens* for their subsequent validation in the next stage of compiler development.
- Implement a graphical user interface.
- Provide a solid foundation for later stages of the compiler, reducing the complexity of syntactic and semantic analysis.

2 Theoretical Framework

Lexical analysis is a fundamental phase in the process of software compilation. Its main purpose is to transform the source code, written in a high-level language, into a sequence of tokens that can later be processed by the syntax analyzer. A token is defined as a meaningful unit of the program, such as identifiers, operators, constants, keywords, literals, punctuation marks, and special characters [1].

The lexical analyzer, commonly known as a lexer or scanner, operates by reading the source code line by line or character by character and extracting lexemes, which are then categorized into tokens according to predefined rules. During this stage, unnecessary elements such as whitespace, tabs, and comments (single-line or multi-line) are discarded, since they do not contribute to the syntactic structure of the program. This preprocessing simplifies subsequent phases by ensuring that the parser works exclusively with relevant lexical units [2].

From a functional perspective, the lexical analysis process can be understood as a pipeline: lexemes enter the scanner, are examined and validated (often through regular expressions), and are finally emitted as tokens, each associated with a type. The definition and precision of these token rules are crucial. We have [1], [3]:

- **Keywords:** Words with a specific meaning in the language.
- **Literals:** Notation for the constant values of some built-in types.
- **Punctuation:** Symbols used to give structure and organization.
- **Identifier:** Names used to identify variables, functions, classes, etc.
- **Operator:** Symbols that specify operations.
- **Constants:** Fixed values that do not change.

Lexical analyzers commonly rely on regular expressions to define the patterns that describe valid tokens in a programming language. Regular expressions provide a concise and powerful formalism for specifying the structure of identifiers, keywords, operators, and other lexical units. For example, a regular expression can capture the general structure of identifiers as a combination of letters, digits, and underscores, or describe numeric literals as sequences of digits with optional decimal points or signs. These expressions are then translated into rules that guide the lexical analyzer in recognizing and classifying lexemes accurately [1].

Although, implementations vary depending on the programming language, the underlying principles remain constant. Tools like Flex are widely used for automatically generating lexical analyzers from regular expressions in languages such as C and C++, whereas in Python, lexical analyzers are often manually implemented using built-in string processing and regex libraries [4], [5].

In summary, lexical analysis serves as the bridge between source code and syntactic analysis, ensuring that only well-defined lexical elements are passed to subsequent stages of compilation. This makes it one of the most crucial phases in the design and operation of modern compilers.

3 Development

The project was implemented as a lexical analyzer with a graphical user interface, designed to combine the theoretical principles of compiler construction with practical execution. The lexical analyzer represents the first stage in the compilation process: it scans the input program and converts it into a sequence of tokens, which are the smallest syntactic units of meaning. These tokens will be used by the parser to verify grammatical correctness and structure.

3.1 Implementation Details

The implementation was divided into several essential modules. First, a set of categories was established to guide the classification of tokens. These categories included:

- **Keywords:** Reserved words of the programming language (e.g., `int`, `float`, `if`, `return`), which define control structures, data types, or language-specific commands.
- **Identifiers:** User-defined names for variables, functions, and other entities. These follow naming conventions that allow letters, digits, and underscores, but must start with a letter or underscore.
- **Operators:** Symbols representing arithmetic, logical, assignment, or relational operations (e.g., $=, >, +$).
- **Punctuation:** Elements used to structure the code, such as commas, semicolons, parentheses, and braces.
- **Constants:** Numeric values, both integers (e.g., 10, 0) and floating-point values (e.g., 3.14159).
- **Literals:** Character or string values enclosed in quotes, which represent constant data to be used in the program.

The analyzer applied a sequence of regular expressions to identify each type of token. These expressions were carefully ordered to avoid conflicts, for example, distinguishing between operators of one or two characters (> vs. >=). Comments and whitespace were explicitly ignored to focus only on syntactically relevant elements.

In addition to lexical rules, a context-free grammar (CFG) was defined to formalize the structure of declarations. This grammar demonstrated compiler design concepts

such as right recursion, left factorization, and epsilon productions to handle multiple identifiers in a declaration. For example, the declaration:

```
int x, y, z;
```

produces the following tokens:

- Keyword: `int`
- Identifiers: `x, y, z`
- Punctuation: `, , ;`

The corresponding grammar is:

```
S -> Decl
Decl -> Type VarList ";"
Type -> TypeSpec TypeList
TypeList -> TypeSpec TypeList | ε
TypeSpec -> "int" | "float" | "char" | "double" | "short" | "long"
          | "signed" | "unsigned" | "const" | "static" | "volatile"
          | "extern"
VarList -> "id" VarTail
VarTail -> "," "id" VarTail | ε
```

- **Non-terminals:** S, Decl, Type, TypeList, TypeSpec, VarList, VarTail
- **Terminals:** "int", "float", "char", "double", "short", "long", "signed", "unsigned", "const", "static", "volatile", "extern", "id", ",", ";"

Derivation sequence:

$$
\begin{aligned}
S &\Rightarrow \text{Decl} \\
&\Rightarrow \text{Type VarList ";"} \\
&\Rightarrow \text{TypeSpec TypeList VarList ";"} \\
&\Rightarrow \text{"int" TypeList VarList ";"} \\
&\Rightarrow \text{"int" } \varepsilon \text{ VarList ";"} \\
&\Rightarrow \text{"int" "id" VarTail ";"} \\
&\Rightarrow \text{"int" "x" "," "id" VarTail ";"} \Rightarrow \cdots \Rightarrow \text{"int" "x" "," "y" "," "z" ";"}
\end{aligned}
$$

Finally, the system was integrated with a graphical interface. The interface included:

- A code editor, where the user could type or load source code.
- A token analysis view, showing each token with its type, value, and position in the source code.
- A classification tab, summarizing tokens by categories.

This modular interface allowed immediate feedback: once the code was analyzed, users could visualize the results in tables, summaries, and categorized lists, making the theoretical concepts more tangible.

3.2 Testing

The analyzer was validated with different snippets of C code like:

- **Simple declaration:**
  - **Input:** `int x, y, z;`
  - **Output:** keyword `int`, identifiers `x, y, z`, punctuators `, , ;`.
- **Floating-point initialization:**
  - **Input:** `float pi = 3.14159;`
  - **Output:** keyword `float`, identifier `pi`, operator `=`, floating constant `3.14159`, punctuator `;`.
- **Conditional statement with string literal:**
  - **Input:**

```
if (x > 0) {
    printf("x is positive");
}
```

  - **Output:** keyword `if`, identifier `x`, operator `>`, integer constant `0`, identifier `printf`, string literal `"x is positive"`, plus the appropriate punctuation.
- **Loop with increment:**
  - **Input:** `for (int i = x; i < 10; i++)`
  - **Output:** keyword `for`, keyword `int`, identifier `i`, operator `=`, identifier `x`, operator `<`, integer constant `10`, operator `++`, and required punctuation.
- **Comments ignored:**
  - **Input:**

```
// check value
int a = 5; /* block comment */
```

  - **Output:** keyword `int`, identifier `a`, operator `=`, constant `5`, punctuator `;`. (Both comments were skipped.)

In all cases the analyzer produced the expected tokens, handled comments correctly, and accurately tracked positions.

4 Results

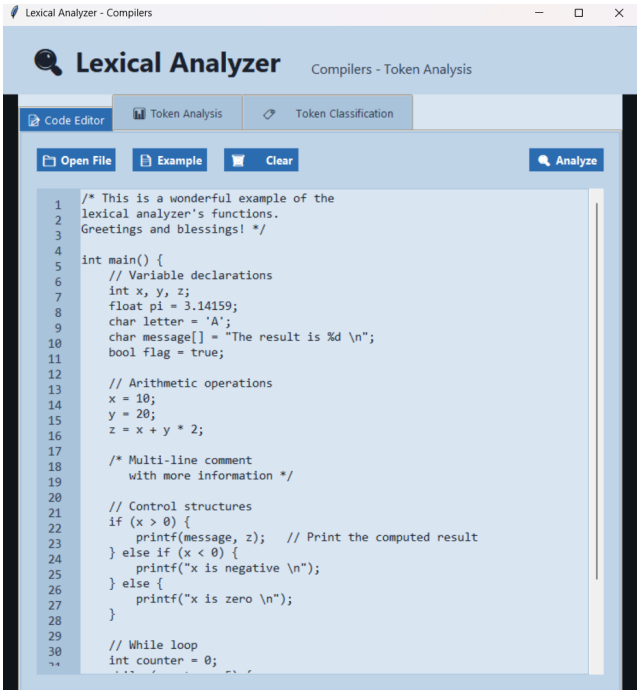

Figure 1: Program Start

The program starts with a menu that contains three main options: The first one is called Code Editor, and it has other sections, starting with one that allows you to upload a txt file saved on the computer; the second one is for charging an example previously defined in the code, and the last one is for cleaning the entry.

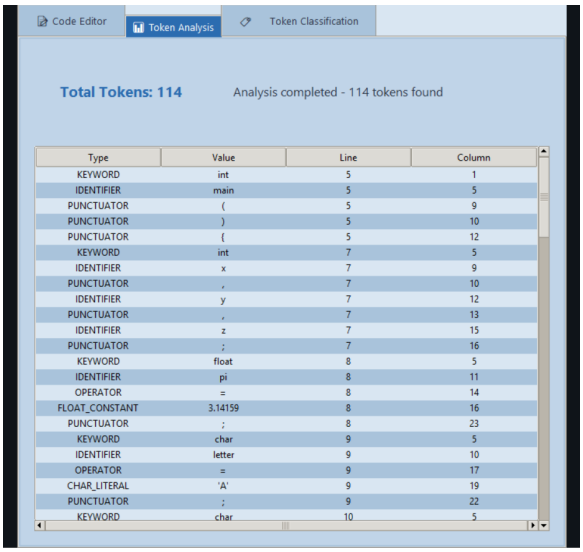

Figure 2: Table with tokens

After inserting a code into the program and clicking the Analyze button, a window will appear showing a table that contains the total number of tokens and a table with a description of the type, value, the line on which it appears, and the column of the counted token.

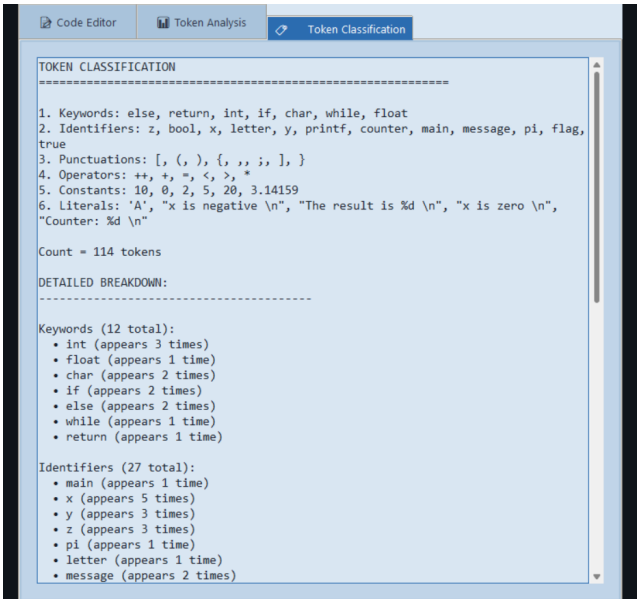

Figure 3: Classification of tokens

Finally, the last section contains token classification by type, such as keywords, identifiers, operators, constants, and literals. Letting us know what characters were used and how many times they appear in the upload code.

5 Conclusions

The development of the lexical analyzer made it possible to verify the application of the theory behind compiler construction. As the basis of the compilation process, it plays an essential role by transforming characters into lexical units while eliminating comments and unnecessary spaces, regardless of whether the input is read from strings or from a text file.

During the implementation, concepts such as lexemes, tokens, and regular expressions were used to define the rules for identifying Keywords, Identifiers, Operators, Constants, Literals, and Punctuation. As a result, the system recognizes lexical units without loss of information and maintains a secure count of the recovered tokens.

The Lexer task simplifies compiler design, increases efficiency, and enables the early detection of errors in the source code, while at the same time establishing solid foundations for the subsequent syntactic and semantic phases.

The graphical interface help us to have a clear and accessible visualization of the results, making a connection between theoretical foundations and the practice of lexical analysis.

Finally, the results obtained confirm the accomplishment of the stated objectives and demonstrate that the correct application of grammars and regular expressions is an indispensable pillar in the development of modern compilers.

6 References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston, MA: Addison-Wesley, 2006. [Online]. Available: https://www-2.dc.uba.ar/staff/becher/Hopcroft-Motwani-Ullman-2001.pdf.

[2] J. R. Levine, *Flex & Bison*. Sebastopol, CA: O'Reilly Media, 2009. [Online]. Available: https://web.iitd.ac.in/~sumeet/flex__bison.pdf.

[3] D. Grune, H. E. Bal, C. J. H. Jacobs, and K. G. Langendoen, *Modern Compiler Design*, 2nd ed. New York, NY: Springer, 2012. [Online]. Available: https://dpvipracollege.ac.in/wp-content/uploads/2023/01/Modern.Compiler.Design.2nd.pdf.

[4] V. Paxson, A. Mason, and J. R. Levine, *Lex & Yacc*, 2nd ed. Sebastopol, CA: O'Reilly Media, 1992. [Online]. Available: http://www.nylxs.com/docs/lexandyacc.pdf.

[5] Python Software Foundation. "Re — regular expression operations," Accessed: Sep. 21, 2025. [Online]. Available: https://docs.python.org/3/library/re.html.