Tokenization of C++ Code: A Python Approach Using Regular Expressions for Lexical Analysis

Saad Saleem-2023623, Rayyan Salman-2023601

Faculty of Computer Sciences, Ghulam Ishaq Khan Institute of Engineering Sciences and Technology
Topi, Swabi, Pakistan

Abstract— **This paper describes the manual development of a lexical analyser for a defined subset of the C++ programming language. The project leverages Python's built-in re module, which facilitates the use of regular expressions for token recognition. The primary goal is to explore the principles of lexical analysis, particularly the application of regular expressions in transforming a C++ source code file into a stream of tokens. The report covers the design of the lexical analyser, token types recognized, challenges encountered during the implementation, and the key lessons learned throughout the process.**

Keywords— **Lexical Analysis, Tokenization, Python, Regular Expressions, Compiler Design, Formal Languages, Automata Theory, Manual Implementation.**

I. Introduction

Lexical analysis is a fundamental phase of the compilation process, transforming a sequence of characters into a stream of tokens. Tokens are the smallest units of meaningful data in a program, often representing keywords, identifiers, literals, and symbols. This phase is crucial as it lays the groundwork for the subsequent stages of compilation, including parsing and code generation.

In this project, a lexical analyser for a subset of the C++ language is developed using Python's re module. The goal of the project is to manually implement a simple lexer that can process basic C++ syntax and recognize common tokens such as keywords, operators, and identifiers.

II. Background and Motivation

An In modern compiler design, lexical analysis is often handled by tools such as Flex, which generates code for tokenizing input based on regular expression patterns. While these tools are efficient, the manual implementation of a lexical analyser provides a deeper understanding of how tokenization works at a fundamental level. By using Python's re module, the project also demonstrates how high-level programming languages can be employed in the development of a lexical analyser without the need for specialized tools.

The motivation for this project is twofold: first, to implement a lexical analyser manually for educational purposes, and second, to explore how Python's regular expressions can be applied to real-world problems.

Beyond educational curiosity, this project also serves as the **semester project for the course CS-224: Formal Languages and Automata Theory**. It allows practical application of theoretical concepts covered in the curriculum, such as regular languages and expressions—both of which underpin the behaviour of lexical analysers.

III. Design and Implementation Approach

Usage Instructions:
python lexer.py <inputfile.cpp> <outputfile.txt>

The lexical analyser, referred to as **PyLexer**, is designed to process source code written in a subset of C++ containing basic syntax, such as variables, keywords, and operators. The core of the implementation uses regular expressions to match predefined patterns in the input code, which are then classified as distinct token types.

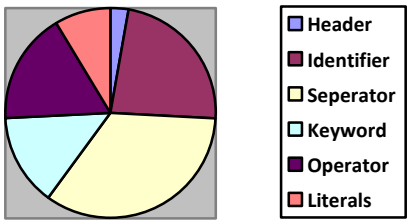
A. Token Types

The **PyLexer** recognizes a variety of token types, including:

1. ***Keywords****: Reserved words in C++ such as* `int`*,* `return`*,* `while`*, etc.*
2. ***Identifiers****: Variables, function names, and other user-defined names.*
3. ***Operators****: Symbols such as* `+`*,* `-`*,* `*`*,* `/`*, etc., used for arithmetic or logical operations.*

4. ***Literals***: *Constants such as numeric values (`10`, `3.14`) and string literals (`"Hello"`).*
5. ***Separators***: *Symbols such as commas `,`, semicolons `;`, parentheses `()`, and braces `{}`.*
6. ***Whitespace***: *Spaces, tabs, and newline characters are ignored.*

7. Graph I
Token distribution for a sample C++ Code



For each token type, a corresponding regular expression pattern is defined. For example, the pattern for recognizing an identifier is as follows:
Identifier_pattern = r"[a-zA-Z_][a-zA-Z0-9_]*"

This pattern matches any string that begins with a letter or underscore, followed by any number of alphanumeric characters or underscores.

The pattern for recognizing a number literal is:
Number_pattern = r"\d+(\.\d+)?"
This matches integer and floating-point numbers.

B. Input Processing and Tokenization

The lexical analyser reads and processes the input source code sequentially, line by line, ensuring a clear mapping between tokens and their respective positions in the original file. This approach enhances traceability, which is essential for accurate error reporting and debugging during the compilation process.

The core mechanism behind the tokenization phase is a **greedy algorithm**, which attempts to match the longest valid token at each position in the input stream. This prevents ambiguity and ensures that multi-character tokens such as ==, <=, or != are not mistakenly split into multiple, shorter tokens like = and =. The predefined regular expression patterns for all token types are applied in a prioritized order, with longer and more specific patterns listed before shorter and more general ones.

For each line, the analyser uses a sliding window approach to iterate through the characters. At each step, it checks whether a substring starting at the current position matches any of the regular expression patterns. If a match is found, the matched token is extracted, classified (e.g., as a keyword, operator, identifier, etc.), and recorded along with its value and the corresponding line number.

TABLE II
Sample Regular Expression Patterns in PyLexer

Token Type	Regex Pattern	Example Match
Header	#include\s*<[^>]+>	#include<fstream>
String	"[^"\n]*"	“hello”
Comment	//.*	//this is a comment
White space	\t\r	\t
Separator	[(){}\[\];,]	{

This process involves the following key steps:

1. **Whitespace and Comment Removal**: The analyser first strips out any whitespace and comments. Single-line comments (//) are ignored from the point of occurrence to the end of the line. Multi-line comments (/* ... */) are removed using a non-greedy regular expression that spans across lines.
2. **Token Matching Loop**: Once the line is cleaned, the analyser enters a matching loop. At each iteration, it applies the regular expressions in a prioritized order. If a valid match is found, the matched substring is removed from the line, and the token is appended to a token list with metadata (type, value, line number). If no match is found, an error is raised or logged, indicating an unrecognized token.
3. **Token Storage**: All matched tokens are stored as tuples in the format (token_type, token_value, line_number). This structured format allows downstream components, such as parsers or semantic analysers, to easily process the token stream without having to re-parse or reclassify token information.
4. **Greedy Matching Priority**: A crucial aspect of the tokenization logic is the order of matching. Operators and keywords with more characters are checked before their shorter counterparts to ensure

correct interpretation. For example, == is matched before =, and <= before <.

This approach not only ensures syntactic correctness but also simplifies the integration of the lexer with the later stages of compilation. The tokenization pipeline is modular, allowing for the easy addition of new token types or expansion to support more C++ features.

IV. Challenges and Solution

During the development of the PyLexer lexical analyser, several practical and conceptual challenges arose. These issues not only tested the robustness of the implementation but also provided valuable opportunities to better understand real-world considerations in compiler and toolchain development.

A. Handling Nested Structures

Challenge: Differentiating punctuation from operators in nested constructs like braces and parentheses.

Solution: Regular expressions were ordered to prioritize multi-character symbols, ensuring accurate token recognition.

B. Keyword vs. Identifier Ambiguity

Challenge: Identifiers like int could be misclassified if not distinguished from keywords.

Solution: A keywords list was checked after identifier matches to reclassify to reclassify token

C. Command-Line Usage

Challenge: Initial unfamiliarity with handling inputs through the command line.

Solution: Pythons's sys.argv was used to accept filename, with error messages shown if no argument was given .

D. Missing or Invalid input file

Challenge: Program crashed if input file didn't exist.

Solution: A try-except block was added to catch File-Not-Found-Error and display a clear, user-friendly error.

V. Testing and Evaluation

The lexical analyser was tested extensively using a range of C++ source files, from simple declarations to more complex snippets involving conditionals, loops, and nested functions. Each test case was designed to verify the accuracy, coverage, and robustness of the tokenization process.

A. Functional Testing

Sample C++ programs were created containing common language constructs such as variable declarations, control structures, function definitions, arithmetic expressions, and comments. Each tokenized output was manually verified against expected results to ensure correct classification and ordering.

B. Comparison with Flex

To evaluate performance and correctness, the same test cases were processed using **Flex**, a well-established lexical analyser generator. The output from PyLexer was directly compared with Flex's token stream. Results showed that PyLexer matched Flex in both accuracy and processing efficiency for the defined subset of C++.

While Flex offered broader language support, PyLexer achieved comparable results within its target scope. The test confirmed that for basic C++ syntax, a manually implemented analyser using Python and regular expressions can perform as efficiently as an industrial tool like Flex—especially in educational or prototype environments.

Test Input :

```
#include <string>
string name = "Rayyan" //strong my name in
variable name
```

Expected Output:

```
[
  ("header", "#include <string>"),
  ("keyword", "string"),
  ("identifier", "name"),
  ("operator", "="),
  ("string_literal", "\"Rayyan\""),
  ("comment", "//strong my name in variable name")
]
```

Output :

```
Line 1: Token = #include <string> → Header
Line 2: Token = string      → Keyword
```

Line 2: Token = name → Identifier
Line 2: Token = = → Operator
Line 2: Token = "Rayyan" → String Literal
Line 2: Token = //strong my name in variable name → Comment

VII. Lessons Learned

The manual implementation of a lexical analyser provided valuable insights into the workings of a compiler's front-end. Some key takeaways include:

A. Importance of Regular Expressions

Regular expressions are powerful tools for pattern matching, and their application in lexical analysis is essential for building efficient and scalable analysers.

B. Token Prioritization

The order in which token patterns are matched plays a critical role in ensuring that the longest and most specific tokens are correctly identified.

C. Handling Complex Syntax

Dealing with complex language syntax, such as nested structures and comments, requires careful planning and implementation.

D. Python's re Module

The project significantly improved our understanding of Python's re module. We learned how to define complex patterns using grouping, quantifiers, and greedy/non-greedy matches—critical for correct token extraction.

VIII. Conclusion

This project successfully demonstrated the design and implementation of a manual lexical analyser for a simplified subset of the C++ programming language using Python's regular expression capabilities. The tool effectively tokenizes input source files into meaningful components, matching the accuracy and structure of industry-standard tools like Flex for its targeted language subset.

Beyond the core functionality, the project provided hands-on experience in command-line interfacing, exception handling, pattern matching, and debugging. It reinforced the importance of error resilience and clear user feedback—crucial for real-world compiler tools.

Additionally, the ability to compare performance with Flex validated the reliability of a hand-coded solution. While PyLexer is not intended to replace production-grade lexers, it serves as an excellent educational tool and a foundation for future projects in compiler construction.

IX. Potential Extensions

A. Addition of More C++ Constructs

Extend support to include more C++ keywords and operators to improve language coverage

Acknowledgment

We would like to acknowledge **Professor Charles Severance**, whose publicly available *Python for Everybody* course on Coursera (University of Michigan) provided valuable foundational knowledge of Regular Expressions in Python that aided us during the implementation of this project.

We also express our gratitude to **Sir Sajid Ali**, our instructor for *Formal Languages and Automata Theory*, whose course offered the theoretical background necessary to understand the principles of lexical analysis.