Universidad Nacional Autónoma de México

Ingeniería en Computación

Compiladores

Lexer

ALUMNO:

422023645
Hernández Diaz Sebastian

Grupo: #5

Semestre:
2025-I

México,CDMX. Septiembre 2024

Contents

Project Report: Lexical Analyzer

Team 4

1 Introduction

1.1 Problem statement

Lexical analysis is a fundamental part in the development of compilers and interpreters for programming languages. The problem to be solved in this project is the creation of a lexical analyzer that can identify different elements in the source code, such as keywords, operators and literals, in a hypothetical programming language similar to C. The main challenge is to build a system that is capable of recognizing these tokens efficiently and accurately, using the tools provided by Python and the PLY (Python Lex-Yacc) library.

1.2 Motivation

Solving this problem is crucial because lexical analysis is the first step in compiling or interpreting a program. Without proper analysis of the tokens, the compiler cannot properly process the source code, resulting in execution errors or the inability to generate executable code. In addition, implementing this project allows to consolidate key knowledge about language processing and the internal workings of compilers.

1.3 Objectives

- Implement a lexical analyzer using Python and PLY.
- Correctly identify and classify the different tokens in the source code, such as reserved words, operators, literals, and special characters.
- Evaluate the accuracy of the lexical analyzer with a set of tests containing various types of inputs.
- Develop an efficient process that ignores white space and non-relevant lines, thus avoiding interruptions in lexical analysis.

2 Framework

Lexical analysis is the process of reading source code and breaking it down into minimal units called *tokens*, which represent elements such as keywords, operators, identifiers, and literals. This phase is essential in compilation, as it translates the source code into a sequence of tokens that will later be processed by syntactic analysis. To build this lexical analyzer, the **PLY (Python Lex-Yacc)** library is used, which allows you to define the rules to identify each type of token by regular expressions. These expressions allow you to describe patterns of characters that the parser recognizes as distinct tokens. The concepts applied in this project include: In this project we will work with the following language:

$$\Sigma = \{a, b, \ldots, y, z, A, B, \ldots, Y, Z, 0, 1, 2, \ldots, 8, 9, =, ==, !=, <, >, \leq, \geq, \&\&, ++, --, ;, +, -, /, *, \&\}$$

Implementing this language we can define different patterns, helping later in the recognition of regular expressions, just as, variable names, punctuation symbols and positive or negative integers.

- **Regular expressions**: A fundamental tool to describe patterns that allow the identification of tokens such as operators, literals or reserved words.
- **Tokens**: Minimal elements of the source code that represent identifiers, operators, or literals, recognized by the parser.
- **Recognition functions**: These functions allow the analyzer to associate specific patterns with tokens that represent particular entities of the programming language.

Finite Automaton: Lexical analysts frequently use a finite automaton (FA) to recognize and categorize tokens effectively. This mathematical model helps the lexical analyst switch between various states based on input characters, allowing them to determine when a set of characters forms a valid token. Finite automata are deterministic (DFA) and non-deterministic (NFA). Because of their efficiency, DFAs are more common in lexical analysis. We will be working with the following DFA on this project:

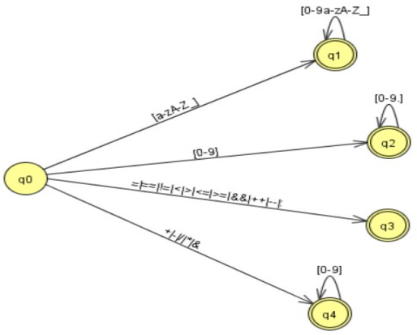

Figure 1: Finite Automaton

Every character in the regular expressions is processed by the DFA, which switches between states until it finds a valid token or detects an error. This makes the DFA a crucial engine for quickly and deterministically identifying the patterns given by regular expressions.

These concepts were fundamental to structure the rules that allow the correct identification of code elements.

Parse Tree

A parse tree, also known as a syntax tree, is a hierarchical tree structure used in the syntactic analysis phase of the compiler. The parse tree represents the grammatical structure of a string according to the rules of a formal grammar. Each node in the tree corresponds to a grammar rule, and the children of the node represent the components of the rule.

The parse tree is the subsequent stage, following lexical analysis. It examines the input's syntactical structure to determine whether or not it follows the proper syntax (of the language in which it was produced). The input string and the language's predefined grammar are used to build the parse tree. The input string is determined to be in the proper syntax if it can be generated using the syntax tree during the derivation process. If not, the syntax analyzer reports the error.

For example, our language allows simple arithmetic expressions and its grammar supports the following rules:

- **Expression**: Expression → Term ('+' — '-') Term
- **Term**: Term → Factor ('*' — '/') Factor
- **Factor**: Factor → '(' Expression ')' — Number

For the input, considering the arithmetic expression: 2 + 4 * (5 - 3) We will build a parse tree based on the grammar rules. The two will represent how the input expression is parsed according to the rules.

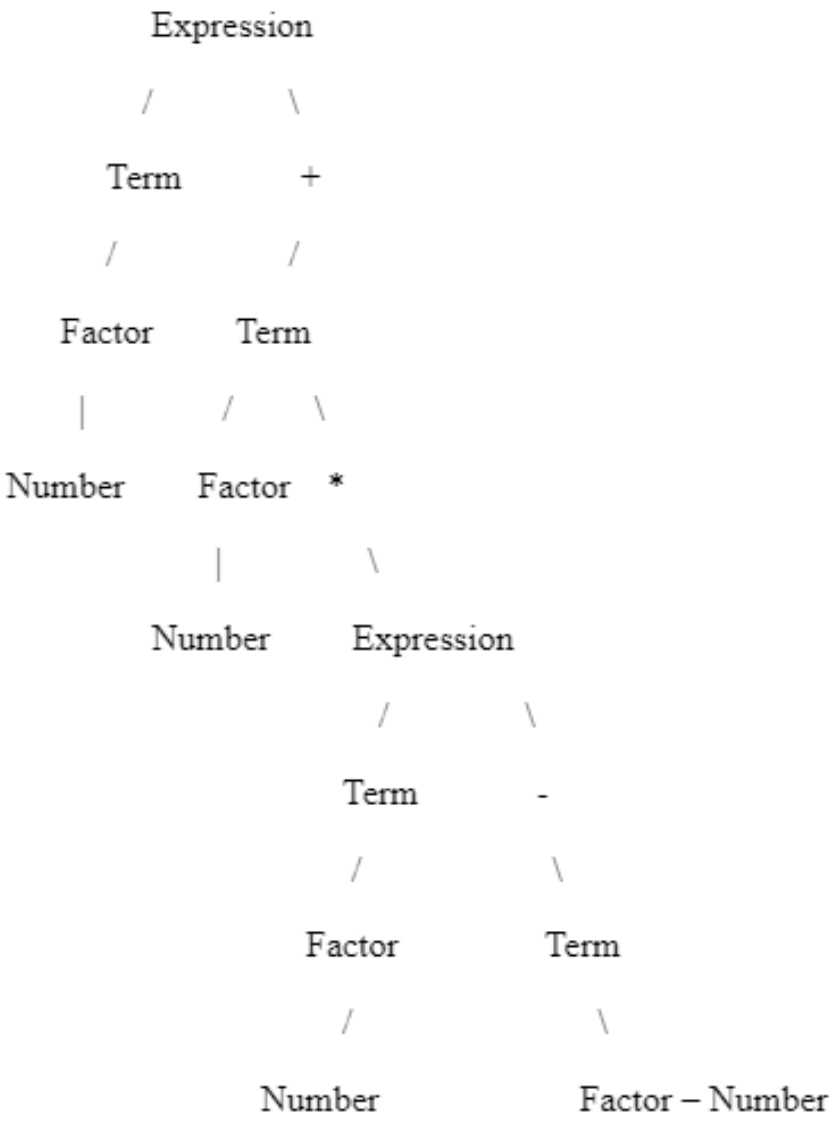

Figure 2: Parse Tree

3 Body

The PLY library, which is created and utilized for the development of Lex and Yacc (lexical analyzer and syntax analyzer), was utilized to implement the lexical analyzer in Python. With some flexibility that helps ensure appropriate implementation, it permits the construction of both components while conforming to their respective rules. A variety of token types were employed in the development of the lexical analyzer, which led to the precise identification of every component of the input utilized to evaluate the analyzer's functioning. These tokens have particular purposes assigned to them in addition to being linked to "regular expressions," which makes them significant.

The tokens are separated into the following categories: division, multiplicity, equals, less, plus, operators, special characters, parentheses, brackets, and division. These tokens are essential. The lexical analyzer was implemented in Python using the PLY library. Each token was identified by a regular expression that describes its pattern in the source code. The implementation was done by dividing the tokens into several categories:

- **Keyword identification**: Specific tokens were implemented for reserved words such as `int`, `float`, `if`, `else`, among others. This was achieved by means of a dictionary of reserved words that were identified in the code.
- **Arithmetic and comparison operators**: Regular expressions were used to recognize operators such as `+`, `-`, `*`,`/`, and comparison operators such as `==`, `!=`, `<=`, `>=`.
- **Literals**: Specific functions were designed to recognize integer type literals, string literals (both single and double), and Boolean literals.
- **Special characters**: Tokens were also handled for special characters such as `(`, `)`, { , }, `;`, among others.

```
keys = r'[\{\}]', which identifies the curly braces { } commonly used
in coding.
parentheses = r'[\(\)]', which identifies parentheses ( ).
brackets = r'[\[\]]', which identifies square brackets [ ].
operators = r'==|!=|<=|>=|<|>', which identifies comparison operators.
multiplication = r'\*', which identifies the multiplication symbol *.
module = r'%', which identifies the module symbol %,
indicating the remainder of a division.
equals = r'=', which recognizes the assignment symbol =.
less = r'-', which identifies the subtraction symbol -.
greater = r'\+', which identifies the addition symbol +.
division = r'/', which identifies the division symbol /.
special = r'[@#$%^&_¿?], which identifies special characters that
may be used in code, everyday expressions, or other contexts.
```

Also, reserved words were added to the tokens using the expression: tokens += list(reserved_words.values()), where reserved_words is a dictionary containing various words such as: if, while, else, for, break, etc.
All whitespaces and line breaks are ignored using the expression: t_ignore = " \t" ."

One of the functions that is utilized is `t_number`, which ascertains whether the input is a number. The message `"Number value too large %s" % t.value` is returned if the number is greater than the capacity of a `int` data type; `t.value` contains the value that was recorded at the time. Here, `r'\d+'` is the regular expression that is employed.

The `t_ID` function has two objectives: first, it identifies reserved words and labels them as "keyword"; second, if it finds a word that is not in the reserved words list, it categorizes it as an "id". The `t_newline` function detects one or more new lines in the input. The `t_error` function is triggered when an unrecognized character is entered, returning "`Illegal character %s % t.value[0]`" to indicate that the character cannot be classified.

These functions aim to identify specific tokens (those that match certain regular expressions) and provide the correct classification name for each one.

A lexical analyzer is built with: `lexer = lex.lex()`. In other words, this creates a lexical analyzer based on the rules previously defined in the code with the tokens, reserved words, and functions.

To use the created lexical analyzer, examples are assigned to variables and then passed to the lexer as follows: `lexer.input(example_input)`. Thanks to the PLY library, the input string is divided and classified into the tokens that were previously defined.

Finally, an infinite loop is created to extract and display the tokens generated by the lexical analyzer `lexer` using the line: `token = lexer.token()`, until there are no more tokens available in the given string, which is achieved with: `if not token: break`. This allows the program to visually display the classification after the code has run.

For testing, several inputs containing code snippets with different operators, reserved words, and literals were provided. The outputs were the tokens corresponding to each element identified in the code. The parser ignored gaps and line breaks, allowing for continuous and efficient reading of the source code.

4 Results

Below are screenshots of the tests performed with the lexical analyzer:

4.1 Test Input

```
int main(){
    julian09 = "julian";
```

```
    for(i = 0; i < 10; i++){
        print("hola");
    }
}
```

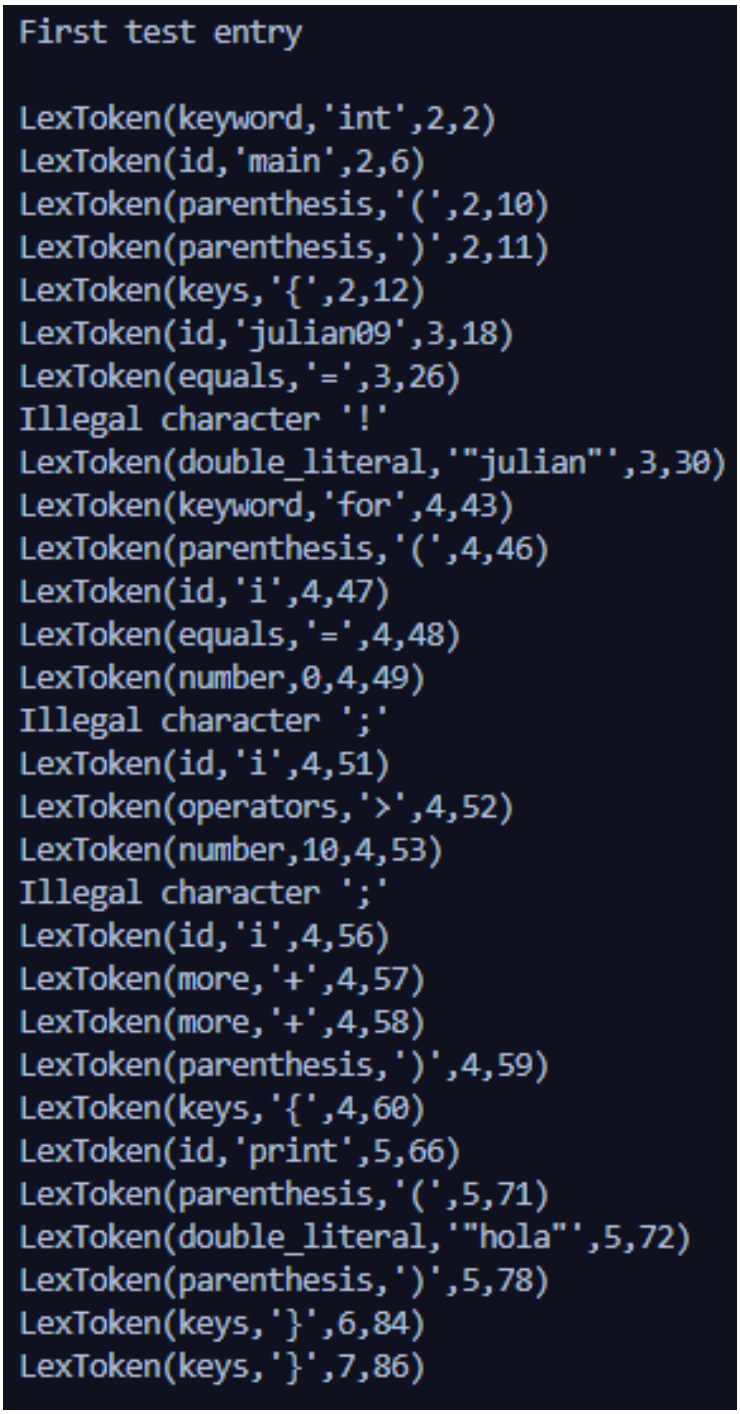

Figure 3: First Test Output.

A section of a string that simulates a C function is defined by the code, which subsequently sends it to a lexical analyzer (lexer). This text is processed by the lexer, which separates it into tokens that represent various code elements including variables, operators, and keywords. It also highlights portions of the code that the regular expressions failed to capture. Ultimately, the algorithm iterates through the created tokens, displaying each one individually until there are no more tokens, enabling us to observe the core lexical components of the input text.

4.2 Second Test Entry

```
Este es un ejemplo de segunda entrada 2
```

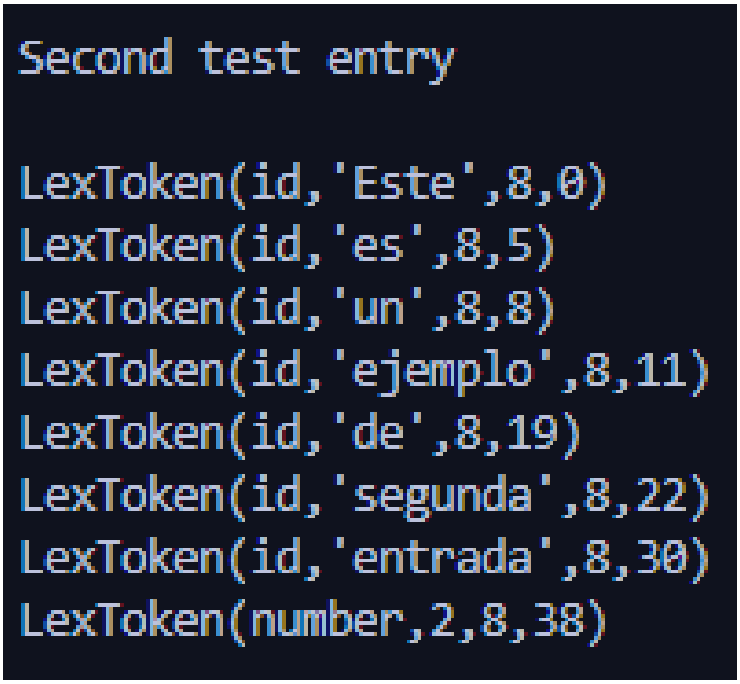

Figure 4: Second Test Output.

This code defines a collection of tokens that the input string's lexical analyzer will be able to identify. Arithmetic and relational operators (`t_operators`, `t_equals`, `t_less`, `t_more`, `t_multiplication`, `t_division`, `t_module`), brackets (`t_brackets`), parentheses (`t_parenthesis`), braces (`t_keys`), and special characters (`t_special`) are among the essential symbols included in these tokens. The lexer can also recognize text literals, double and single (`t_double_literal` and `t_simple_literal`), and integers (`t_number`). The lexer creates tokens for

each of these components in the input string through this method, printing each token one at a time until there are none left.

4.3 Third Test Entry

`10+84=94`

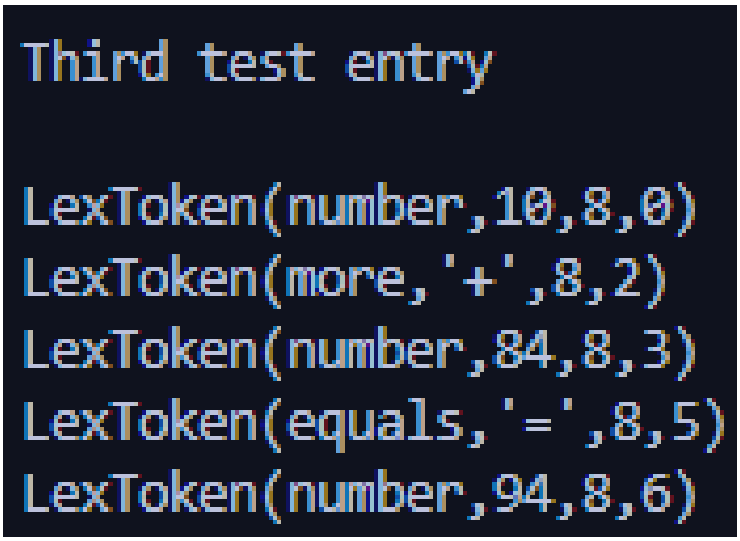

Figure 5: Third Test Output.

The lexical analyzer in this code receives the string `ejemplo_entrada3` and performs a basic arithmetic operation (`"10+84=94"`). In order to analyze this string, the lexer will look for previously defined tokens, including the equals sign (`t_equals`), the plus sign (`t_more`), and numbers (`t_number`). Tokens for the integers 10, 84, and 94, the plus operator, and the = sign will be created for each of these items during the iteration. Until the lexer has processed the full string and no more tokens are available, each token will be displayed on the screen.

5 Conclusions

The project of implementing a lexical analyzer has proven to be an essential tool for understanding the process of compiling a programming language. Correctly identifying tokens is the first step towards generating executable code. The results obtained validate that regular expressions designed for tokens are capable of handling a wide variety of inputs, reflecting a good understanding of the theoretical concepts applied. It is important to note that the use of PLY considerably simplified the task of implementing a lexical analyzer, since

it allows the management of token recognition rules in a structured way. This project reaffirms the relevance of understanding how compilers work and how programming languages process information from its initial stage.

6 Bibliography

R. Sheldon, "compiler," WhatIs, 2022. https://www.techtarget.com/whatis/definition/compiler (accessed Sep. 08, 2024).

GeeksforGeeks, "Introduction of Lexical Analysis," GeeksforGeeks, Jul. 13, 2015. https://www.geeksforgeeks.org/introduction-of-lexical-analysis/ (accessed Sep. 08, 2024).

GeeksforGeeks, "Introduction of Lexical Analysis," GeeksforGeeks, Jul. 13, 2015. https://www.geeksforgeeks.org/introduction-of-lexical-analysis/ (accessed Sep. 10, 2024).

GeeksforGeeks, "Introduction to Syntax Analysis in Compiler Design," GeeksforGeeks, Sep. 22, 2015. https://www.geeksforgeeks.org/introduction-to-syntax-analysis-in-compiler-design/ (accessed Sep. 10, 2024).