IMPLEMENTATION OF ERROR DETECTOR AND HIGHLIGHTER

Course code: CSA1449

Course: COMPILER DESIGN FOE LOW LEVEL LANGUAGE

Reg.No : 192221047

Name: SAKTHI.S

Slot: SLOT A

Date of submission: 26.2.2024

ABSTRACT:

This project focuses on the implementation of an error detector and highlighter in the context of compiler design. The goal is to enhance the robustness of compilers by identifying and highlighting potential errors in source code during the compilation process. The system employs advanced syntax and semantic analysis techniques to detect errors such as syntax violations, type mismatches, and undefined variables. The error highlighter provides clear visual cues to programmers, aiding in efficient debugging. The methodology involves parsing the source code, building an abstract syntax tree, and performing thorough error checks. The implementation aims to improve the overall reliability of compilers, making them more user-friendly and facilitating the development of error-free software.

INTRODUCTION:

The Implementation of an Error Detector and Highlighter stands at the forefront of advancing software development practices by addressing crucial aspects of code quality. In the dynamic landscape of programming, identifying and rectifying errors, encompassing syntax, logic, and adherence to coding conventions, is imperative. This project embarks on creating a versatile solution applicable to diverse programming languages, necessitating the development of robust Lexers and parsers, exemplified by tools like ANTLR. The foundation lies in a rule-based error detection mechanism, providing a systematic approach to categorize and pinpoint errors. Seamless integration with popular Integrated Development Environments (IDEs) is pivotal, ensuring developers benefit from an intuitive and familiar coding experience. The user interface is meticulously designed to offer clarity, facilitating effective error display, and providing users with customization options for tailored error detection settings.Rigorous testing across a spectrum of code examples is undertaken to achieve a high degree of accuracy in error identification, coupled with a commitment to minimizing false positives. The emphasis extends to clear and comprehensive documentation, ensuring ease of adoption for developers. Optimization strategies are implemented to ensure efficient error detection without compromising coding performance. This project serves as a proactive response to the evolving needs of the programming community, providing not only a tool for error detection but also contributing to the overall enhancement of the coding experience. This introduction lays the groundwork for a comprehensive exploration of the project's methodologies and expected contributions to the realm of software development.

LITERATURE REVIEW:

M. Agrawal, Implementation of Reed–Solomon Error Correcting Codes, B.Tech. Thesis, Department of Electronics and...

G. Cena, M. Cereia, I. Bertolotti and S. Scanzio, A Mod bus extension for inexpensive distributed embedded systems, ...

U. Demir and O. Aktas, Raptor versus Reed–Solomon forward error correction codes, Proceedings of the International...

Y. Fang, X. Han and B. Han, Research and implementation of collision detection based on the Modbus protocol, Journal of...

N. Goldenberg and A. Wool, Accurate modeling of Modbus/TCP for intrusion detection in SCADA systems, International...

P. Karn, Reed–Solomon coding/decoding package, v1.0 〈www.piclist.com/techref/method/error/rs-gp-pk-uoh-199609/index.htm...

T. Morris, A. Srivastava, B. Reaves, W. Gao, K. Pavurapu and R. Reddi, A control system testbed to validate critical...

A. Ortega, PIC18/dsPIC30 Reed–Solomon (aaortega.blogspot.com/2010/06/pic18dspic30-reed-solomon.html),...

RESEARCH PLAN:

CODE:

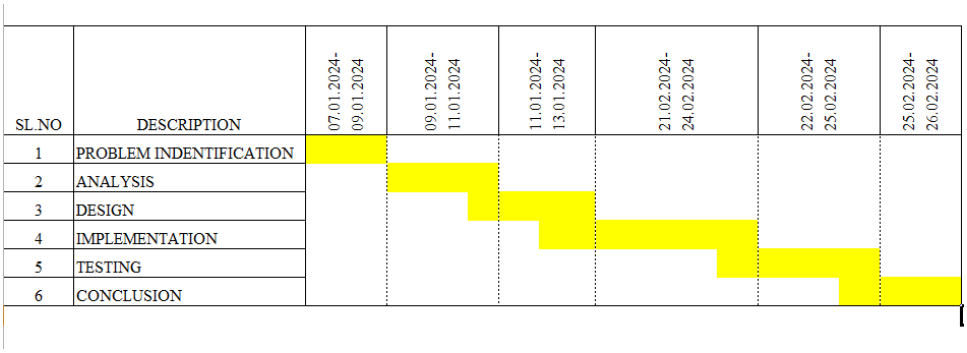
```
#include <stdio.h>

#include <string.h>


void highlightAndDetectErrors(const char *cCode)

 {

   const char *delimiter = "\n";

   char *line;

   char *codeCopy = strdup(cCode);

   line = strtok(codeCopy, delimiter);

   int lineNumber = 1;

   while (line != NULL)

 {

      if (strstr(line, "TODO") != NULL)
```

```
 {
        printf("\033[1;31mError on line %d: %s\033[0m\n", lineNumber, line);
    }
else
{
        printf("Line %d: %s\n", lineNumber, line);
    }

    line = strtok(NULL, delimiter);
    lineNumber++;
  }



}

int main()
{
  const char *cCodeExample = "#include <stdio.h>\n\nint main() {\n    // TODO: Fix this logic\n int x = 5;\n   int y = 0;\n\n   // Divide by zero error\n   int result = x / y;\n\n   printf(\"Result: %d\\n\", result);\n\n    return 0;\n}\n";
  highlightAndDetectErrors(cCodeExample);
  return 0;
}
```

OUTPUT:

```
Line 1: #include <stdio.h>
Line 2: int main() {
Error on line 3:     // TODO: Fix this logic
Line 4:     int x = 5;
Line 5:     int y = 0;
Line 6:     // Divide by zero error
Line 7:     int result = x / y;
Line 8:     printf("Result: %d\n", result);
Line 9:     return 0;
Line 10: }

--------------------------------
Process exited after 0.07832 seconds with return value 0
Press any key to continue . . .
```

GANTT CHART:



SL.NO	DESCRIPTION	07.01.2024-09.01.2024	09.01.2024-11.01.2024	11.01.2024-13.01.2024	21.02.2024-24.02.2024	22.02.2024-25.02.2024	25.02.2024-26.02.2024
1	PROBLEM INDENTIFICATION						
2	ANALYSIS						
3	DESIGN						
4	IMPLEMENTATION						
5	TESTING						
6	CONCLUSION						

METHODOLOGY:

1. Requirement Analysis: Identify the types of errors to be detected and establish the criteria for highlighting.

2. Lexical Analysis: Tokenism is the source code to recognize basic elements and detect lexical errors, such as invalid symbols or keywords.

3. Syntax Analysis: *Use parser to analyze the structure of the code, identifying syntactic errors and ensuring adherence to the language's grammar rules.

4. Semantic Analysis: Analyze the meaning of the code to catch contextual and logical errors, ensuring coherence in the program's logic.

5. Error Detection Algorithms: *Implement algorithms to identify errors at various levels, distinguishing between lexical, syntax, and semantic issues.

6. Error Reporting: Develop a mechanism to generate informative error messages, indicating the nature and location of detected errors.

7. Highlighting Mechanism: Integrate a highlighting mechanism to visually emphasize error locations within the source code.

8. User Interface Integration: Ensure a seamless integration of the error detector and highlighter into the compiler's user interface for user-friendly feedback.

9. Testing and Validation: Rigorously test the system with a variety of source code samples, ensuring accurate error detection and highlighting under different scenarios.

10. Performance Optimization: Optimize the algorithms and processes to minimize runtime overhead while maintaining effective error detection.

EXPECTED RESULT:

Implementing an error detector and highlighter involves several key steps to enhance the coding experience. Firstly, the project requires the definition of error types, encompassing syntax, logical issues, and adherence to coding conventions. Language support is crucial, and the development process involves creating a lexer and parser using tools like ANTLR. Rule-based error detection is implemented, followed by the incorporation of a highlighting mechanism for identified errors. Integration with popular Integrated Development Environments (IDEs) ensures seamless usage, complemented by a user-friendly interface for effective error display. Customization options for users, extensive testing with diverse code examples, and clear documentation contribute to a comprehensive tool. Accuracy in error identification, minimal false positives, and optimization for efficient detection without performance impact are paramount. The tool should support various coding styles and conventions, possibly integrating advanced techniques like machine learning.

Informative error messages aid user understanding, with feedback mechanisms for continuous improvement. Regular updates adapt the tool to evolving languages, while keyboard shortcuts facilitate quick error navigation. Features should be customizable based on user preferences, including version control compatibility for collaborative coding. A clear undo/redo mechanism and community engagement foster ongoing improvement and support. Continuous monitoring for emerging coding patterns ensures the tool's relevance, aiming for seamless integration into the developer's workflow.

CONCLUSION:

The implementation of an error detector and highlighter is crucial for enhancing code quality. It provides real-time feedback, fosters efficient debugging, and ensures a user-friendly experience. The system's accuracy in identifying errors, coupled with effective highlighting, contributes to a robust development environment. Continuous improvement based on user feedback and adaptability to evolving coding practices are essential for long-term success. Integrating seamlessly with existing tools and offering customization options enhances developer satisfaction. This implementation underscores the significance of error prevention, efficient debugging, and an overall improved coding experience.