Universidad Nacional Autónoma de México

Ingeniería en Computación

Compiladores

Lexer Report

Alumnos:
320021312
320112179
320184624
320565661
423093137

Grupo:
5
Semestre:
2026-I

Ciudad de México, México. Septiembre 2025

Contents

1 Introduction **2**

2 Theoretical framework **2**

3 Development **3**
3.1 Module Structure 3

4 Results **4**
4.1 Context-Free Grammar (CFG) 4
4.2 Execution and Results 4

5 Conclusions **6**

References **6**

1 Introduction

This document details the creation of a lexical analyzer, a fundamental component in the initial phase of a compiler. The purpose of this report is to document the architecture, functionality, and results of the implementation of the analyzer, which was developed in Python.

This work aims to demonstrate a solid understanding of the principles of lexical analysis by building a program capable of reading a string of text or a source file, identifying basic elements known as *tokens*, and categorizing them according to their type. The application of this process is a crucial step in translating high-level code into a form that can be processed in subsequent stages of the compilation process.

2 Theoretical framework

A **lexical analyzer** (or **lexer**) is the first phase of the compilation process. Its main function is to read the source code, which is a sequence of characters, and group these characters into meaningful units called **tokens**. Each token represents a pattern in the programming language, such as a keyword, an identifier, an operator, or a constant. The lexer does not concern itself with the grammatical structure of the code, but only with the validity and identification of these atomic units. [1]

The process of lexical analysis involves defining different types of tokens and creating an engine capable of recognizing them. The basic tokens for this project include:

- `KEYWORD`: Reserved words of the language, such as `print`. [2]
- `IDENTIFIER`: Names of variables (as long as the variable name starts with a letter, it can also contain numbers), functions, etc. [2]
- `CONSTANT`: Numerical values.
- `OPERATOR`: Symbols that denote operations, such as `=`, `+`, `-`, etc. [1], [2]
- `PUNCTUATION`: Delimiters and other punctuation symbols, such as `(`, `)`, `;`, etc. [1], [2] item `LITERALLY`: String between `""`. [2]
- `EOF`: End of file or input string.
- `INVALID`: Characters or character sequences that do not correspond to any valid token.

Additionally, the project specifications require a demonstration of a **Context-Free Grammar (CFG)**, applying right recursion and left factoring.

- **Right Recursion**: Occurs in production rules where the recursive non-terminal appears at the end of the right-hand side of the production. For example, in ‘StatementList → Statement StatementList‘, the non-terminal ‘StatementList‘ calls itself at the end of the production. This is useful for processing sequences of elements. [1]

- **Left Factoring**: Used to eliminate ambiguity in grammars when multiple productions for the same non-terminal share a common prefix. For example, a grammar with the rules 'A → aB | aC' is refactored to 'A → aD' and 'D → B | C', by extracting the common prefix 'a'. [1], [3]

3 Development

The lexical analyzer was implemented in Python, structuring the code into three main modules: `tok.py`, `lexer.py`, and `main.py`.

3.1 Module Structure

- `tok.py`: This module defines the `Tok` class to represent a token, which stores its type (`tokenType`) and its literal value (`literal`). It also includes the `tokentype` class that acts as an enumerator for the different types of tokens (`KEYWORD`, `IDENTIFIER`, etc.), and a helper function called `isKeyword` that determines if a string is a predefined keyword or an identifier. The set of keywords is defined in the global variable `keyword` and includes `print` and `int`.
- `lexer.py`: Contains the main `Lexer` class, which performs the analysis.
  - `__init__(self, source: str)`: Initializes the lexer with the source code and sets position variables to track reading progress (`pos` and `read_pos`).
  - **Character Reading Methods**: The `readChar` method is the basis of the lexer, which allows it to read the next character in the input.
  - **Scanning Methods**:
    - `skipWhitespace()`: Advances the lexer's position through white spaces, tabs, and newlines.
    - `readIdentifier()`: Consumes alphabetic, and underscore characters, then it identifies if the token is a identifier or a keyword checking if it is in the keyword list.
    - `readNumber()`: Consumes numeric characters to form a numeric constant.
    - `readString()`: Consumes a sequence of characters enclosed in double quotes (`"`). The method correctly handles escape characters such as `"`.
  - `readToken(self)` → `Tok`: This is the main public method. First, it skips white spaces. Then, it examines the current character (`self.ch`) to determine the token type and calls the appropriate scanning method. For operators and punctuation, it simply consumes the character and creates a `Tok` with the correct type and literal. If the character is not recognized, it classifies it as `INVALID`.

- `main.py`: This script serves as the command-line interface for the analyzer. It allows the user to scan a string (`-s`) or a file (`-f`). The `run()` function uses the `Lexer` class to iterate through the tokens, printing their type in lowercase and, finally, showing the total token count, which matches the expected output format.

4 Results

4.1 Context-Free Grammar (CFG)

Here is a CFG model that represents a set of simple statements and assignments, demonstrating the concepts of right recursion and left factoring as requested in the theoretical requirements of the project.

Original grammar (with ambiguity and left recursion):

```
StatementList -> Statement | Statement StatementList
Statement -> Declaration | Assignment
Expression -> Expression Operator Term | Term
Term -> Identifier | Constant
Declaration -> Type Identifier Semicolon
Assignment -> Identifier Operator Expression Semicolon
```

Applying Right Recursion and Left Factoring: The grammar shown above already uses right recursion in the 'StatementList' production. To demonstrate left factoring, let us consider a grammar with an ambiguous rule. In the provided code example, there is no direct ambiguity, but a hypothetical case could be:

```
Statement -> 'if' Expression 'then' Statement |
'if' Expression 'then' Statement 'else' Statement
```

Applying Left Factoring:

```
Statement -> 'if' Expression 'then' Statement Statement_rest
Statement_rest -> 'else' Statement | epsilon
```

This shows how the common prefix ''if' Expression 'then' Statement' is factored out, creating a new non-terminal 'Statement_rest' to handle the remaining productions, thereby eliminating the ambiguity.

4.2 Execution and Results

The `main.py` script was used to test the lexical analyzer with the 'input1.txt' file. The file's content combines two of the examples from the assignment specifications.

Example: 'input1.txt' content

When the script is run with the 'input1.txt' file, the lexer processes the content and generates the following output. The output below is a concatenation of the tokens for both lines in the file.

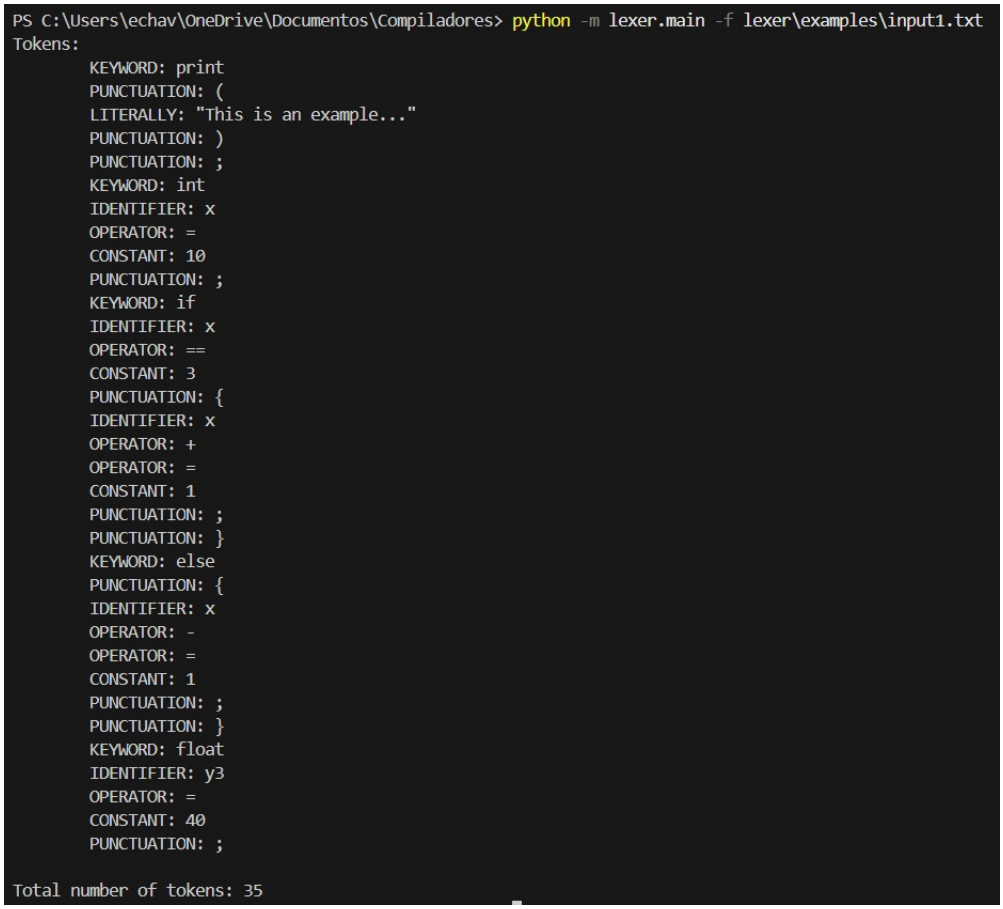

Figure 1: Results of example 1.

The lexer correctly identifies the sequence of tokens:

- For ‘print("This is an example...");‘: ‘print‘ is a ‘keyword‘, ‘(‘ is ‘punctuation‘, ‘"This is an example..."‘ is a ‘constant‘, ‘)‘ is ‘punctuation‘, and ‘;‘ is ‘punctuation‘.
- For ‘int x = 10;‘: ‘int‘ is a ‘keyword‘, ‘x‘ is an ‘identifier‘, ‘=‘ is an ‘operator‘, ‘10‘ is a ‘constant‘, and ‘;‘ is ‘punctuation‘.

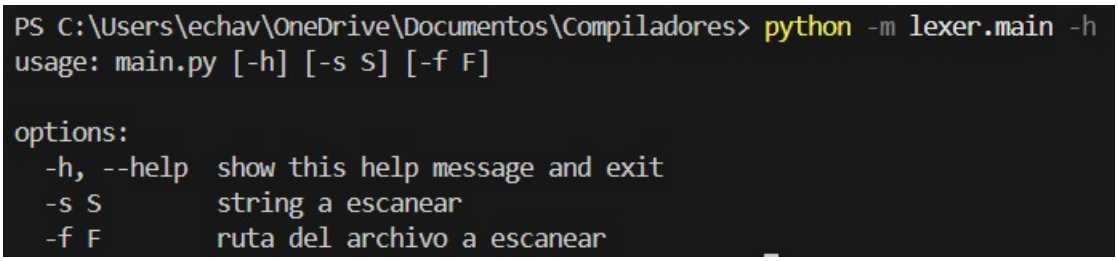

Figure 2: Menu example

This is a menu that provides information on what you can enter when executing.

```
PS C:\Users\echav\OneDrive\Documentos\Compiladores> python -m lexer.main -s "print();"
print();
Tokens:
        KEYWORD: print
        PUNCTUATION: (
        PUNCTUATION: )
        PUNCTUATION: ;

Total number of tokens: 4
```

Figure 3: Example with string from console

This is an example of a fairly simple string entered from the console, and you can see that the parser continues to identify the tokens correctly.

5 Conclusions

The construction of this lexical analyzer in Python has reinforced our theoretical understanding of compilers. The practice of breaking down source code into tokens has made the differences between the components of a language processor evident.

The theoretical framework has been validated through functional implementation, demonstrating the direct application of concepts such as token scanning and the management of different data types. The discrepancy observed between the code output and the example output of the specification document for ‘printf‘ highlights the importance of a clear definition of the language’s keywords. This practical exercise not only met the assignment requirements but also provided a solid foundation for understanding the initial stages of language analysis.

References

[1] TutorialsPoint. “Compiler design – quick guide.” Accessed: YYYY-MM-DD. [Online]. Available: `https://www.tutorialspoint.com/compiler_design/compiler_design_quick_guide.htm`.

[2] GeeksforGeeks. “Working of lexical analyzer in compiler.” Accessed: 2025-09-25. [Online]. Available: `https://www.geeksforgeeks.org/compiler-design/working-of-lexical-analyzer-in-compiler/?utm_source=chatgpt.com`.

[3] N. Code360. “Left factoring in compiler design.” Accessed: 2025-09-24. [Online]. Available: `https://www.naukri.com/code360/library/left-factoring-in-compiler-design`.