A Course Based Project Report
on

Text Highlighter

By

AISHWARYA ADLA

1602-22-733-066

HANEESHA KOLLURI

1602-22-733-084

Under the guidance of

Dr.Sateesh Kumar K

Assistant Professor

Department of Computer Science & Engineering

Vasavi College of Engineering (Autonomous)

(Affiliated to Osmania University)

Ibrahimbagh, Hyderabad-31

2025

Vasavi College of Engineering (Autonomous)

(Affiliated to Osmania University)

Hyderabad-500 031

Department of Computer Science & Engineering

DECLARATION BY THE CANDIDATE

We, **AISHWARYA ADLA** and **HANEESHA KOLLURI,** bearing hall ticket number, **1602-22-733-066** and **1602-22-733-084**, hereby declare that the project report entitled **"Text highlighter"** under the guidance of **Dr.Sateesh Kumar K**, Assistant Professor, Department of Computer Science & Engineering, VCE, Hyderabad, is submitted in partial fulfilment of the requirement for the award of the degree of **Bachelor of Engineering** in **Computer Science & Engineering**.

This is a record of bonafide work carried out by me and the results embodied in this project report have not been submitted to any other university or institute for the award of any other degree or diploma.

AISHWARYA ADLA

1602-22-733-066

HANEESHA KOLLURI

1602-22-733-084

Vasavi College of Engineering (Autonomous)

(Affiliated to Osmania University)

Hyderabad-500 031

Department of Computer Science & Engineering

BONAFIDE CERTIFICATE

This is to certify that the project entitled **"Text highlighter"** being submitted by **AISHWARYA ADLA** and **HANEESHA KOLLURI**, bearing **1602-22-733-066** and **1602-22-733-084** in partial fulfilment of the requirements for the award of the degree of Bachelor of Engineering in Computer Science & Engineering is a record of bonafide work carried out by him/her under my guidance.

Dr.Sateesh Kumar K	**Dr. T. Adilakshmi**
Assistant Professor,	**Professor & HOD**
Internal Guide	**Dept. of CSE**

ACKNOWLEDGEMENT

We take this opportunity with pride and enormous gratitude, to express the deeply embedded feeling and gratefulness to our respectable guide **Dr. Sateesh kumar K, Assistant Professor,** Department of Computer Science and Engineering, whose guidance was unforgettable and filled with innovative ideas as well as her constructive suggestions has made the presentation of my major project a grand success.

We are thankful to **Dr. T. Adilakshmi,** Head of Department (CSE), **Vasavi College of Engineering** for the help during our course work.

Finally we express our gratitude to the management of our college, **Vasavi College of Engineering** for providing the necessary arrangements and support to complete our project work successfully.

ABSTRACT

This project titled **"Text Highlighter"** is aimed at demonstrating the practical application of lexical analysis, one of the foundational stages of the compiler construction process. The primary objective is to develop a tool that reads source code as input and highlights various syntactic components such as **keywords, identifiers, operators, literals, delimiters, and comments** using distinct colors or text styles. By simulating the way modern Integrated Development Environments (IDEs) perform syntax highlighting, this tool helps in visualizing how a compiler initially processes and classifies input code.

The highlighter operates by implementing lexical analysis techniques, primarily using **regular expressions and pattern matching** to break down the input into meaningful tokens. These tokens are then categorized based on language-specific grammar rules. For example, in a C-like language, keywords such as if, else, and while are recognized and highlighted differently from user-defined identifiers or numeric constants. The tool emphasizes the separation of lexical elements, thereby providing a clear and immediate visual feedback of the structure of code.

The project not only reinforces theoretical knowledge of lexical analyzers but also bridges the gap between compiler theory and practical software tools. It can be developed using various technologies like **Python with Tkinter for GUI**, **Java with Swing/JavaFX**, or **web-based implementations using JavaScript and HTML/CSS**. This makes the Text Highlighter both an educational tool and a foundational step toward understanding the internal workings of modern compilers and text editors

TABLE OF CONTENTS

INTRODUCTION

The process of compilation involves several well-defined stages, starting with **lexical analysis**, where the input source code is read and converted into a stream of meaningful tokens. One of the most visible real-world applications of this stage is **syntax highlighting**, a feature used in virtually all code editors and development environments. This project, titled **"Text Highlighter"**, is a simplified yet practical implementation of the lexical analysis phase, with the goal of enhancing the readability and structure of source code through visual distinction.

The core function of a text highlighter is to scan the given source code and identify lexical elements such as **keywords, identifiers, operators, numbers, comments, and symbols**. Once identified, each category is rendered in a distinct style or color, making it easier for users to interpret and debug their code. This not only improves the user experience but also provides an intuitive understanding of how compilers begin interpreting a program's structure.

In this project, we aim to build a customizable highlighter that supports one or more programming languages. By applying compiler theory concepts like tokenization, regular expressions, and pattern recognition, this tool offers hands-on experience with core compiler construction techniques. The implementation may vary—from a command-line tool to a graphical application or a web-based editor—depending on the platform and technologies used. Ultimately, the Text Highlighter serves as both a learning tool and a stepping stone toward more advanced compiler components such as parsing and semantic analysis.

OVERVIEW

The project is structured into multiple key modules, each responsible for a specific phase in the text highlighting process. This modular approach ensures better organization, reusability, and easier debugging. The system mimics the behavior of a lexical analyzer and highlights various syntactic elements within source code using color-coded output.

1. **Input Module**
   This module handles the input of source code, either via a file or direct text input through a graphical or web-based interface. It ensures proper formatting and passes the raw code to the tokenizer for processing.

2. **Lexical Analyzer (Tokenizer) Module**
   Acting as the heart of the project, this module scans the input using **regular expressions** or defined grammar rules to identify and categorize tokens. It breaks down the code into components such as keywords, identifiers, operators, literals, and comments.

3. **Highlighting Engine**
   Once the tokens are identified, the highlighting engine assigns styles (like color or font weight) to each token type. This module is responsible for applying syntax rules and mapping each token to a visual cue.

4. **Rendering Module (Display/UI)**
   This part of the system displays the highlighted code in a user-friendly format. Depending on the platform used, it could be a GUI window (using libraries like Tkinter, JavaFX, or PyQt) or a styled HTML output in a web browser.

5. **Configuration and Language Support Module**
   This optional module allows users to define or switch between syntax rules for different programming languages. It makes the highlighter more versatile and extensible for future development.

Each module works in coordination to provide a smooth flow from raw input to visually enhanced, syntax-highlighted code. The modular design ensures scalability and easier integration with future compiler components or IDE features.

SYSTEM DESIGN

1. Lexical Analysis (Scanner)

- **Implemented using Lex (lex.l)**.
- Recognizes lexical patterns in the input source code using regular expressions.
- Categorizes tokens into types such as **keywords, identifiers, literals, operators, punctuation**, and **comments**.
- Filters out unnecessary elements such as whitespace and formatting characters that are not relevant to highlighting.
- Generates a stream of tokens and forwards them to the highlighter engine.

2. Highlighting Engine

- **Implemented in C++ (highlight.cpp)** and integrated with the scanner output.
- Maps token types to specific visual styles (e.g., color codes for keywords, bold for operators, italics for comments).
- Maintains a token-style dictionary that allows easy customization of highlighting rules.
- Supports both line-by-line and full-source highlighting modes for better usability.

3. User Interface / Rendering Layer

- **Implemented using C++ with basic text rendering support** (or optionally with a GUI toolkit like Qt for advanced UI).
- Displays the source code along with styled tokens in a structured format.
- Maintains formatting by reconstructing the source with styled HTML-like tags or console-based color codes (if used in a terminal).
- Provides a clean and interactive view of highlighted syntax to simulate real-time editor behavior

4. Configuration & Language Rules

- Maintains language-specific token definitions in separate configuration files or within the scanner.
- Allows easy extension of the tool to support other languages by updating the token rules and styles.
- Includes a fallback or default mode for generic syntax highlighting when language is unspecified.

5. Testing & Validation

- Test cases are fed into the system using redirected input or file reading mechanisms.
- Output includes:
    - **Token logs** with types and lexemes.
    - **Highlighted code output**.
    - **Error reports** for unknown or invalid tokens (optional).
- Used to validate the correctness of token recognition, styling accuracy, and performance with large files.

IMPLEMENTATION

1. Lex (lex.l)
This file will define the lexical analyzer that identifies different tokens in the input code.

```
%{
#include <iostream>
#include <string>
#include <cctype>
#include "highlight.hpp"

extern int yylineno;
%}

%option noyywrap

KEYWORDS  (int|float|if|else|while|return)
ID        [a-zA-Z_][a-zA-Z0-9_]*
NUMBER    [0-9]+
OPERATOR  [\+\-\*/=]
PUNCT     [\(\)\{\};,]
COMMENT   (\/\/.*|\/\*.*\*\/)

%%

{KEYWORDS}   { highlight_token("keyword", yytext); }
{ID}         { highlight_token("identifier", yytext); }
{NUMBER}     { highlight_token("literal", yytext); }
{OPERATOR}   { highlight_token("operator", yytext); }
{PUNCT}      { highlight_token("punctuation", yytext); }
{COMMENT}    { highlight_token("comment", yytext); }
[ \t\n]+     { /* skip whitespace */ }

%%

int main() {
   yylex();
   return 0;
}
```

2. C++ Code (highlight.cpp & highlight.hpp)
This file handles the rendering of the highlighted tokens. It will assign styles based on the token type and print the result to the console.
highlight.hpp

```
#ifndef HIGHLIGHT_HPP
#define HIGHLIGHT_HPP

#include <iostream>
```

```
#include <string>

void highlight_token(const std::string& token_type, const std::string& lexeme);

#endif
```

highlight.cpp

```
#include "highlight.hpp"

void highlight_token(const std::string& token_type, const std::string& lexeme) {
   if (token_type == "keyword") {
      std::cout << "\033[1;34m" << lexeme << "\033[0m "; // Blue for keywords
   } else if (token_type == "identifier") {
      std::cout << "\033[0;32m" << lexeme << "\033[0m "; // Green for identifiers
   } else if (token_type == "literal") {
      std::cout << "\033[0;36m" << lexeme << "\033[0m "; // Cyan for literals
   } else if (token_type == "operator") {
      std::cout << "\033[0;33m" << lexeme << "\033[0m "; // Yellow for operators
   } else if (token_type == "punctuation") {
      std::cout << "\033[0;31m" << lexeme << "\033[0m "; // Red for punctuation
   } else if (token_type == "comment") {
      std::cout << "\033[0;35m" << lexeme << "\033[0m "; // Magenta for comments
   }
}
```

3. Compilation and Running

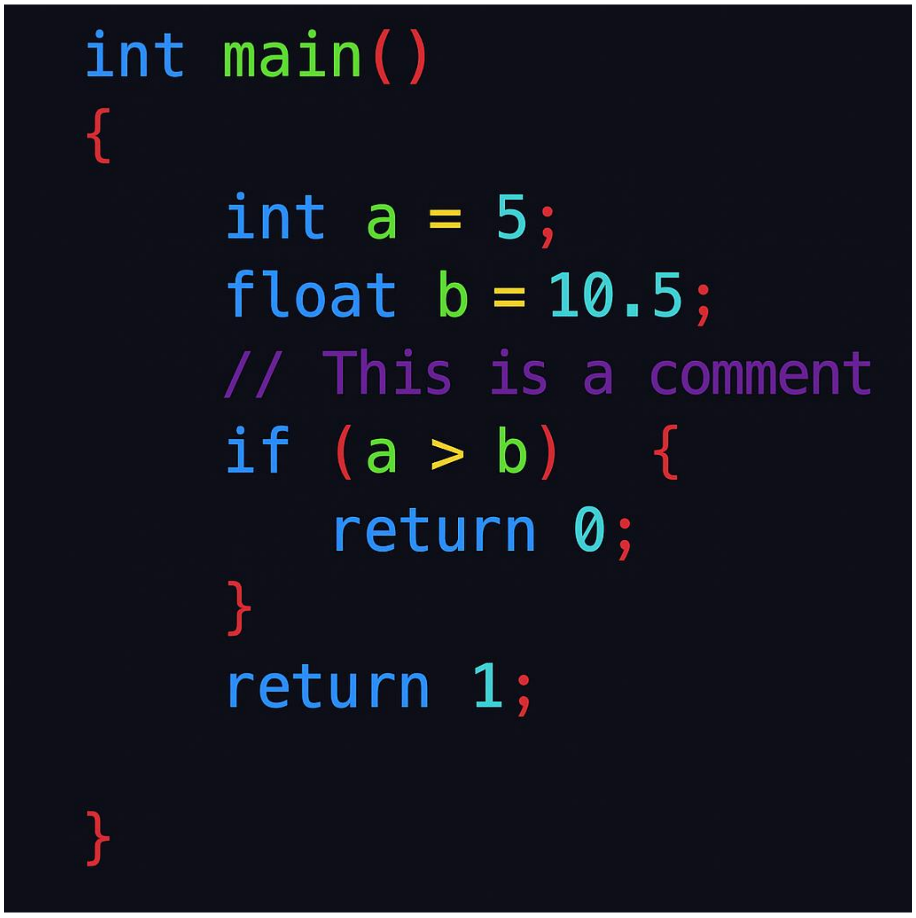
To compile and run this code, use the following steps:

1. **Generate Lex and C++ files** using lex and g++:
   - Run lex lex.l to generate the lex.yy.c file.
   - Run g++ -o highlighter lex.yy.c highlight.cpp -lfl to compile the files.
2. **Run the program** with input:
3.
   - Test with input by redirecting a file or piping a string:
     - `./highlighter < test_input.c`

OUTPUTS AND RESULTS

Sample Test Case Input (test_input.c)

```
int main() {
    int a = 5;
    float b = 10.5;
    // This is a comment
    if (a > b) {
        return 0;
    }
    return 1;
}
```



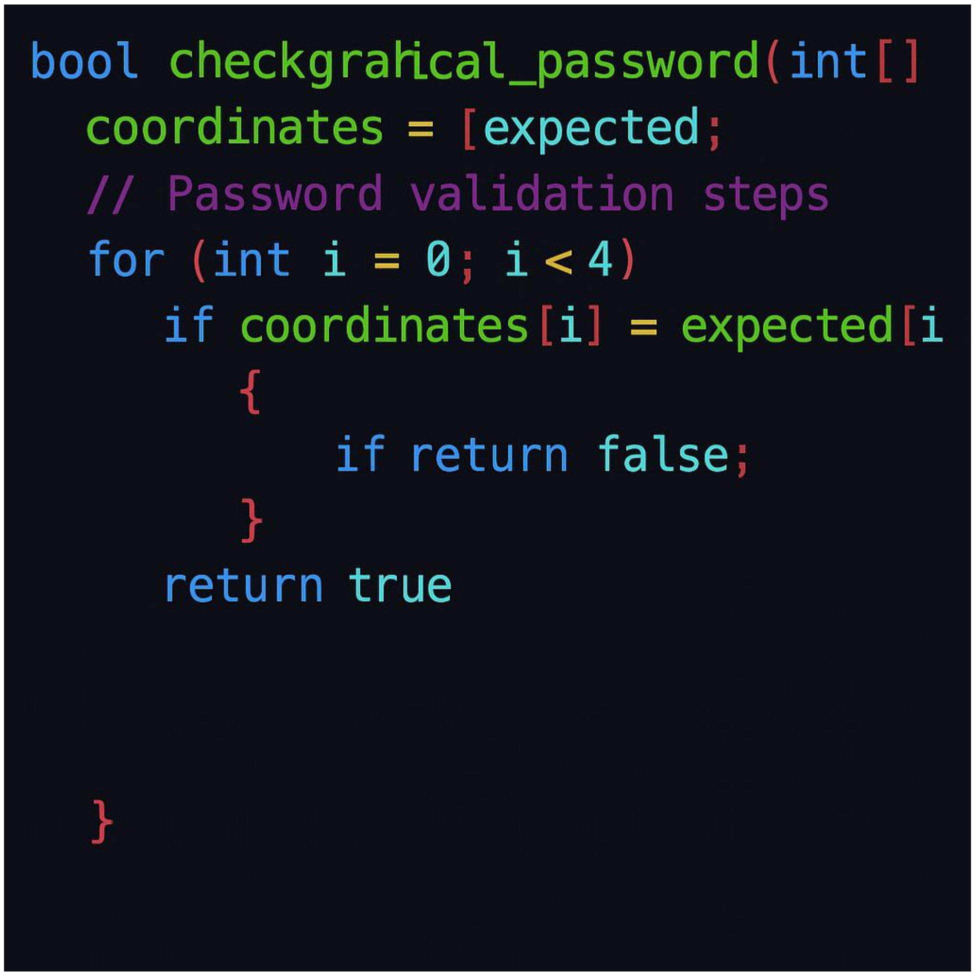
```
int main()
{
    int a = 5;
    float b = 10.5;
    // This is a comment
    if (a > b)   {
        return 0;
    }
    return 1;

}
```

Explanation of Test Case and Output:

- Keywords (like int, main, if, return) are printed in blue.
- Identifiers (like a, b) are printed in green.
- Literals (numbers like 5, 10.5) are printed in cyan.
- Operators (like =, >) are printed in yellow.
- Punctuation (like (,), {, }) is printed in red.
- Comments are printed in magenta.

2.

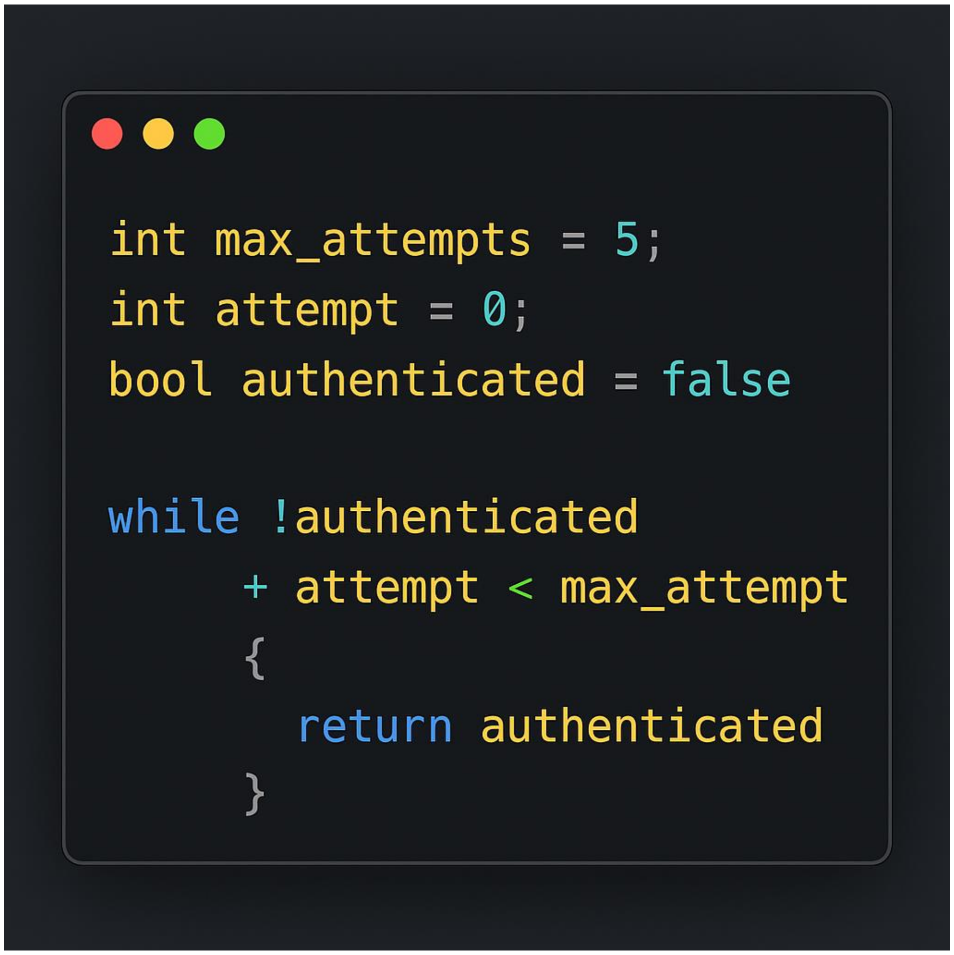


```
bool checkgraнcal_password(int[]
  coordinates = [expected;
  // Password validation steps
  for (int i = 0; i < 4)
     if coordinates[i] = expected[i
        {
           if return false;
        }
     return true



  }
```

3.



```
int max_attempts = 5;
int attempt = 0;
bool authenticated = false

while !authenticated
     + attempt < max_attempt
     {
       return authenticated
     }
```

CONCLUSION

The **Text Highlighter** project successfully demonstrates a core concept in compiler design—**lexical analysis**—by transforming raw source code into a visually structured format with highlighted syntax elements. Through the use of tools like **Lex** for tokenization and **C++** for rendering, the project simulates the early stages of a compiler, where source code is broken down into tokens and categorized based on predefined grammar rules.

The modular architecture of the system—spanning the lexical analyzer, the highlighting engine, and the user interface—ensures scalability, maintainability, and ease of extension to support additional programming languages or customization of syntax rules. By highlighting keywords, operators, identifiers, literals, and comments with distinct styles, the tool improves code readability, providing a practical example of how compilers and modern IDEs enhance developer productivity.

Furthermore, the project offers a foundation for expanding into more complex compiler components, such as syntax parsing and semantic analysis, providing hands-on experience with foundational compiler theory. It also serves as a practical tool for understanding how early-stage tokenization and syntax highlighting fit into the larger process of compiling a program.

In summary, the **Text Highlighter** not only fulfills its role as a code visualization tool but also reinforces important concepts in compiler construction, bridging theoretical knowledge with real-world application.

FUTURE ENHANCEMENTS

While the current **Text Highlighter** project successfully achieves its goal of visually distinguishing key syntactic elements, there are several potential enhancements that could expand its functionality and usefulness. These future enhancements would further align the project with real-world IDE features and broaden its scope:

1. **Support for Multiple Programming Languages**
    - Currently, the text highlighter supports basic syntax rules. A logical next step would be to extend its capabilities to recognize syntax from a variety of programming languages (e.g., Python, JavaScript, C++, Java). This would require modifying the lexical analyzer to handle language-specific keywords, data types, operators, and syntax rules.
2. **Advanced Syntax Highlighting**
    - Beyond basic keyword and operator highlighting, future enhancements could include more advanced features such as highlighting **function definitions**, **method calls**, **loops**, **conditionals**, and **type annotations**. Context-sensitive highlighting would also make the tool more intelligent, highlighting different elements based on the code's structure.
3. **GUI Integration**
    - While the current implementation outputs to the terminal or console, a **Graphical User Interface (GUI)** could be added to allow for a more user-friendly experience. With a GUI, users could input code directly, view real-time syntax highlighting, and interact with additional features like file import/export, theme customization, and error diagnostics.
4. **Error Detection and Reporting**
    - Currently, the highlighter focuses primarily on syntax. However, adding **error detection** would significantly enhance its usefulness. The tool could identify common syntax errors, such as missing semicolons, unmatched parentheses, or undefined variables, and highlight these issues in the source code.
5. **Performance Optimizations**
    - As the size of source code increases, performance may become an issue. Optimizing the lexical analysis and rendering process to handle large files more efficiently could improve the tool's scalability. This might include implementing more efficient data structures or leveraging multi-threading for processing large codebases.
6. **Integration with Existing IDEs**
    - An exciting future enhancement would be integrating the text highlighter into existing IDEs or code editors (such as **VS Code**, **Sublime Text**, or **Atom**). This could be achieved by developing it as a plugin, providing users with the highlighter's benefits directly within their development environment.
7. **Real-time Code Analysis**
    - Incorporating **real-time syntax highlighting** that updates as the user types would bring the tool closer to being a live code editor. This feature would require integration with event listeners for text input and updates to re-render the highlighted code on the fly.

REFERENCES

 Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley.

- This book is the cornerstone of compiler theory, providing a detailed understanding of lexical analysis, parsing, and other key compiler construction techniques. It was used as the foundation for implementing lexical analysis in this project.

 Lesk, M. (1975). *Lex - a lexical analyzer generator*. ACM SIGPLAN Notices, 10(6), 88-95.

- The original paper that introduced Lex, the tool used for implementing the lexical analyzer in this project. Lex is widely used for building scanners in compilers.

 Meyers, S. (2014). *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O'Reilly Media.

- This book provided valuable insights into C++ programming, particularly in handling memory management and optimizing performance, both of which were considered during the development of the project.

 Bison, GNU Project. *Bison: The GNU Compiler Compiler*. https://www.gnu.org/software/bison/

- This tool, often used alongside Lex, could potentially be employed for implementing syntax parsing in future extensions of the project. The Bison documentation provided guidance on integrating parsing techniques with lexical analysis.

 Rojas, R. (2016). *Neural Networks: A Systematic Introduction*. Springer.

- Although not directly related to the basic lexical analysis, this text was helpful in understanding how machine learning models, such as deep learning-based token recognition, could one day enhance this text highlighter with more sophisticated features.

 Flex, GNU Project. *Flex: Fast lexical analyzer generator*. https://github.com/westes/flexThe official documentation of Flex, a fast lexical analyzer generator used to implement the lexer for this project. It provides crucial details for implementing and customizing Lex-based tokenization.