C++ Lexer Analyzer

(Detailed Report)

Course: CS-224 – Theory of Automata

Instructor: Sajid Ali

Team: Muhammad Ammar Saleem 2023378

Raja Bilal Khurram 2023591

Submission Date: 10/05/25

Table of Contents

Introduction

The **C++ Lexer Analyzer** is used to break down our C and C++ source code into manageable pieces. This gives us an insight on how the compiler is understanding what we are writing in the backend. We used Python's Streamlit framework to provide our users with an interactive platform to input, analyze, and compare code snippets. Our project provides breaking down code into tokens, identifying errors, visualizing token statistics, comparing two different codes to check if they are lexically identical or not, and generating parse trees. This project has helped us implement the core concepts of the theory of automata (Complex Regular Expressions) and also help us understand the intricacies of lexical analysis in compiler design.

Project Overview

The primary objective of this project is to understand how we can use the concepts of theory of automata in real world problems like the working of all compilers that form the backbone of all programs and create an interactive lexical analyzer that not only tokenizes C/C++ code but also provides additional functionalities such as error detection, statistical visualization, parse tree generation, and code comparison. We used Streamlit to create an interactive frontend for our project.

Connection to Theory of Automata

Throughout our course we started from the basic question, what are languages? Kept working on and on out way understood Regular Expressions and DFA's. We focused on abstract machines and the problems they can solve. This project is a core example of the application of this course:

- **Finite Automata:** Our lexical analyzer operates similarly to a deterministic finite automaton (DFA), scanning input strings (code) and transitioning between states based on character patterns to identify tokens. If something is alien to the system it stores it in the error table.
- **Regular Expressions:** Token patterns are defined using regular expressions, a fundamental concept in automata theory, to specify the lexical structure of programming languages.

By implementing these concepts practically, the project bridges the gap between theoretical knowledge and real-world application.

Features of the C Lexer Analyzer

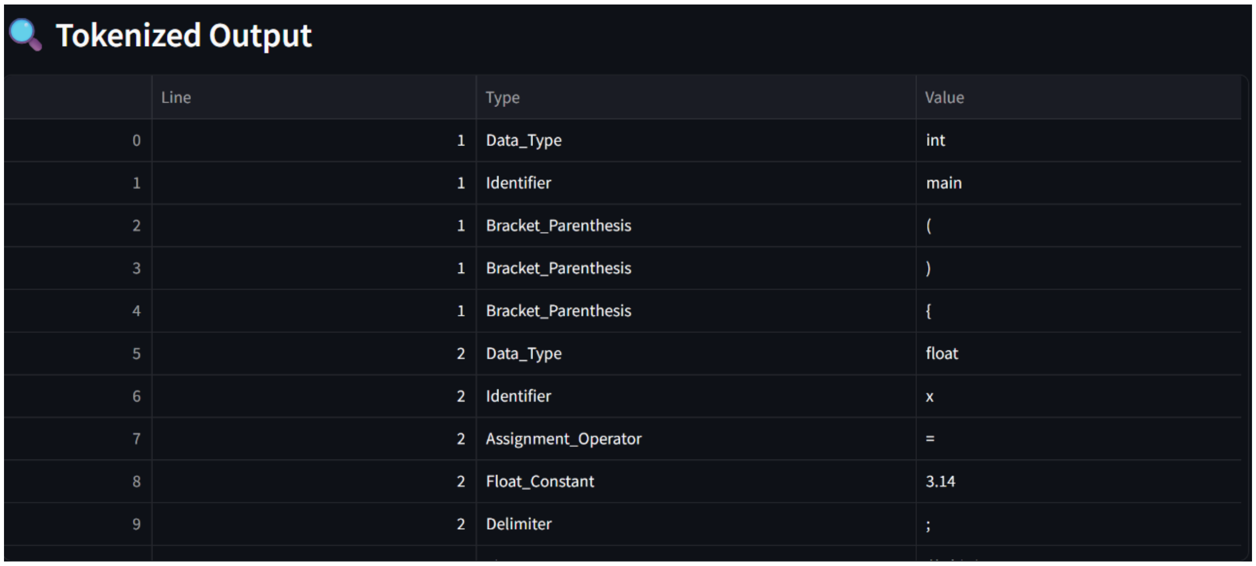
1. Tokenization

Functionality:
The analyzer scans the input C/C++ code and breaks it down into a sequence of tokens. Each token represents a meaningful element of the language, such as keywords, identifiers, operators, literals, and delimiters.

Implementation Details:

- Utilizes Python's re module to define regular expressions for various token types.
- Processes the code line by line, matching patterns to identify tokens.
- Stores tokens along with their line numbers and types for further analysis.

Purpose:
Demonstrates the practical application of regular expressions and finite automata in lexical analysis.



Tokenized Output

	Line	Type	Value
0	1	Data_Type	int
1	1	Identifier	main
2	1	Bracket_Parenthesis	(
3	1	Bracket_Parenthesis	)
4	1	Bracket_Parenthesis	{
5	2	Data_Type	float
6	2	Identifier	x
7	2	Assignment_Operator	=
8	2	Float_Constant	3.14
9	2	Delimiter	;

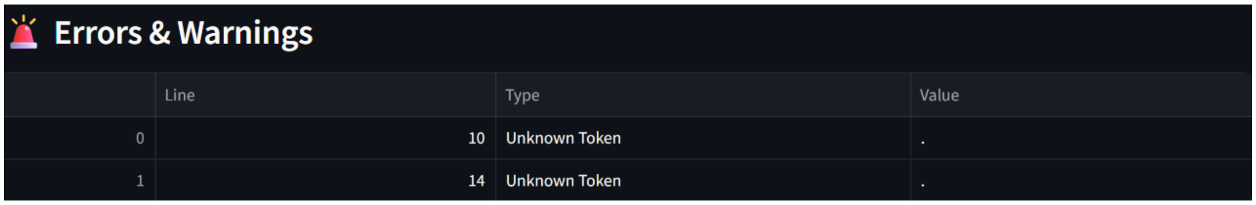
2. Error Detection

Functionality:
Identifies unrecognized or invalid tokens in the source code, aiding in debugging and code correction.

Implementation Details:

- Tokens that do not match any predefined pattern are classified as 'Unknown_Token'.
- Collects these tokens along with their line numbers for user review.

Purpose:
Highlights the importance of accurate token definitions and the role of error handling in compiler design.



Errors & Warnings

	Line	Type	Value
0	10	Unknown Token	.
1	14	Unknown Token	.

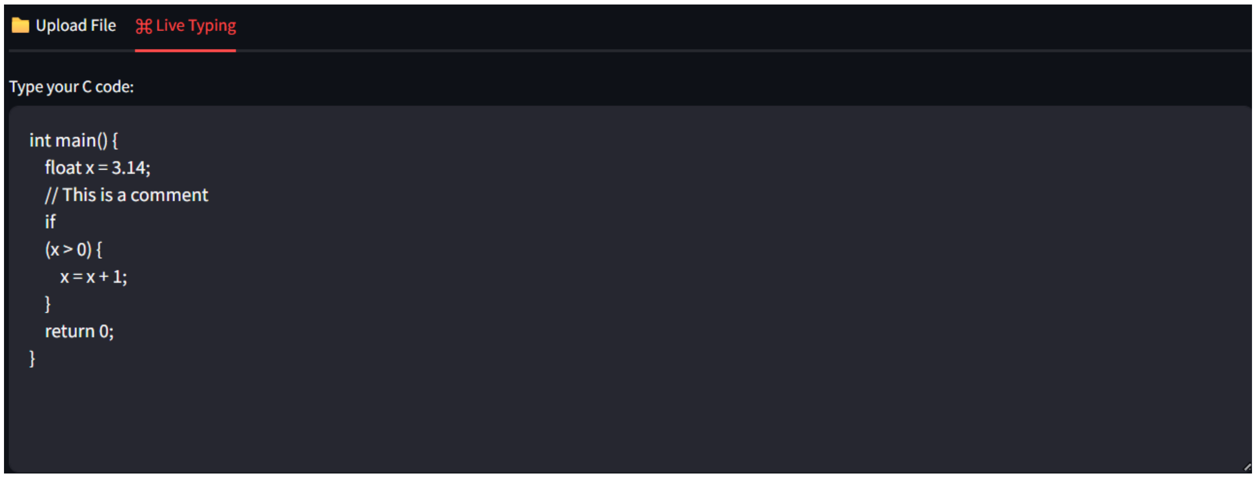
3. Live Coding Interface

Functionality:
Provides a real-time code editor where users can type or paste C/C++ code and instantly view the lexical analysis results.

Implementation Details:

- Employs Streamlit's st.text_area for code input.
- Automatically updates tokenization results upon code changes.

Purpose:
Facilitates immediate feedback, reinforcing the understanding of how code modifications affect lexical structure.



Type your C code:

```
int main() {
  float x = 3.14;
  // This is a comment
  if
  (x > 0) {
    x = x + 1;
  }
  return 0;
}
```

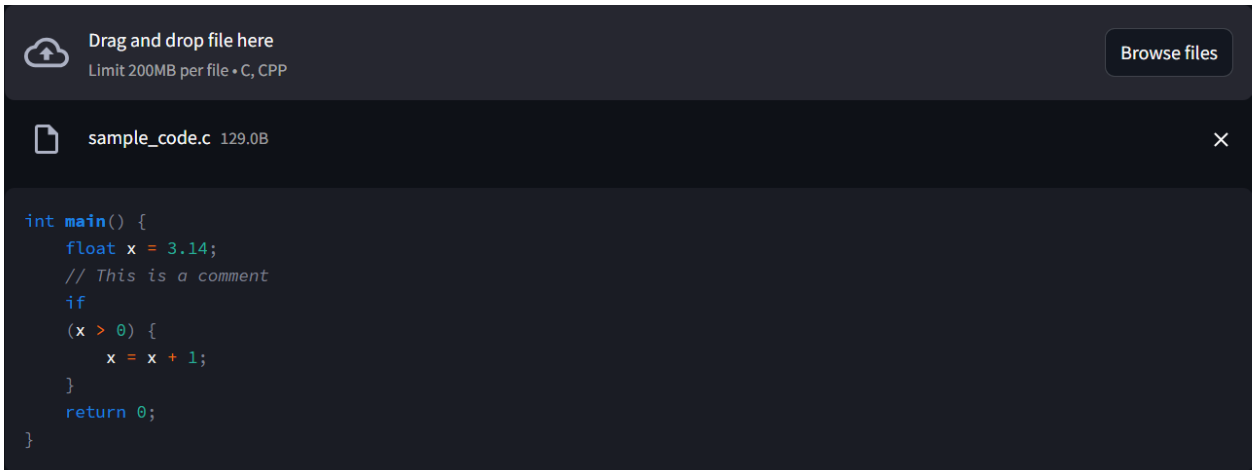
4. File Upload and Analysis

Functionality:
Allows users to upload .c or .cpp files for analysis.

Implementation Details:

- Uses Streamlit's st.file_uploader to accept file inputs.
- Reads and decodes the file content for tokenization.

Purpose:
Demonstrates the scalability of lexical analysis tools to handle complete source files.



```
int main() {
    float x = 3.14;
    // This is a comment
    if
    (x > 0) {
        x = x + 1;
    }
    return 0;
}
```

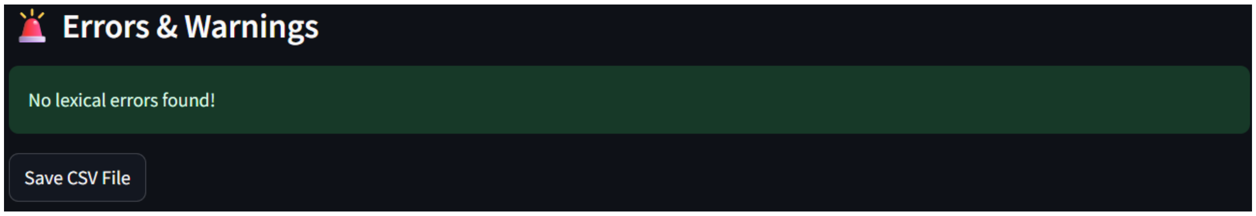
5. Output Download Option

Functionality:
Allows users to save the tokenized output of their C/C++ code analysis as a CSV file for external use or further examination.

Implementation Details:

- After tokenization, the output is displayed in a table using Streamlit's st.dataframe.
- A "Save CSV File" button allows users to export the displayed tokens to a .csv file.

Purpose:
Provides users with a way to preserve and share lexical analysis results, facilitating further processing, report writing, or debugging across different platforms.


🚨 Errors & Warnings

No lexical errors found!

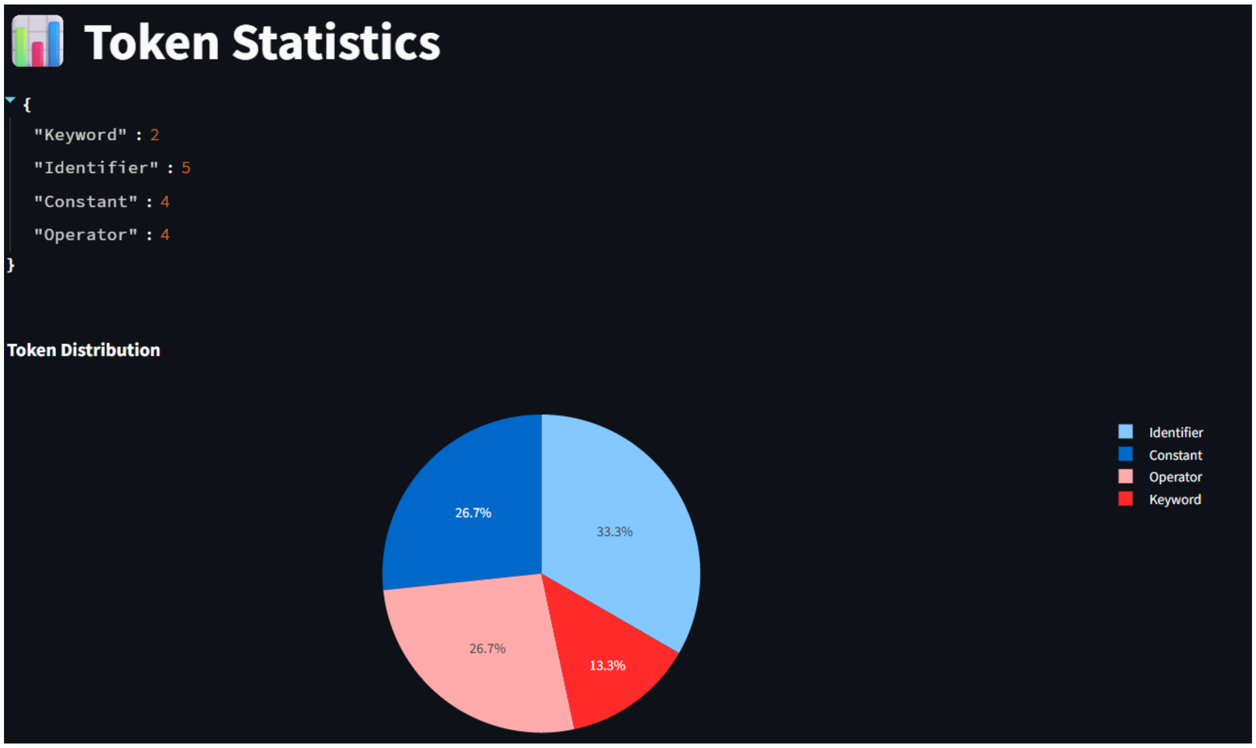
6. Token Statistics Visualization

Functionality:
Displays a statistical breakdown of token types present in the analyzed code.

Implementation Details:

- Counts occurrences of each token type.
- Utilizes Plotly to generate interactive pie charts representing the token distribution.

Purpose:
Provides insights into code composition, aiding in understanding programming patterns and structures.


📊 Token Statistics

```
{
  "Keyword" : 2
  "Identifier" : 5
  "Constant" : 4
  "Operator" : 4
}
```

Token Distribution

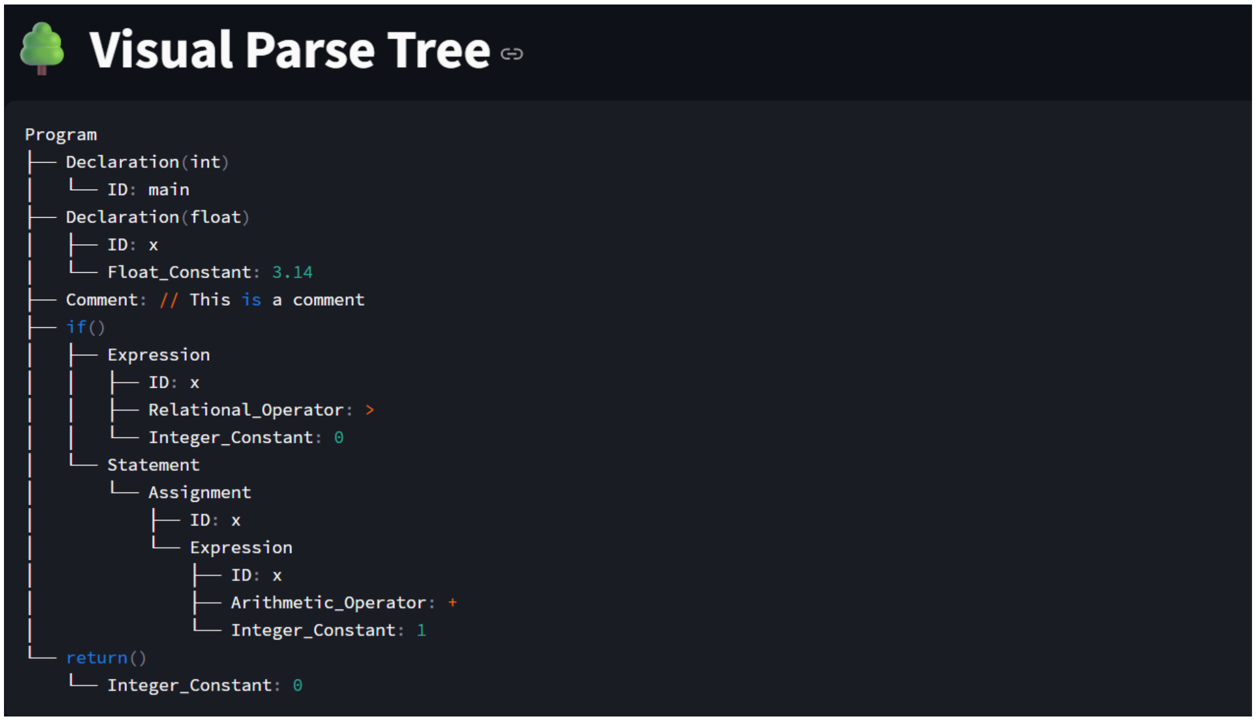
7. Complex Parse Tree Generation

Functionality:
Constructs a hierarchical parse tree representing the syntactic structure of the code.

Implementation Details:

- Implements a simplified parser that recognizes basic C constructs such as declarations, assignments, and control statements.
- Uses the anytree library to build and render the parse tree.

Purpose:
Visualizes the relationship between tokens and their syntactic roles, reinforcing parsing concepts.



Visual Parse Tree

```
Program
├── Declaration(int)
│   └── ID: main
├── Declaration(float)
│   ├── ID: x
│   └── Float_Constant: 3.14
├── Comment: // This is a comment
├── if()
│   ├── Expression
│   │   ├── ID: x
│   │   ├── Relational_Operator: >
│   │   └── Integer_Constant: 0
│   └── Statement
│       └── Assignment
│           ├── ID: x
│           └── Expression
│               ├── ID: x
│               ├── Arithmetic_Operator: +
│               └── Integer_Constant: 1
└── return()
    └── Integer_Constant: 0
```

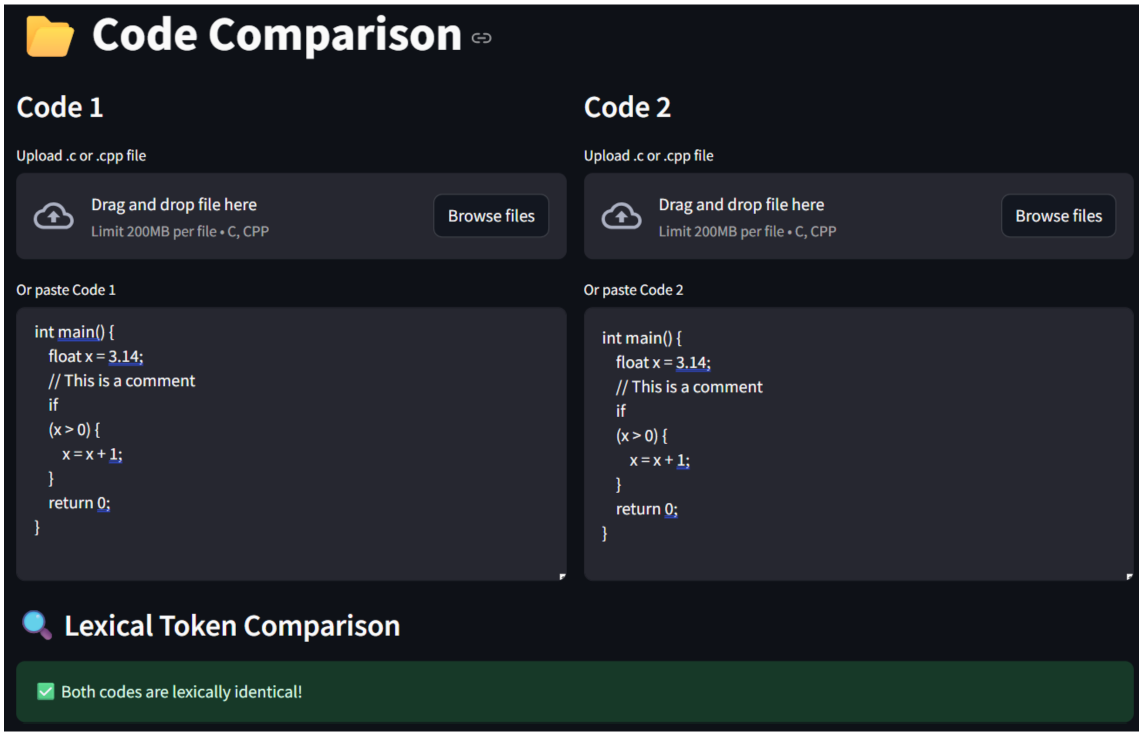
8. Code Comparison Tool

Functionality:
Enables users to compare two code snippets or files to identify lexical differences.

Implementation Details:

- Accepts two separate code inputs via text areas or file uploads.
- Tokenizes both inputs and compares the resulting token sequences.
- Highlights differences in tokens and identifies unique tokens in each code.

Purpose:
Assists in understanding how different code implementations can vary lexically, even if functionally similar.



Code 1:

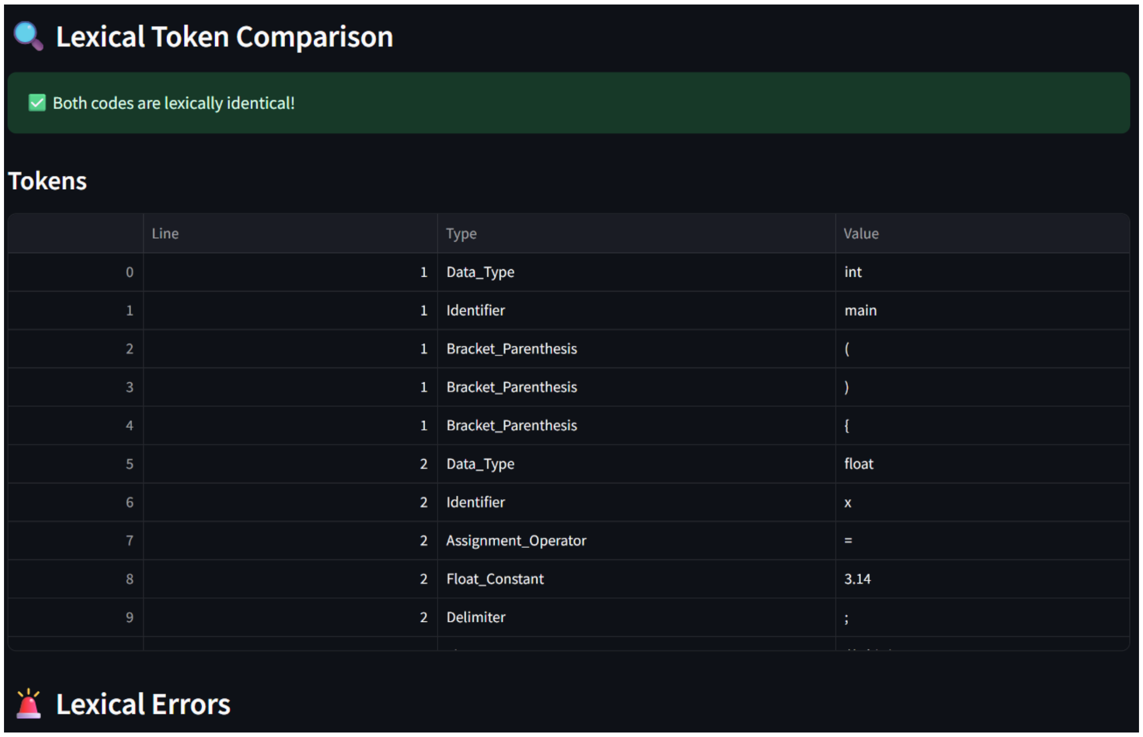
```
int main() {
  float x = 3.14;
  // This is a comment
  if
  (x > 0) {
    x = x + 1;
  }
  return 0;
}
```

Code 2:

```
int main() {
  float x = 3.14;
  // This is a comment
  if
  (x > 0) {
    x = x + 1;
  }
  return 0;
}
```

Lexical Token Comparison

✅ Both codes are lexically identical!



Lexical Token Comparison

✅ Both codes are lexically identical!

Tokens

	Line	Type	Value
0	1	Data_Type	int
1	1	Identifier	main
2	1	Bracket_Parenthesis	(
3	1	Bracket_Parenthesis	)
4	1	Bracket_Parenthesis	{
5	2	Data_Type	float
6	2	Identifier	x
7	2	Assignment_Operator	=
8	2	Float_Constant	3.14
9	2	Delimiter	;

Lexical Errors

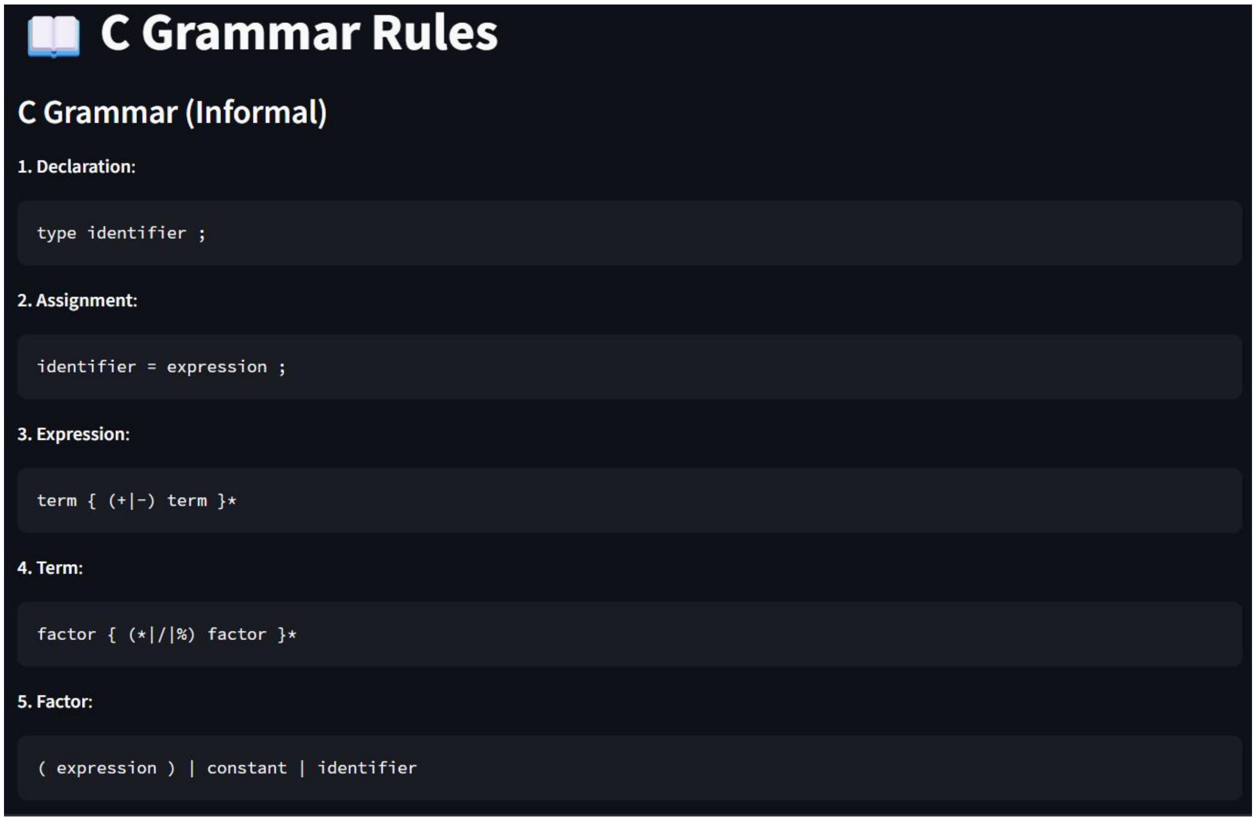
9. Grammar Reference Section

Functionality:
Provides a reference guide outlining the grammar rules and structures recognized by the analyzer.

Implementation Details:

- Displays formatted grammar rules using markdown.
- Covers basic constructs such as declarations, assignments, expressions, and control statements.

Purpose:
Serves as a learning aid for understanding the grammatical structure of C/C++ code.



📖 C Grammar Rules

C Grammar (Informal)

1. Declaration:

```
type identifier ;
```

2. Assignment:

```
identifier = expression ;
```

3. Expression:

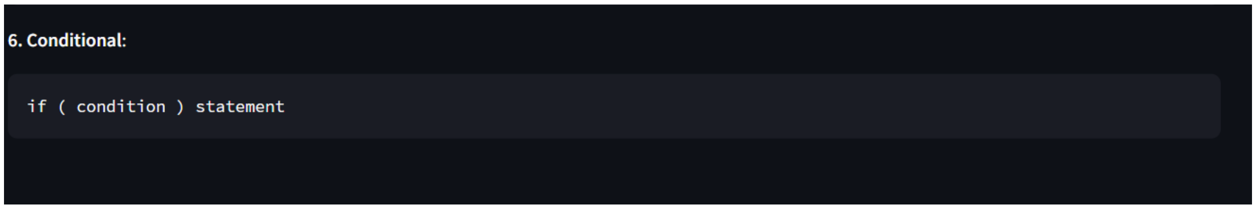
```
term { (+|-) term }*
```

4. Term:

```
factor { (*|/|%) factor }*
```

5. Factor:

```
( expression ) | constant | identifier
```



6. Conditional:

```
if ( condition ) statement
```

User Interface and Experience

The application features a clean and intuitive interface, organized into several tabs accessible via a sidebar menu:

- **Home:** Offers options for live coding and file uploads, displaying tokenization results and errors.
- **Statistics:** Presents token distribution charts for visual analysis.
- **Parse Tree:** Displays the syntactic structure of the code in a tree format.
- **Compare:** Facilitates side-by-side code comparisons to identify lexical differences.
- **Grammar:** Provides a reference guide to the grammar rules recognized by the analyzer.

Each section is designed to be interactive and user-friendly, enhancing the overall learning experience.

Tokens Handled

Our lexical analyzer is capable of identifying and classifying a wide range of tokens from C source code. These include keywords (e.g., int, return), identifiers (user-defined variable and function names), operators (arithmetic, logical, relational), constants (integers, floating-point numbers, characters, and strings), delimiters (commas, semicolons, brackets), and special symbols. The analyzer scans the input line by line and uses regular expressions to match each token accurately while preserving line numbers for context. This robust tokenization process is essential for any further stages of compilation or analysis, ensuring that the source code is broken down into its fundamental syntactic components.

Challenges Faced

One of the most demanding aspects of the project was constructing the parser tree, which took a lot of time and repetitive debugging. Unlike many other lexical analyzers that stop at tokenization, our project stood out by integrating a parser tree to visually represent the syntactic structure of the input code. This involved defining grammar rules, managing precedence and associativity, and recursively breaking down complex expressions—all of which took several days of focused effort. Building the parser tree is a key differentiating feature that sets our project apart from others in the course.

Technical Implementation Details

- **Programming Language:** Python
- **Framework:** Streamlit
- **Libraries Used:**
    - re for regular expressions and pattern matching.
    - pandas for data manipulation and display.
    - plotly.express for generating interactive charts.
    - anytree for constructing and rendering parse trees.
    - streamlit_option_menu for enhanced sidebar navigation.

The application is structured to ensure modularity and scalability, allowing for future enhancements and the addition of more complex parsing capabilities.

Learning Outcomes

Throughput the project we faced a lot of problems and learned a lot. Especially as last semester we had a project in python for the first time and couldn't perform well. This time we worked really hard and achieved this:

- **Practical Application of Automata Theory:** Gained hands-on experience in implementing concepts such as finite automata and regular expressions in a real-world tool.
- **Understanding of Lexical Analysis:** Deepened knowledge of how source code is tokenized and the significance of each token type in the compilation process.
- **Error Handling in Lexical Analysis:** Learned to identify and manage unrecognized tokens, emphasizing the importance of robust error detection mechanisms.
- **Visualization of Code Structure:** Developed skills in representing code syntax through complex parse trees, aiding in the comprehension of code hierarchies.
- **Comparative Code Analysis:** Acquired the ability to analyze and compare different code snippets lexically, fostering a deeper understanding of code variations.
- **User Interface Design:** Enhanced proficiency in creating interactive and user-friendly applications using Streamlit.

Conclusion

Our project has allowed us to use concepts learned in class in real world applications and understand what the main objective of our course was. By implementing a tool that performs comprehensive Tokenization of our C/C++ code by reading it word by word and understanding it (lexical analysis), error detection, statistical visualization, complex parse tree generation, and code comparison. The project has not only reinforced what we learned in class but also provided us an insight into practical applications of the Theory of Automata.

Bibliography

1. Hopcroft, J.E., Motwani, R., & Ullman, J.D. (2006). Introduction to Automata Theory, Languages, and Computation (3rd Edition). Pearson.
   Link: https://www.pearson.com/en-us/subject-catalog/p/introduction-to-automata-theory-languages-and-computation/P200000003158
2. GeeksforGeeks – Lexical Analysis in Compiler Design
   Link: https://www.geeksforgeeks.org/lexical-analysis/
3. Streamlit Official Documentation
   Link: https://docs.streamlit.io/
4. Python re Module Documentation
   Link: https://docs.python.org/3/library/re.html
5. C++ Reference – Language Syntax and Grammar
   Link: https://en.cppreference.com/w/cpp/language