Universidad Nacional Autónoma de México
Facultad de Ingeniería

Computer Engineering

Subject: Compilers (0434)

Account numbers:
320075669
423066940
423035975
320199866
320061682

Group: 5

Semester: 2026 - I

Mexico, CDMX. September 25th 2025

Contents

1 Introduction **2**

2 Theoretical Framework **2**
2.1 Lexical Components 3
2.2 Pattern Recognition Techniques 3
2.3 Secondary Functions of a Lexical Analyzer 4
2.4 Practical Tokenization Workflow 4
2.5 Connecting to Syntax Analysis: Context-Free Grammars 4

3 Body of Work (Development) **5**
3.1 Tool and Language Selection 5
3.2 Grammar 5
3.3 Token Definition 6
3.4 Lexical Analyzer Implementation 8
3.5 Graphical User Interface (GUI) Design 8

4 Results **9**

5 Conclusions **11**

References **12**

Compilers
Group: 5
UNAM / FI / DIE
Semestre 2026-1

Lexical Analyzer

1 Introduction

- **Problem Statement**: The first phase of the compilation process, lexical analysis, consists of scanning the source code to decompose it into a sequence of fundamental components called *tokens*. The correct identification and classification of these tokens is crucial, as an error in this stage prevents the subsequent syntactic and semantic analysis phases from functioning correctly. The challenge is to build a tool that automates this process for a defined subset of a C-like language.
- **Justification**: A robust lexical analyzer is a fundamental piece in the construction of any compiler. It abstracts the complexity of the source text into a structured format that simplifies subsequent phases. This project serves as a practical application of formal language theory, providing a tangible tool that improves development efficiency by automating a critical and error-prone task.
- **Objectives**:
  - To implement a lexical analyzer in Python capable of processing source code from a string or a file.
  - To correctly identify and classify the required tokens: keywords, identifiers, numeric constants, operators, and punctuation symbols.
  - To provide a total count of all identified tokens in the input source code, complying with the project requirements.
  - To develop a graphical user interface (GUI) that allows the user to input code directly or load it from a file, and that clearly displays the resulting token sequence and their total count.
  - To demonstrate, from a theoretical standpoint, an understanding of Context-Free Grammars (CFG) by modeling an example declaration statement, applying the principles of right recursion and left factoring.

2 Theoretical Framework

The compilation process of any programming language involves an analysis phase and a synthesis phase. This project focuses on **lexical analysis**, the first step of the analysis phase. Its primary task is to read the input characters from the source program, group them into meaningful units called **lexemes**, and produce a sequence of **tokens** as output. This stream of tokens is then passed to the next phase, the syntax analyzer (or parser), transforming raw source code into a more structured format (Aho, Lam, Sethi, & Ullman, 2007) [1].

Key Functions of a Lexical Analyzer

- **Tokenization:** Is the process of breaking a stream of text into smaller pieces called tokens. These tokens may be words, punctuation marks, numbers, or even subword units, depending on the context and application. Tokenizers serve as the foundational layer for many text processing tasks in both natural language processing (NLP) and compiler design (Gupta, 2025) [8]. The lexical analyzer would produce a stream of tokens like:

  - int (keyword token)
  - x (identifier token)
  - = (operator token)
  - 5 (constant token)
  - ; (punctuation token)

- **Removal of Whitespace and Comments:** The lexical analyzer is responsible for removing all unnecessary characters from the source code. White spaces (spaces, tabs) and comments are irrelevant to the actual program logic, so the lexical analyzer removes them (Mahmood, 2024) [9]. This makes the subsequent parsing process much more efficient.

- **Error Detection:** It can detect and report basic errors at the character level, such as an invalid character that doesn't belong to any defined token pattern. There are lexical errors such as:

  - Exceeding length of identifier or numeric constants.
  - Appearance of illegal characters.
  - Unmatched string.
  - Spelling Error.
  - Replacing a character with an incorrect character.
  - Removal of the character that should be present.
  - Transposition of two characters.

- **Symbol Table Management:** Symbol Table is a data structure used in compilers and interpreters to keep track of identifiers like variables, functions, and objects. It stores information about these identifiers throughout their use in a program. It also helps detect errors and handle memory management (Mahmood, 2024) [10].

2.1 Lexical Components

The process of tokenization revolves around three core concepts:

Token: A pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a lexical unit, such as ‘KEYWORD‘ or ‘IDENTIFIER‘.

Pattern: A description of the form that the lexemes of a token can take. For a reserved word like ‘if‘, the pattern is the sequence of characters ‘i-f‘. For an identifier, it might be a letter followed by any number of letters or digits.

Lexeme: A sequence of characters in the source program that matches the pattern for a token. For instance, ‘printf‘ is a lexeme that matches the pattern for an identifier.

2.2 Pattern Recognition Techniques

To identify which strings of characters form valid lexemes, lexical analyzers rely on formal language theory.

- **Regular Expressions**: The patterns for tokens are most commonly specified using **regular expressions**. A regular expression is a formal notation for describing a set of strings. The set of strings defined by a regular expression is known as the language it generates, denoted as $L(r)$. They are the standard for defining the syntax of lexical units.

- **State Transition Diagrams**: While regular expressions define the patterns, they are typically implemented using **state transition diagrams**. These diagrams provide a visual representation of how an analyzer moves between states as it processes an input string character by character to recognize a lexeme. Tools like 'lex' or 'ply.lex' automatically convert regular expressions into optimized state transition diagrams.

2.3 Secondary Functions of a Lexical Analyzer

Because the lexical analyzer is the only phase that reads the entire source code character by character, it is often assigned additional secondary tasks:

- **Stripping Whitespace and Comments:** It removes non-essential information such as comments and whitespace before it reaches the parser.
- **Error Reporting Correlation:** It helps correlate compiler error messages with the source program by tracking line and column numbers.
- **Macro Preprocessing:** In languages that support macros, the lexical analyzer can be responsible for macro expansion.

2.4 Practical Tokenization Workflow

In a practical setting using a tool like Python's 'ply' (which emulates 'lex'), the workflow is as follows:

A) **Token Specification:** First, a list of all possible token names is declared. Then, rules are defined to associate these names with patterns. Simple tokens can be defined with simple regular expression strings. More complex tokens are often defined using functions that contain a regular expression and can perform custom actions, such as converting a string of digits into an integer value.

B) **Scanning Process:** The lexer scans the input string, matching the longest possible prefix of the remaining input against the defined regular expression patterns.

C) **Token Generation:** For each matched lexeme, the lexer produces a token object. This object typically contains attributes such as its 'type' (the token name), its 'value' (the actual lexeme), its 'lineno' (line number), and its 'lexpos' (position within the input string).

Special rules are used to handle ignored characters (like whitespace) and to manage lexical errors, such as encountering a character that does not belong to any defined pattern.

2.5 Connecting to Syntax Analysis: Context-Free Grammars

While lexical analysis deals with regular languages, the next phase, syntactic analysis, works with **Context-Free Grammars (CFGs)**. A CFG is a set of recursive rules used to define the hierarchical structure of a program.

To handle more complex structures, grammar transformations are needed:

- **Right Recursion**: To parse a list of declarations, a right-recursive rule is often used.
- **Left Factoring**: This technique is used to eliminate ambiguity when two or more rules share a common prefix.

These techniques are essential for building a parser that can correctly interpret the stream of tokens produced by the lexical analyzer (Aho et al., 2007).

3 Body of Work (Development)

This section details the process of building the lexical analyzer, from the selection of technologies to the implementation of the user interface. The project was developed entirely in Python, leveraging powerful libraries to achieve the desired functionality efficiently.

3.1 Tool and Language Selection

The choice of tools and programming language was fundamental to the project's success.

- **Python**: Python was selected as the primary programming language due to its clear syntax, readability, and extensive ecosystem of libraries. These features facilitate rapid development and make the code easier to maintain, which is ideal for academic projects.
- **PLY (Python Lex-Yacc)**: For the core task of lexical analysis, we used the `ply` library. It is a well-established Python implementation of the classic Unix tools Lex and Yacc. We specifically used `ply.lex` because it provides a straightforward yet powerful way to define token rules using regular expressions, automatically generating an efficient finite automaton for tokenization. This allowed us to focus on the logic of the language's tokens rather than the low-level implementation of the scanner.
- **Tkinter and ttkbootstrap**: For the graphical user interface (GUI), we chose `Tkinter`, which is Python's standard GUI toolkit. To give the application a modern and professional look with minimal effort, we used the `ttkbootstrap` library, a wrapper around Tkinter's themed widgets (ttk). This combination allowed us to create a functional, cross-platform, and visually appealing interface quickly.

3.2 Grammar

The proposal of CFG for our lexical analyzer is the following:
$S \rightarrow D;$
$D \rightarrow int\ L \mid int\ L = E$
$L \rightarrow (a - z|A - Z|\_)R$
$R \rightarrow (a - z|A - Z|\_|0 - 9)R$
$E \rightarrow (0 - 9)|(0 - 9)E$
Where:

- S is the Start symbol.
- D is the part of the Declaration code.
- L is for the Literals we can use to our variable.
- E is for the Expression we are going to assign to our variable.
- R is for the Rest of the literals and numbers we can use for our variable

This CFG is useful to write the expression $int\ x = 10;$, where we want to analyze and tokenize all the elements to clasify them. Due to some ambiguities it could present non-determinism, so we need to do left factoring for reducing all these cases.

Applying left factoring

The D assignment $D \rightarrow int\ L \mid int\ L = E$ has the common factor $int\ L$, so we can express it as

$$D \rightarrow int\ L\ D'$$
$$D' \rightarrow = E \mid \epsilon$$

Also the expression $E \rightarrow (0-9) \mid (0-9)E$ could be factored in the way:

$$E \rightarrow (0-9)E'$$
$$E' \rightarrow\ (0-9)E' \mid \epsilon$$

This procedure will help to avoid non-determinism, also with the help of the right recursion already done, where we have non.terminal elements in the right side, so this may help with the repetitive and recursive string we can make by stacking letters or numbers, for example.
Finally the grammar is defined by the expressions:
$S \rightarrow D$;
$D \rightarrow int\ L\ D'$
$D' \rightarrow = E \mid \epsilon$
$L \rightarrow (a-z|A-Z|\_)R$
$R \rightarrow (a-z|A-Z|\_|0-9)R$
$E \rightarrow (0-9)E'$
$E' \rightarrow\ (0-9)E' \mid \epsilon$

We could represent the grammar by making a finite state machine, were we have the states and the transitions described for making the example. The states in the finite state machine have two types of transition, the recursive ones and the transitory ones. The recursive ones can make strings very large, from one to infinite elements. The terminal elements are semicolon (just like in the C statements).

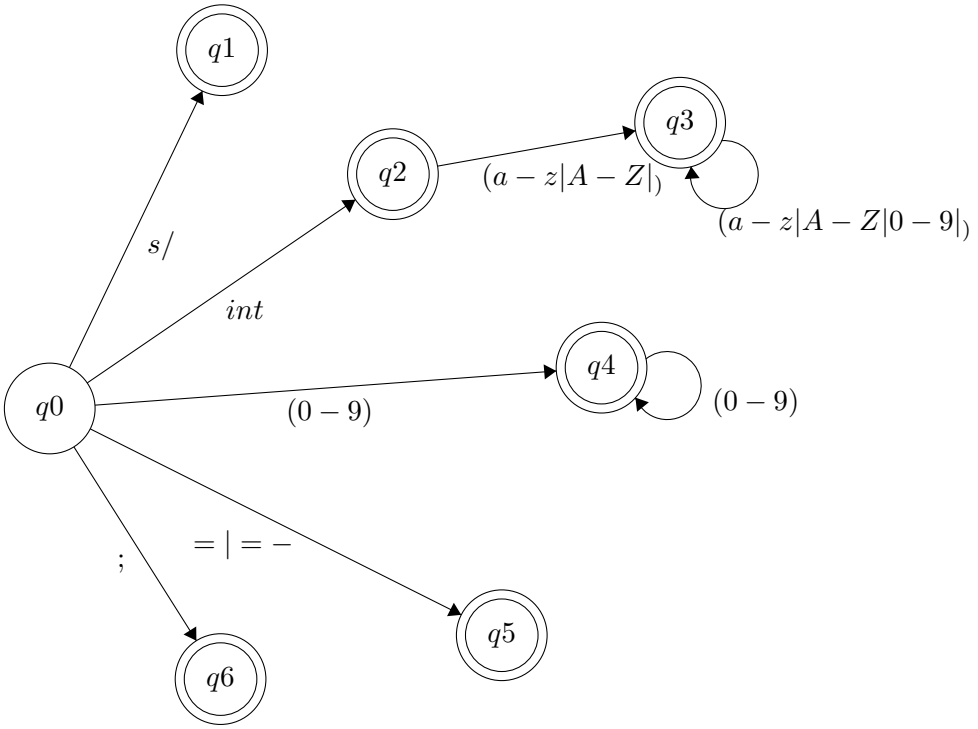

Figure 1. DFA for the presented grammar.

3.3 Token Definition

After designing the grammar, the next step was defining a tuple that contain the name of the tokens that our lexer will recognize. The tokens are:

- Keywords.
- Identifiers.
- Operators.
- Constants.
- Punctuators.
- Literals.

Once we defined the tokens, we established the regular expressions for each token type. Following the conventions of `ply.lex`, these rules were implemented as functions:

- **t_KEYWORDS**: Defined with the pattern:

```
def t_KEYWORDS(t):
    r'\b(const|double|float|int|short|char|long|struct|break|for|if|else|switch|case|do|while|default|goto|void|return)\b'
    return t
```

  The word boundaries $\backslash b$ ensure that it only matches complete words, preventing *int* in *internal* from being tokenized as a keyword.

- **t_IDENTIFIER**: Uses the pattern:

```
def t_IDENTIFIER(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    return t
```

  to match any valid variable or function name, which must start with a letter or underscore and can be followed by any number of letters, digits, or underscores.

- **t_OPERATOR**: Matches common C operators like `==`, `!=`, `=`, `+`, `-`, etc., using the pattern:

```
def t_OPERATOR(t):
    r'==|!=|<=|>=|<|>|=|\+|\-|\*|/'
    return t
```

- **t_CONSTANT**: Recognizes numeric constants, including integers and floating-point numbers, with the pattern:

```
def t_CONSTANT(t):
    r'\d+(\.\d+)?'
    return t
```

- **t_PUNCTUATION**: Identifies punctuation symbols such as parentheses, brackets, braces, and semicolons using the character set:

```
def t_PUNCTUATION(t):
    r'[\(\):\[\]\{\};,.]'
    return t
```

- **t_LITERAL**: Matches string literals enclosed in double quotes with the pattern:

```
def t_LITERAL(t):
    r'"([^"\\]|\\.)*"'
    return t
```

3.4 Lexical Analyzer Implementation

The implementation of the lexer is contained within the `LEX_C.py` script. The core logic is built around the `ply.lex` library.

- **Lexer Construction**: The lexer object is created by calling `lex.lex()`. This function inspects the global scope of the module for any objects (functions or variables) whose names start with `t_` and uses them to build the master regular expression and the underlying finite automaton.
- **Handling Ignored Characters**: Insignificant characters like spaces, tabs, and newlines do not contribute to the program's logic. They are handled by defining a special variable `t_ignore = ' ¬'`, which instructs the lexer to simply discard these characters whenever they are encountered. Additionally, a specific rule for C-style headers *('#include ¡...¿')* was created to ignore them completely.
- **Error Handling**: A robust lexer must handle unexpected input gracefully. The `t_error` function is automatically invoked by `ply` when it encounters a character that does not match any of the defined token rules. Our implementation of this function records an error message indicating the illegal character and its position, and then calls `t.lexer.skip(1)` to discard the character and continue tokenizing the rest of the input.
- **Analysis Function**: The `analyze_code(code)` function serves as the main entry point to the lexer. It takes the source code as a string, feeds it to the lexer, iterates through the generated tokens, classifies them into categories, and formats the results into a single string for display, including a total token count.

3.5 Graphical User Interface (GUI) Design

The GUI, implemented in the main script, provides a user-friendly way to interact with the lexical analyzer.

- **Layout and Components**: The application window is structured logically. A header contains the title and a theme selector. The main area is vertically organized with a large text widget for code input, a frame with action buttons ("Load File", "Analyze Code"), and another text widget for displaying the analysis results.
- **Functionality**: The user can either type code directly into the input area or load it from a `.txt` file using the "Load File" button, which opens a system file dialog.
- **Interaction Logic**: When the "Analyze Code" button is clicked, the `analyze` method is triggered. This method retrieves all text from the input widget and passes it to the `LEX_C.analyze_code` function. The formatted string returned by the function, containing the categorized tokens and the total count, is then inserted into the output text widget, which is set to a disabled (read-only) state to prevent user modification. This clear separation of the UI logic from the analysis logic makes the application modular and easy to understand.

4 Results

With the help of AI, we developed a graphical interface that allows the user to use the lexical analyzer more easily and in a more user-friendly way, making it more convenient to understand and operate the program. Below, we present 5 examples demonstrating the functionality of our program.

- 1. The program allows writing directly within the window. Although it is also possible to upload files with the '.txt' extension, you can write code manually and still run the program to view the tokens. For example, we wrote a simple program to multiply two numbers and then add the result.

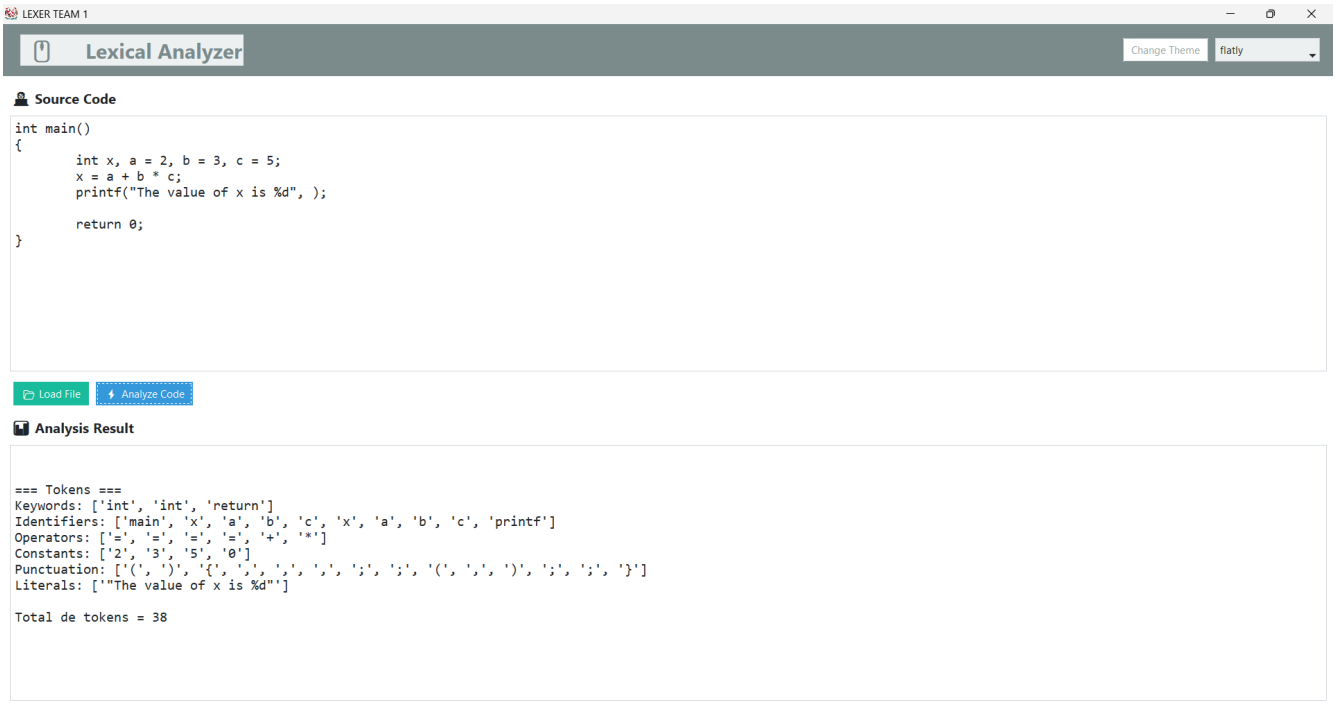

Figure 1: Example 1

- 2. In this example, the code was loaded from the system where the analyzer was executed, and the file describes the algorithm to convert a text into ASCII code.

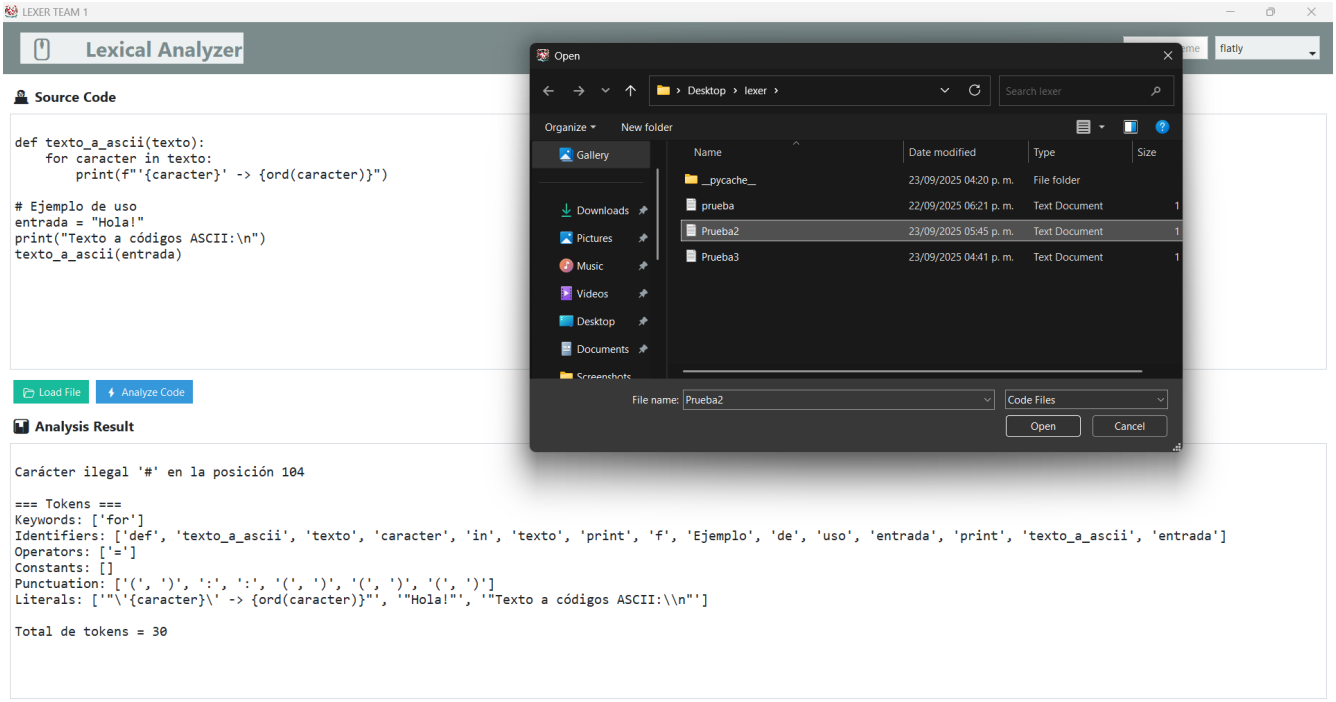

Figure 2: Loading file example

- 3. Once again, as an example of the lexical analyzer's functionality, we loaded another file with a larger code than in Example 1, with the purpose of demonstrating the correct operation regardless of the length of the input.

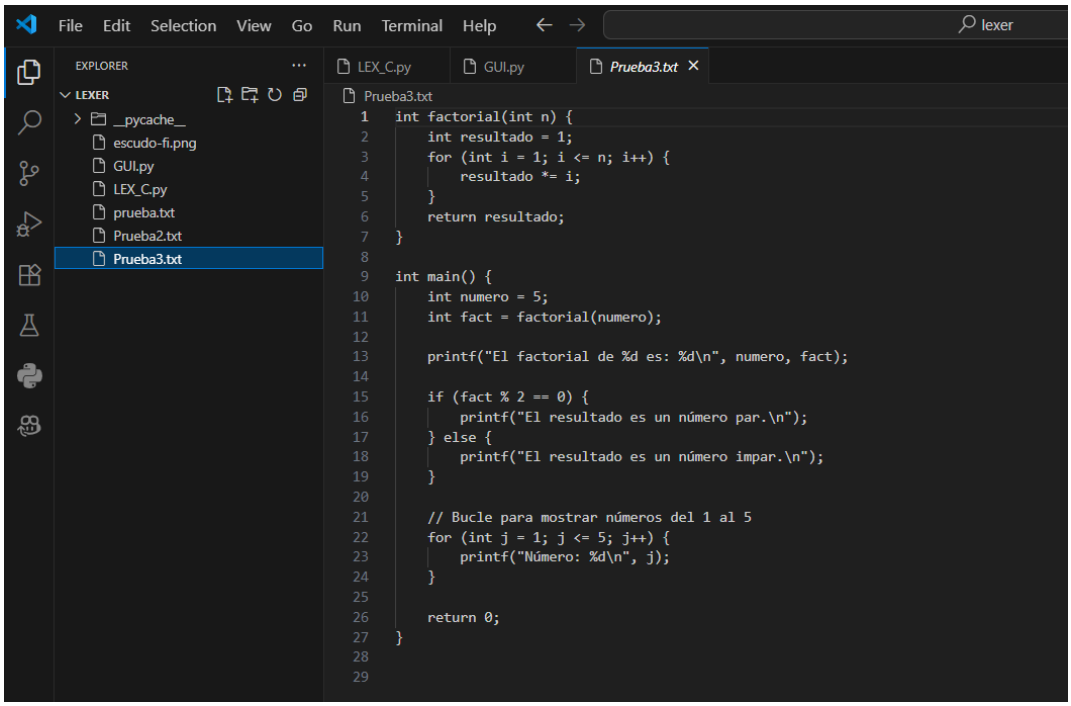

Figure 3: Loading file example 3

Example 3

5 Conclusions

Upon completing this project, we can affirm that all established objectives have been successfully met. The developed tool functions as a precise and efficient lexical analyzer for the defined language subset.

The main goal of creating an analyzer in Python that correctly identifies and classifies keywords, identifiers, constants, operators, and punctuation marks was achieved. The use of regular expressions proved to be a powerful method for pattern recognition, reinforcing the practical application of formal language theory. The final implementation not only produces a detailed stream of tokens but also provides a total count, satisfying a central project requirement.

Furthermore, the development of a graphical user interface significantly enhances the tool's usability. The theoretical objective was also met by successfully modeling a declaration statement with a CFG. This project has provided valuable hands-on experience in the first and fundamental phase of compiler construction, creating a solid foundation for the subsequent stages of syntactic and semantic analysis.

References

[1] Beazley, D. M. (2007). *PLY (Python Lex-Yacc)*. Retrieved September 21, 2025, from https://www.dabeaz.com/ply/ply.html

[2] Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Pearson/Addison Wesley.

[3] Beazley, D. M. (n.d.). *Multiple parsers and lexers. In PLY — Python Lex-Yacc documentation*. Retrieved September 22, 2025, from https://ply.readthedocs.io/en/latest/ply.html#multiple-parsers-and-lexers

[4] Cidecame / Universidad Autónoma del Estado de Hidalgo. (n.d.). *Atributos de los componentes léxicos*. Retrieved September 22, 2025, from http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro32/23_atributos_de_los_componentes_lxicos.html

[5] Editorial Transdigital. (n.d.). *Compiladores: fases de análisis*. Retrieved September 23, 2025, from https://www.editorial-transdigital.org/compiladores-fases-de-analisis/

[6] Su, Y., & Yan, S. Y. (2011). *Principles of compilers: A new approach to compilers including the algebraic method*. Higher Education Press; Springer-Verlag Berlin Heidelberg. ISBN 978-3-642-20834-8.

[7] Silva Bata, M. Á. (2023, June 19). *Analizador Sintáctico y Léxico con Python (PLY)* [Video]. YouTube. Retrieved September 24, 2025, from https://www.youtube.com/watch?v=iXArNJWLYes

[8] Gupta, N. (2025, February 24). *The Comprehensive Guide to Tokenization: Concepts, Techniques, and Implementation*. Medium. Retrieved September 25, 2025, from https://aignishant.medium.com/the-comprehensive-guide-to-tokenization-concepts-techniques-and-implementation-a5f6958e2b2a

[9] Mahmood, S. (2024, December 15). *Lexical Analysis*. Medium. Retrieved September 25, 2025, from https://medium.com/@Saman-Mahmood/lexical-analysis-304503896874

[10] Mahmood, S. (2024, December 13). *Symbol Table in compiler construction*. Medium. Retrieved September 25, 2025, from https://medium.com/@Saman-Mahmood/symbol-table-in-compiler-construction-13970a0025d8

[11] *Lexical Error*. (2025, July 23). Geeks for Geeks. Retrieved September 25, 2025, from https://www.geeksforgeeks.org/compiler-design/lexical-error/