Custom Lexical Analyzer for the Python Language

A course project report

By

Haripreeth Dwarakanath Avarur [RA2011003010011]
Gajulapalli Naga Vyshnavi [RA2011003010049]

Under the guidance of
Dr. K. Vijaya
(Associate Professor, Department of Computing Technologies)

In partial fulfillment for the course of

18CSC304J - Compiler Design

in the department of Computing Technologies,
School of Computing,
Faculty of Engineering and Technology,
For the degree of
Bachelor of Technology in Computer Science and Engineering

INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be **University** *u/s 3 of UGC Act, 1956)*

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
Kattankulathur, Chengalpattu Taluk, Kanchipuram District, Tamil Nadu, India - 603203

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this project report titled "Custom Lexical Analyzer for the Python Language" is the bonafide work of "Haripreeth Dwarakanath Avarur", and "Gajulapalli Naga Vyshnavi", who carried out the project work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

SIGNATURE

Dr. K. Vijaya

Associate Professor,

Department of Computing Technologies,

SRM Institute of Science and Technology

TABLE OF CONTENTS

AIM

The aim of our project is to build a customized lexical analyzer for the python language. We imported a file which had been written in the python language and tokenize it. This is made so that we can process it further for symbolic analyzer and further processing for understanding the python code.

ABSTRACT

A lexical analyzer is an integral component of a compiler or interpreter that scans the input source code and breaks it down into a sequence of tokens. Tokens are the basic building blocks of a programming language, including keywords, identifiers, strings, operators, numbers, and punctuation symbols. The primary goal of a lexical analyzer is to simplify the job of the parser by reducing the complexity of the input source code. This paper discusses the concept of lexical analyzers and their importance in the compilation process. It also outlines the three essential terms in a lexical analyzer, namely lexemes, tokens, and patterns, and the methods used to implement them. Additionally, it highlights the broader applications of lexical analyzers in text processing, such as natural language processing, data mining, and information retrieval.

INTRODUCTION

A lexical analyzer, is a program or a part of a program that reads source code and breaks it down into a series of tokens, which are the basic building blocks of a programming language. It is also named as a lexer or a scanner.

The task of a lexical analyzer is to scan the input source code and identify individual tokens, such as keywords, identifiers, string, operators, number and punctuation symbols. It then outputs these tokens to the parser, which uses them to build a syntax tree and ultimately execute the program.

Example:

- Keywords:
  if, else, for, while, and, or, not, def, class, import, pass

- Identifier:
  my_variable, myFunction, MyClass

- Strings:
  "Hello, world!", 'This is a string.'

- Operators:
  - +, -, *, /, ==, !=, <, >, <=, >=, and, or, not

- Numbers:
  49,11,-7,203

- Punctuation:
  (,), {, }, [,], :, ;, ., ,

The main purpose of a lexical analyzer is to simplify the job of the parser by reducing the complexity of the input source code. By breaking it down into a series of tokens, the parser can more easily identify and interpret the structure of the program, and execute it correctly.

It is also an essential component of the compiler or interpreter of a programming language and It is usually the first stage of the compilation process and acts as a front-end to the compiler.

Lexical analyzers are often implemented using regular expressions, finite automata, or other formal methods. In addition to programming languages, lexical analyzers are used in other applications that require text processing, such as natural language processing, data mining, and information retrieval.

Lexical analyzer contains three important terms:

1. Lexemes:
   It is the smallest unit of meaning in a programming language represented by a sequence of characters in the source code.

2. Tokens:
   It is a sequence of characters representing a single unit of syntax in a programming language, identified during lexical analysis.

3. Patterns:
   It is a set of rules or regular expressions used to match and identify lexemes in the source code.

IMPLEMENTATION:

The given code is a C++ program that reads a file named "input.txt" and performs lexical analysis on its content.
At the beginning of the code, the necessary header files are included, such as iostream, fstream, string, and regex.

The program defines several functions for identifying different types of tokens:

- isKeyword: This function takes a string as input and checks if it is a keyword in the Python programming language. The list of keywords is defined in the function itself.
- isIdentifier: This function takes a string as input and checks if it matches the pattern of a valid Python identifier. The pattern is defined using a regular expression that matches a string starting with an underscore or a letter, followed by any number of underscores, letters, or digits.

- isString: This function takes a string as input and checks if it matches the pattern of a string literal in Python. The pattern is defined using a regular expression that matches a string starting and ending with double quotes, and containing any number of characters in between.

- isNumber: This function takes a string as input and checks if it matches the pattern of a numeric literal in Python. The pattern is defined using a regular expression that matches any number of digits.

- isOperator: This function takes a character as input and checks if it is a valid operator in Python. The list of operators is defined in the function itself.

- isPunctuation: This function takes a character as input and checks if it is a valid punctuation symbol in Python. The list of punctuation symbols is defined in the function itself.

The main function is where the actual lexical analysis is performed. It first opens the input file named "input.txt" using an ifstream object. It then reads each line of the file using the getline function, which stores the line in the line variable.

The for loop then iterates through each character in the line variable. If the character is a space, newline, tab, operator, or punctuation symbol, the program checks if the word variable is empty or not. If it is not empty, it checks if the word variable matches any of the recognized token patterns using the functions defined earlier. If it does, it prints out the corresponding message indicating the type of token. If it does not match any recognized pattern, it prints out a message saying that it is not a recognized token.

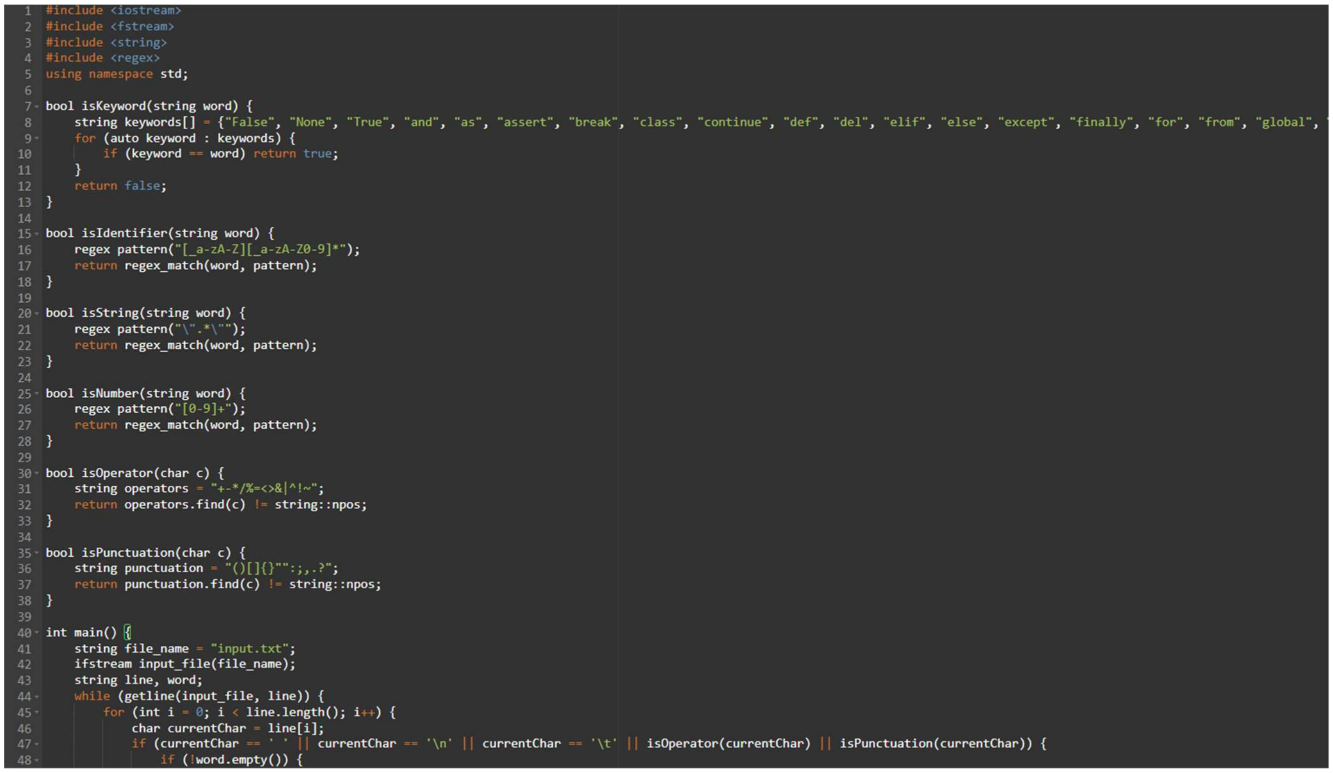
If the character is an operator or a punctuation symbol, the program prints out a message indicating the type of symbol.

If the character is not a space, newline, tab, operator, or punctuation symbol, it adds the character to the word variable.

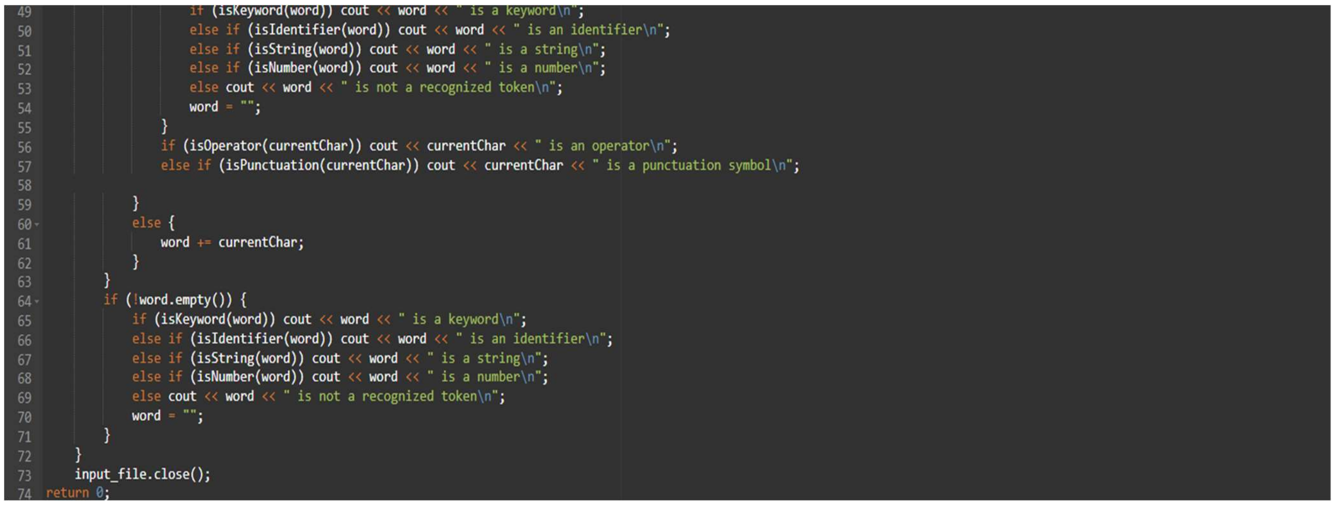
At the end of each line, the program checks if the word variable is empty or not. If it is not empty, it performs the same checks as before and prints out the corresponding message indicating the type of token.

Finally, the main function closes the input file and returns 0 to indicate successful execution of the program.

In summary, this program reads a file and identifies the different types of tokens in its content, such as keywords, identifiers, strings, numbers, operators, and punctuation symbols, by using regular expressions and predefined lists. It demonstrates the basic concepts of lexical analysis, which is a crucial component of the compilation process in programming languages.

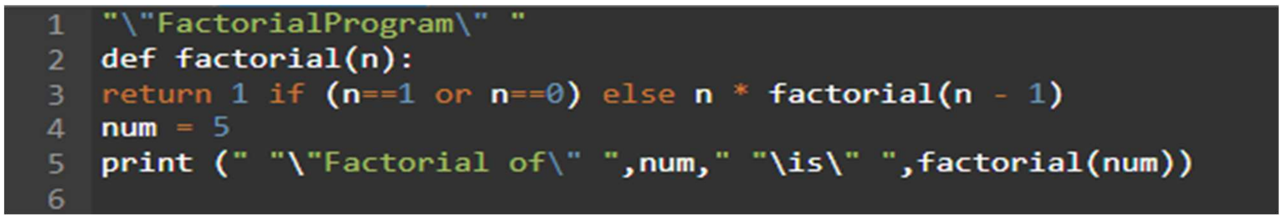


```
#include <iostream>
#include <fstream>
#include <string>
#include <regex>
using namespace std;

bool isKeyword(string word) {
    string keywords[] = {"False", "None", "True", "and", "as", "assert", "break", "class", "continue", "def", "del", "elif", "else", "except", "finally", "for", "from", "global", 
    for (auto keyword : keywords) {
        if (keyword == word) return true;
    }
    return false;
}

bool isIdentifier(string word) {
    regex pattern("[_a-zA-Z][_a-zA-Z0-9]*");
    return regex_match(word, pattern);
}

bool isString(string word) {
    regex pattern("\".*\"");
    return regex_match(word, pattern);
}

bool isNumber(string word) {
    regex pattern("[0-9]+");
    return regex_match(word, pattern);
}

bool isOperator(char c) {
    string operators = "+-*/%=<>&|^!~";
    return operators.find(c) != string::npos;
}

bool isPunctuation(char c) {
    string punctuation = "()[]{}"":;,.?";
    return punctuation.find(c) != string::npos;
}

int main() {
    string file_name = "input.txt";
    ifstream input_file(file_name);
    string line, word;
    while (getline(input_file, line)) {
        for (int i = 0; i < line.length(); i++) {
            char currentChar = line[i];
            if (currentChar == ' ' || currentChar == '\n' || currentChar == '\t' || isOperator(currentChar) || isPunctuation(currentChar)) {
                if (!word.empty()) {
```



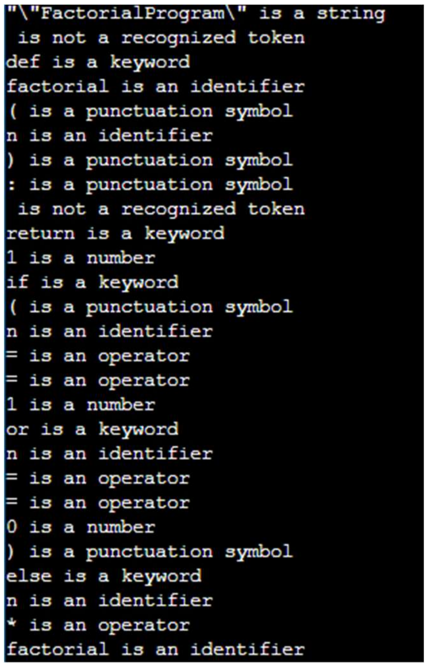
```
                    if (isKeyword(word)) cout << word << " is a keyword\n";
                    else if (isIdentifier(word)) cout << word << " is an identifier\n";
                    else if (isString(word)) cout << word << " is a string\n";
                    else if (isNumber(word)) cout << word << " is a number\n";
                    else cout << word << " is not a recognized token\n";
                    word = "";
                }
                if (isOperator(currentChar)) cout << currentChar << " is an operator\n";
                else if (isPunctuation(currentChar)) cout << currentChar << " is a punctuation symbol\n";

            }
            else {
                word += currentChar;
            }
        }
        if (!word.empty()) {
            if (isKeyword(word)) cout << word << " is a keyword\n";
            else if (isIdentifier(word)) cout << word << " is an identifier\n";
            else if (isString(word)) cout << word << " is a string\n";
            else if (isNumber(word)) cout << word << " is a number\n";
            else cout << word << " is not a recognized token\n";
            word = "";
        }
    }
    input_file.close();
return 0;
```

The input.txt file is an external file which has been imported to the above program. The content of the input.txt is as follows:

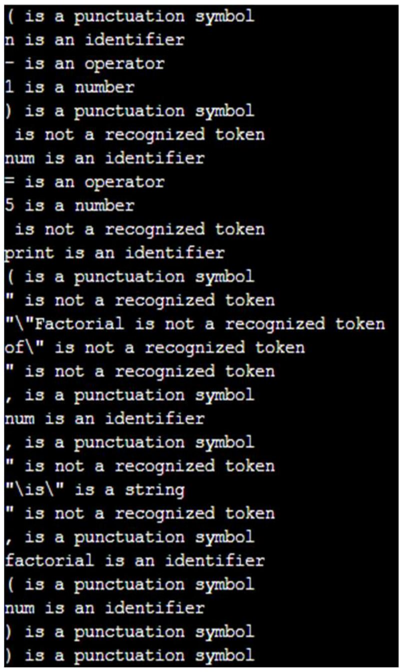


```
"\"FactorialProgram\" "
def factorial(n):
return 1 if (n==1 or n==0) else n * factorial(n - 1)
num = 5
print (" "\"Factorial of\" ",num," "\is\" ",factorial(num))

```

<u>Output:</u>



```
"\"FactorialProgram\" is a string
 is not a recognized token
def is a keyword
factorial is an identifier
( is a punctuation symbol
n is an identifier
) is a punctuation symbol
: is a punctuation symbol
 is not a recognized token
return is a keyword
1 is a number
if is a keyword
( is a punctuation symbol
n is an identifier
= is an operator
= is an operator
1 is a number
or is a keyword
n is an identifier
= is an operator
= is an operator
0 is a number
) is a punctuation symbol
else is a keyword
n is an identifier
* is an operator
factorial is an identifier
```



```
( is a punctuation symbol
n is an identifier
- is an operator
1 is a number
) is a punctuation symbol
 is not a recognized token
num is an identifier
= is an operator
5 is a number
 is not a recognized token
print is an identifier
( is a punctuation symbol
" is not a recognized token
"\"Factorial is not a recognized token
of\" is not a recognized token
" is not a recognized token
, is a punctuation symbol
num is an identifier
, is a punctuation symbol
" is not a recognized token
"\is\" is a string
" is not a recognized token
, is a punctuation symbol
factorial is an identifier
( is a punctuation symbol
num is an identifier
) is a punctuation symbol
) is a punctuation symbol
```

EXPLANATION:

Input preprocessing: This stage involves cleaning up the input text and preparing it for lexical analysis. This may include removing comments, whitespace, and other non-essential characters from the input text.

Tokenization: This is the process of breaking the input text into a sequence of tokens. This is usually done by matching the characters in the input text against a set of patterns or regular expressions that define the different types of tokens.

Token classification: In this stage, the lexer determines the type of each token. For example, in a programming language, the lexer might classify keywords, identifiers, operators, and punctuation symbols as separate token types.

Token validation: In this stage, the lexer checks that each token is valid according to the rules of the programming language. For example, it might check that a variable name is a valid identifier, or that an operator has the correct syntax.

Output generation: In this final stage, the lexer generates the output of the lexical analysis process, which is typically a list of tokens. This list of tokens can then be passed to the next stage of compilation or interpretation.

ADVANTAGES AND DISADVANTAGES:

Advantages of a lexical analyzer:

1. Simplifies the job of a parser: The main advantage of a lexical analyzer is that it simplifies the job of a parser by breaking down the input source code into a sequence of tokens. This makes it easier for the parser to identify and interpret the structure of the program, and execute it correctly.

2. Reduces the complexity of the input source code: A lexical analyzer reduces the complexity of the input source code by identifying individual tokens such as keywords, identifiers, strings, operators, numbers, and punctuation symbols. This simplifies the task of the parser and makes it easier for developers to write and debug code.

3. Improves the performance of the compiler or interpreter: A lexical analyzer improves the performance of the compiler or interpreter by eliminating redundant characters and whitespace from the input source code. This reduces the size of the input and speeds up the compilation process.

4. Makes code more readable: By breaking down the input source code into a sequence of tokens, a lexical analyzer makes code more readable and easier to understand. This is particularly important for large codebases with many contributors, as it ensures that everyone is using a consistent set of coding conventions.

5. Enables syntax highlighting: A lexical analyzer can be used to enable syntax highlighting in text editors and IDEs. Syntax highlighting makes it easier for developers to read and understand code by highlighting different types of tokens with different colors.

Issues/Disadvantages of a lexical analyzer:

1. Ambiguity: A programming language can have some constructs that are ambiguous and difficult to disambiguate during lexical analysis. For example, the meaning of a symbol like "+" may depend on the context in which it appears.

2. Efficiency: The performance of a lexical analyzer can be a concern for large programs or applications with high performance requirements. The lexer must scan the input source code character by character and generate a sequence of tokens, which can be a computationally intensive task.

3. Error handling: The lexer must be able to detect syntax errors and provide meaningful error messages to the user. Error handling can be complex, especially when the source code contains multiple errors or when the errors occur in complex constructs.

4. Parsing conflicts: The lexer and parser must work together seamlessly to ensure that the syntax tree is constructed correctly. In some cases, the lexer and parser may have conflicts that need to be resolved.

5. Language extensions: When a programming language is extended or modified, the lexer must be updated to recognize the new constructs and generate the corresponding tokens.

6. Code size: The size of the lexical analyzer code can be significant, especially when the language being analyzed is complex or has many features.

APPLICATIONS:

1. Tokenizing source code: The primary function of a lexical analyzer is to break down the source code into a stream of tokens, which are then used by the parser to construct the syntax tree.

2. Error detection and recovery: The lexical analyzer can also identify syntax errors and provide error messages to the user, making it easier to locate and fix errors in the source code. Additionally, some advanced lexers can perform error recovery by skipping over invalid tokens or trying to correct them.

3. Code highlighting: A lexical analyzer can be used to highlight different elements of the source code in an IDE or text editor, making it easier for the user to read and navigate the code.

4. Program analysis: The tokens produced by the lexer can be used for various program analysis tasks, such as detecting and reporting unused variables or dead code.

5. Optimization: The lexer can also be used to perform some optimization tasks, such as reducing the number of memory allocations by reusing tokens that occur multiple times in the source code.

CONCLUSIONS:

In conclusion, a lexical analyzer is a fundamental component of a compiler or interpreter that performs the task of analyzing the source code of a program and breaking it down into a sequence of tokens. The lexer scans the input source code character by character, groups them into lexemes, maps them to tokens defined by the language grammar, classifies the tokens, handles errors, and produces a sequence of tokens that represent the program.

The importance of a well-designed and optimized lexical analyzer cannot be overstated. It plays a crucial role in the accuracy and performance of a compiler or interpreter, leading to faster program execution and better user experience. However, designing and implementing a lexer can be challenging, requiring careful consideration of factors such as regular expressions, error handling, performance, and scalability.

Therefore, it is essential to understand the role and function of a lexical analyzer and the key factors that influence its design and implementation. A successful lexer must be accurate, efficient, and maintainable, while also providing robust error handling and compatibility with language extensions. Ultimately, a well-designed lexical analyzer can have a significant impact on the overall quality and functionality of a compiler or interpreter.

RESULTS:

We successfully implemented a lexical analyzer for the python language, where, we imported a text file and tokenized each of the parameters. Each parameter has been classified.

REFERENCES:

- https://byjus.com/gate/lexical-analysis/#:~:text=on%20Lexical%20Analysis-,What%20is%20Lexical%20Analysis%3F,whitespace%20in%20the%20source%20code.
- https://www.geeksforgeeks.org/introduction-of-lexical-analysis/
- https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm
- https://www.javatpoint.com/the-phases-of-a-compiler-lexical-analysis
- https://en.wikipedia.org/wiki/Lexical_analysis
- https://www.codingninjas.com/codestudio/library/lexical-analysis-in-compiler-design
- https://www.youtube.com/watch?v=4nx8LEGy9kI