Lexer

Enrique Calderón
Luis Salazar
Hansel Tepal
Luis Ugartechea
Universidad Nacional Autónoma de México
Facultad de Ingeniería
March 18, 2025

Introduction

Our lexical analyzer follows the KISS principle, as you would expect for a software implementation. As we know the process is essential for the compiler to work, we need to take into consideration that this should reduce future problems during the next steps of analysis.
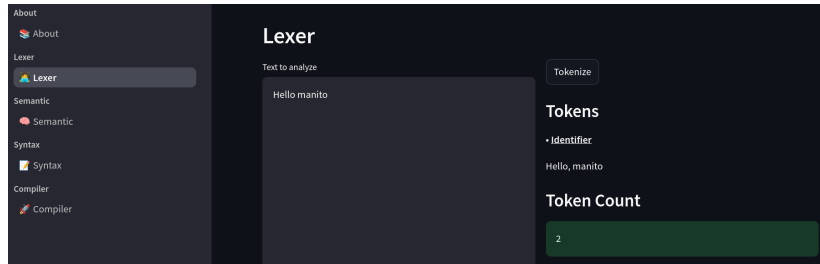

Figure 1: Our lexer :)

Rules for the lexer

In order to design and implement a lexical analyzer we should be able to identify tokens, which are the smallest unit of a program. Each token most follow a grammatical rule depending on his type, for the output we will classify the tokes and print the total amount for the lexem input.

Figure 2: Block diagram

Implementation logic

Based on the rules previously defined for the lexer we decided to classify the tokens in the following categories:

- **Keywords**: Reserved words that have a special meaning in the language.
- **Identifiers**: A sequence of characters that represent a name.
- **Operators**: Symbols that represent an operation.
- **Constants**: A value that does not change during the execution of a program.
- **Punctuation**: Symbols that are used to separate or group expressions.
- **Literals**: A sequence of characters that represent a value.
- **Comments**: A sequence of characters that are not interpreted by the compiler.

Technologies

For this project we used the following libraries:

- ► re
- ► unittest
- ► streamlit
- ► colecctions

Main functions

The most important usde library is the re library which allow us to design patterns using regular expression and also search those patterns in a string.

- ▶ .sub(): This function is for replacing the matches in a string of a given pattern with an specific string
- ▶ .findall(): This function returns a list of strings of all the matched patterns inside a string
- ▶ .replace(): Replaces all the matched string with anote string.

Internal work re

This library works with patterns using a finite state machine, ensuring an liner complex time for the search, which is an optimal complexityl.

For the work, the sre_parser.py file tokenizes all the regular expression, expanding all the special characters, sequences and symbols into a sre (simple regular expression).Then the sre_compile.py turns this sre expressions into bytecode and finally the _sre.c file implements the machine.

Order matters

The code was implemented in python using the `re` library to match the tokens with the regular expressions.

We faced some problems related to the order of the regular expressions, so we decided to follow the next order:

1. **Comments**: Start with the comments because they are the easiest to identify.
2. **Literals**: Otherwise they could be confused with the keywords or identifiers.
3. **Keywords**: The most important part of the code.
4. **Identifiers**: Most common part of the code.
5. **Constants**: If they are not identified as identifiers.
6. **Operators**: The last part of the code that is not alphanumeric and mean something.
7. **Punctuation**: The organization of the code.

Code implementation

To maintain the code clean we decided to use OOP, so we created a class for the lexer with methods such as:

- ▶ `tokenize`: To classify the tokens. It returns a dictionary with the tokens of the input.
- ▶ `token_count`: To count the total amount of tokens. It returns the total amount of tokens.

Implementation APP

The implementation will be divided into the following steps:

1. Read the input string
2. Tokenize the input
3. Classify the tokens
4. Print the total amount of tokens

Brief Example for the lexical analyzer

```
int main() {
    // This is a comment
    if(a == b) {
        print("Hello, World!");
    } else {
        print('Goodbye, World!');
    }
    int x = 10;
    int y = 20;
    int z = x + y;
    if(x >< y && x < y || x > y) {
        print("Complex condition");
    }
    return;
}
```

Results and Conclusions

► Our simple implementation clarifies lexical analysis in a compiler, making it ideal for lightweight and efficient use. The code is easy to understand and maintain, and it can be easily adapted to other programming languages.

► For now, the project is still in the testing phase, as work is underway on other phases to create a more comprehensive compiler. We are confident that our implementation will be successful and that it will be a valuable tool for future projects.