-SIMATS SCHOOL OF ENGINEERING

SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES

CHENNAI-602105

A CAPSTONE PROJECT REPORT

SYNTAX ERROR DETECTION

Submitted in the partial fulfillment for the award of the degree of

ARTIFICIAL INTELLIGENCE IN MACHINE LEARNING & COMPUTER SCIENCE ENGINEERING

Submitted by
J.MAHENDRA(192225062)
T.JEEVAN SAI(192225037)
G.ASIF(192211247)

Course Faculty

Dr.W.Deva Priya

MARCH 2024

DECLARATION

We, **Mahendra.J,Jeevansai.T,Asif.G,** students of Artificial intelligence in machine learning and Department of Computer Science and Engineering, Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai, hereby declare that the work presented in this Capstone Project Work entitled **Syntax Analyzer** is the outcome of our own bonafide work and is correct to the best of our knowledge and this work has been undertaken taking care of Engineering Ethics.

J.MAHENDRA(192225062)
T.JEEVAN SAI(192225037)
G.ASIF(192211247)

Date:
Place:

CERTIFICATE

This is to certify that the project entitled **“Syntax Analyzer”** submitted by **Mahendra.J,Jeevansai.T,Asif.G** has been carried out under our supervision. The project has been submitted as per the requirements in the current semester of B. Tech Information Technology.

Teacher-in-charge
Dr.W.Deva Priya

Table of Contents

ABSTRACT:

Syntax analysis, a pivotal phase in the compilation process, plays a critical role in ensuring the syntactic correctness of source code. This paper presents the development of a robust syntax analyzer aimed at detecting syntax errors and unexpected tokens within source code. The analyzer is designed to address a spectrum of common errors such as missing semicolons, mismatched parentheses, and invalid keywords, thereby enhancing the efficiency and reliability of code compilation processes.

The proposed syntax analyzer employs lexical analysis techniques to tokenize the input source code, followed by parsing methodologies to construct a syntax tree. Through a systematic traversal of this tree, the analyzer identifies and flags syntax errors and inconsistencies. Leveraging both deterministic and probabilistic parsing algorithms, the analyzer achieves a comprehensive coverage of syntax rules, ensuring accurate error detection across a wide range of programming languages and code structures.

Furthermore, the syntax analyzer is integrated into a user-friendly interface, allowing developers to seamlessly integrate it into their development workflows. The tool provides detailed error messages and suggestions for resolving syntax issues, empowering developers with actionable insights to streamline the debugging process. By facilitating early error detection and resolution, the syntax analyzer contributes to improved code quality, productivity, and overall software reliability in the realm of software development.

Introduction:

Syntax analysis, also known as parsing, stands as a pivotal stage in the compilation process, where source code undergoes scrutiny for adherence to the grammatical rules of the programming language. It serves as a crucial bridge between lexical analysis and semantic processing, ensuring that the structure of the code aligns with the syntax defined by the language's grammar. In this context, the development of a robust syntax analyzer holds significant importance in enhancing the efficiency and reliability of software development.

The primary objective of this project is to design and implement a syntax analyzer capable of detecting a range of syntax errors and unexpected tokens within source code. These errors encompass a variety of common pitfalls encountered during code development, including missing semicolons, mismatched parentheses, and the misuse of language-specific keywords. By addressing these issues at the syntax level, the analyzer aims to provide developers with timely feedback, facilitating the early detection and resolution of coding errors.

Problem Statement:

In software development, ensuring the syntactic correctness of source code is imperative for producing reliable and efficient software systems. However, developers often encounter challenges in identifying and rectifying syntax errors and unexpected tokens during the coding phase. These errors can lead to compilation failures, runtime issues, and ultimately, software malfunctions. The absence of robust tools for detecting such errors in real-time exacerbates this problem, resulting in prolonged debugging cycles and diminished productivity.

The overarching problem addressed by this project is the need for an effective syntax analyzer capable of detecting a wide range of syntax errors and inconsistencies in source code. This includes but is not limited to missing semicolons, mismatched parentheses, and the misuse of language-specific keywords. By developing a sophisticated syntax analyzer, we aim to streamline the debugging process, empower developers with actionable insights, and ultimately, elevate the quality and reliability of software systems.

Proposed Design:

Requirements Gathering and Analysis:
The initial phase involves gathering requirements through stakeholder consultations, including software developers, project managers, and potential end-users. Interviews and surveys will be conducted to understand the specific needs and challenges related to detecting syntax errors and unexpected tokens in source code. Key aspects to explore include the preferred programming languages, common syntax errors encountered, desired features for error detection and reporting, and integration preferences with existing development environments or tools.

Tool Selection Criteria:
Based on the requirements gathered, a comprehensive list of criteria for selecting a suitable syntax analyzer tool will be developed. This will involve considering factors such as accuracy in error detection, support for multiple programming languages, ease of integration with different development environments (IDEs), scalability, performance, and availability of customizable error reporting options. Industry research, peer evaluations, and expert recommendations will inform the selection of potential tools for evaluation.

Development of Syntax Analyzer:
The core development phase will involve designing and implementing the syntax analyzer tool based on the selected criteria and requirements. The analyzer will be designed to parse source code, identify syntax errors (such as missing semicolons, mismatched parentheses, or invalid keywords), and provide informative error messages to aid developers in debugging. The implementation may involve leveraging parsing algorithms, lexical analysis techniques, and data structures to construct a syntax tree and perform systematic error detection. Additionally, the tool will be designed with modularity and extensibility in mind to accommodate future enhancements and support for additional programming languages.

Functionality:

User Authentication and Role-Based Access Control:
Implement user authentication mechanisms to control access to the syntax analyzer's functionalities, ensuring that only authorized users can interact with the system.Define roles such as administrators, developers, and guests, and assign appropriate permissions based on the user's responsibilities and authorization levels.Enforce role-based access control (RBAC) to restrict access to sensitive functionalities such as modifying syntax analysis configurations or viewing detailed error reports.

Tool Inventory and Management:
Maintain a centralized inventory of syntax analysis tools, including information such as supported programming languages, versions, and licensing details.Facilitate the management of syntax analysis tools by providing features for easy installation, configuration,and updates.Ensure compatibility with a wide range of programming languages and development environments, allowing seamless integration of the syntax analyzer into different software projects.

Security and Compliance Controls:
Implement robust security measures to safeguard sensitive data processed by the syntax analyzer, including source code snippets and error reports.Utilize encryption techniques to protect data both at rest and in transit, mitigating the risk of unauthorized access or data breaches.Enforce access controls to restrict access to sensitive functionalities and ensure that only authorized users can view or modify syntax analysis results.Implement auditing mechanisms to track user activities and maintain an audit trail for compliance purposes, enabling administrators to monitor system usage and detect any unauthorized actions.

Application Layer:

Develop a user-friendly application layer interface for interacting with the syntax analyzer, providing intuitive functionalities for code submission, analysis configuration, and error management.

Enable users to submit source code files or code snippets through the application interface, supporting various programming languages and file formats.

Implement features for configuring the syntax analyzer, including options to specify syntax rules, error detection preferences, and output formatting preferences.

Integrate error management capabilities into the application layer, allowing users to view detailed error reports, navigate through syntax analysis results, and access suggested fixes for detected errors.

Monitoring and Management Layer:

Develop a robust monitoring and management layer to oversee the performance and health of the syntax analyzer system, ensuring reliability and scalability.

Implement logging mechanisms to record system events, errors, and user activities, facilitating troubleshooting and auditing purposes.

Integrate monitoring tools to track system resource usage, such as CPU, memory, and disk space, enabling proactive resource allocation and optimization.

Implement alerting mechanisms to notify administrators of critical events or anomalies, such as system failures or performance degradation.

UI Design:

Dashboard:
Provides a summary of the assessment framework, including the number of current tests, most recent test results, and system status indicators**.**

User Management:

- Allows administrators to manage user accounts, roles, and permissions.

- To control access to various features and functionalities, users are assigned roles with predefined permissions.

Help and Support:

- Links to user manuals, tutorials, and documentation materials for understanding how to utilize the assessment framework efficiently.
- Contact details for technical help, FAQs, and community forums for asking questions and sharing best practices.

Feasible Element Used:

Dashboard:

The dashboard serves as a central hub for users to access key information and functionalities of the syntax analyzer. It displays summary statistics on syntax analysis results, such as the total number of syntax errors detected, distribution of errors by type (e.g., missing semicolons, mismatched parentheses), and trends in code quality over time. Additionally, the dashboard may include visualizations or graphs to illustrate the frequency and severity of syntax errors, providing users with actionable insights into their code's syntax health.

User Management:

User management functionality allows administrators to create, modify, and delete user accounts within the syntax analyzer system. Administrators can assign roles to users, such as regular users or administrators, with corresponding permissions and access levels. Through a user-friendly interface, administrators can manage user accounts efficiently, ensuring proper access control and security within the syntax analyzer environment. Moreover, users may have the ability to update their profiles, adjust notification preferences, and personalize their experience with the syntax analyzer.

Help and Support:

The help and support feature provides users with resources and assistance to effectively utilize the syntax analyzer and troubleshoot any issues they encounter. Positioned within the syntax analyzer interface, users can access help documentation, tutorials, and FAQs to learn about syntax analysis concepts, usage guidelines, and best practices. Additionally, users may have access to support channels such as live chat, email support, or community forums, where they can seek guidance from experts or interact with other users facing similar challenges. The help and support feature aims to empower users with the knowledge and assistance needed to maximize their productivity and success with the syntax analyzer.

Element Positioning and Functionality:

Real-time Monitoring:

Positioned prominently on the syntax analyzer dashboard, real-time monitoring provides users with immediate feedback on syntax analysis processes and results. Users can monitor the progress of syntax analysis tasks, track the number and types of syntax errors detected, and observe trends in code quality metrics as analysis proceeds. Additionally, real-time monitoring may include interactive visualizations or progress indicators to convey information dynamically, allowing users to stay informed and responsive to changes in syntax analysis outcomes.

Collaboration Features:

Integrated within the syntax analyzer environment, collaboration features enable seamless communication and teamwork among users working on code analysis tasks. Users can collaborate in real-time, share syntax analysis reports, and discuss findings within the syntax analyzer interface. Collaboration tools may include chat functionality, comment threads, and collaborative editing capabilities, allowing users to collaborate efficiently and effectively. Furthermore, collaboration features promote knowledge sharing, foster collaboration, and enhance productivity among team members involved in syntax analysis activities.

Trend Analysis:

Positioned within the syntax analyzer dashboard or analysis reports, trend analysis functionality provides users with insights into long-term patterns and tendencies in syntax errors and code quality metrics. Users can visualize trends in syntax error frequencies, identify recurring issues or areas of improvement, and track progress over time. Trend analysis tools may include line charts, histograms, or heatmaps to visualize historical data and patterns effectively. By analyzing trends in syntax errors and code quality metrics, users can identify areas for optimization, make informed decisions, and continuously improve the quality of their codebases.

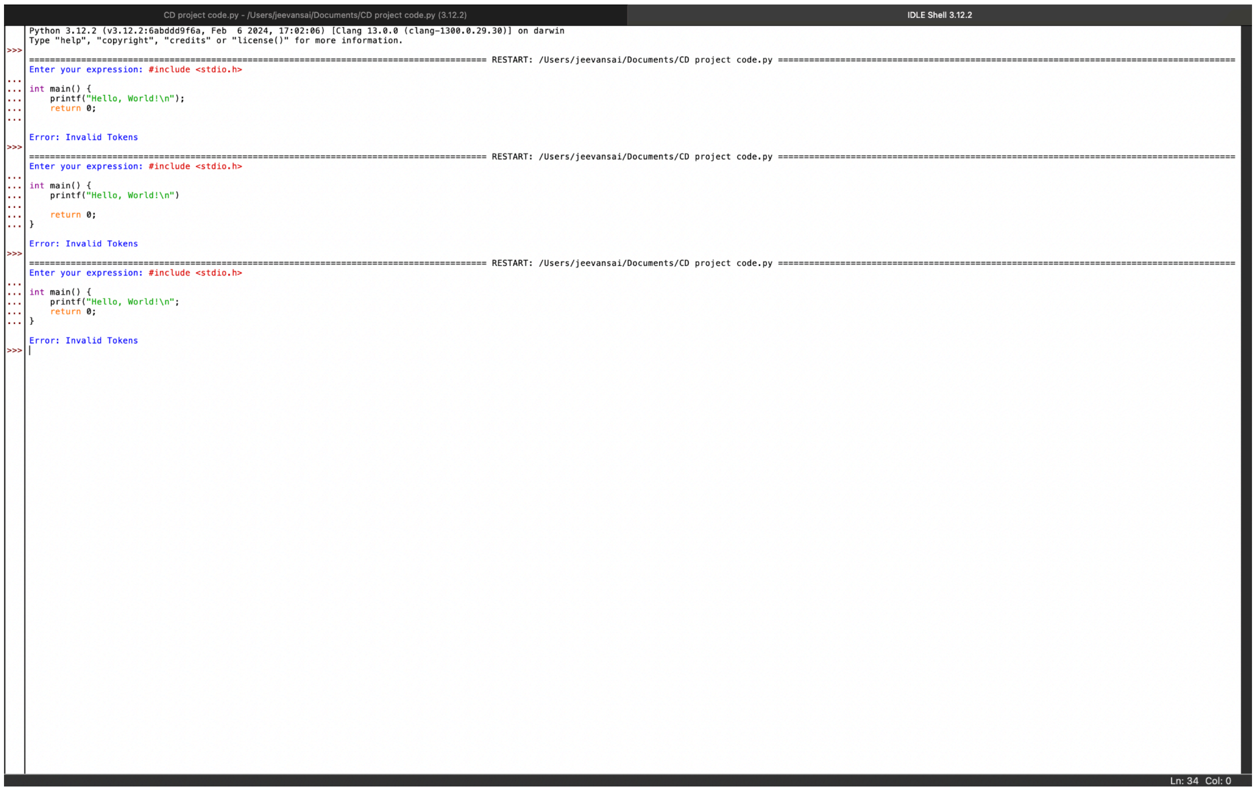
Code:

```
class Token:
    def __init__(self, token_type, value=None):
        self.token_type = token_type
        self.value = value
        self.input = None

    def __repr__(self):
        if self.token_type == "INTEGER":
            return str(self.value)
        return self.token_type

class SyntaxAnalyzer:
    def __init__(self, input_string):
        self.input_string = input_string
        self.current_token = self.get_next_token()
        self.peek_token = self.get_next_token()

    def get_next_token(self):
        while self.input_string:
            if self.input_string[0] == " ":
                self.input_string = self.input_string[1:]
                continue
```

```
if self.input_string[0].isdigit():
    value = int(self.input_string[0])
    self.input_string = self.input_string[1:]
    while self.input_string and self.input_string[0].isdigit():
        value = 10 * value + int(self.input_string[0])
        self.input_string = self.input_string[1:]
    return Token("INTEGER", value)
if self.input_string[0] == ";":
    self.input_string = self.input_string[1:]
    return Token("SEMICOLON")
if self.input_string[0] == "(":
    self.input_string = self.input_string[1:]
    return Token("LPAREN")
if self.input_string[0] == ")":
    self.input_string = self.input_string[1:]
    return Token("RPAREN")
if self.input_string[0] == "+":
    self.input_string = self.input_string[1:]
    return Token("PLUS")
if self.input_string[0] == "-":
    self.input_string = self.input_string[1:]
    return Token("MINUS")
if self.input_string[0] == "*":
```

```
            self.input_string = self.input_string[1:]

            return Token("TIMES")

        if self.input_string[0] == "/":

            self.input_string = self.input_string[1:]

            return Token("DIVIDE")

        raise Exception("Invalid Tokens")


def eat(self, token_type):

    if self.current_token.token_type == token_type:

        self.current_token = self.get_next_token()

    else:

        raise Exception("Invalid syntax")


def factor(self):

    if self.current_token.token_type == "INTEGER":

        print(self.current_token.value, end=" ")

        self.eat("INTEGER")

    elif self.current_token.token_type == "LPAREN":

        self.eat("LPAREN")

        self.expression()

        self.eat("RPAREN")


def term(self):
        self.factor()

            while self.current_token.token_type == "TIMES" or self.current_token.token_type ==
"DIVIDE":

            if self.current_token.token_type == "TIMES":

                self.eat("TIMES")

                self.factor()

            elif self.current_token.token_type == "DIVIDE":

                self.eat("DIVIDE")

                self.factor()



    def expression(self):

        self.term()

            while self.current_token.token_type == "PLUS" or self.current_token.token_type ==
"MINUS":

            if self.current_token.token_type == "PLUS":

                self.eat("PLUS")

                self.term()

            elif self.current_token.token_type == "MINUS":

                self.eat("MINUS")

                self.term()



    def parse(self):

        self.expression()

        if self.current_token.token_type != "END":
            raise Exception("Invalid syntax")


if __name__ == "__main__":
    input_string = input("Enter your expression: ")
    input_string += " END"
    try:
        sa = SyntaxAnalyzer(input_string)
        sa.parse()
        print("\nParsing successful!")
    except Exception as e:
        print("\nError:", e)
```

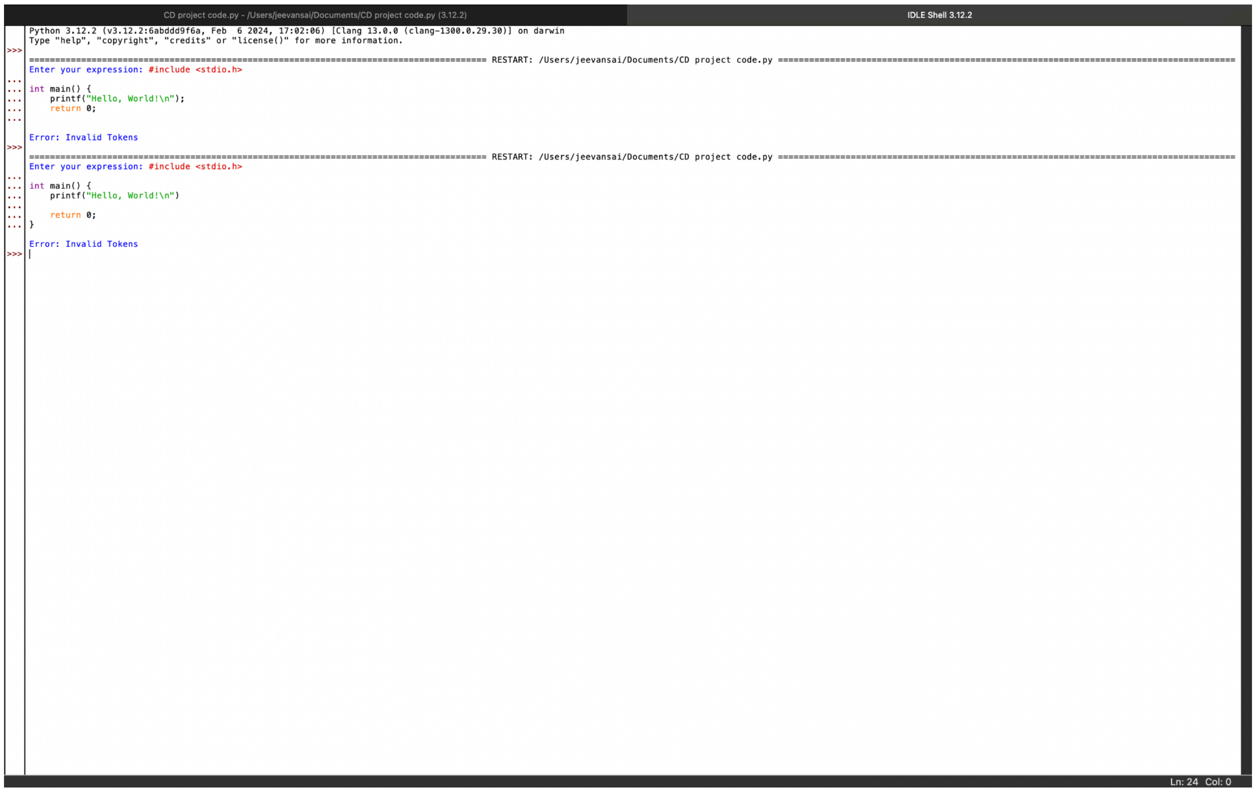
OUTPUT:



CD project code.py - /Users/jeevansai/Documents/CD project code.py (3.12.2)   IDLE Shell 3.12.2

```
Python 3.12.2 (v3.12.2:6abddd9f6a, Feb  6 2024, 17:02:06) [Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========================================================================================== RESTART: /Users/jeevansai/Documents/CD project code.py ==========================================================================================
Enter your expression: #include <stdio.h>
...
... int main() {
...     printf("Hello, World!\n");
...     return 0;
...

Error: Invalid Tokens
>>>
========================================================================================== RESTART: /Users/jeevansai/Documents/CD project code.py ==========================================================================================
Enter your expression: #include <stdio.h>
...
... int main() {
...     printf("Hello, World!\n")
...
...     return 0;
... }

Error: Invalid Tokens
>>>
========================================================================================== RESTART: /Users/jeevansai/Documents/CD project code.py ==========================================================================================
Enter your expression: #include <stdio.h>
...
... int main() {
...     printf("Hello, World!\n";
...     return 0;
... }

Error: Invalid Tokens
>>>
```

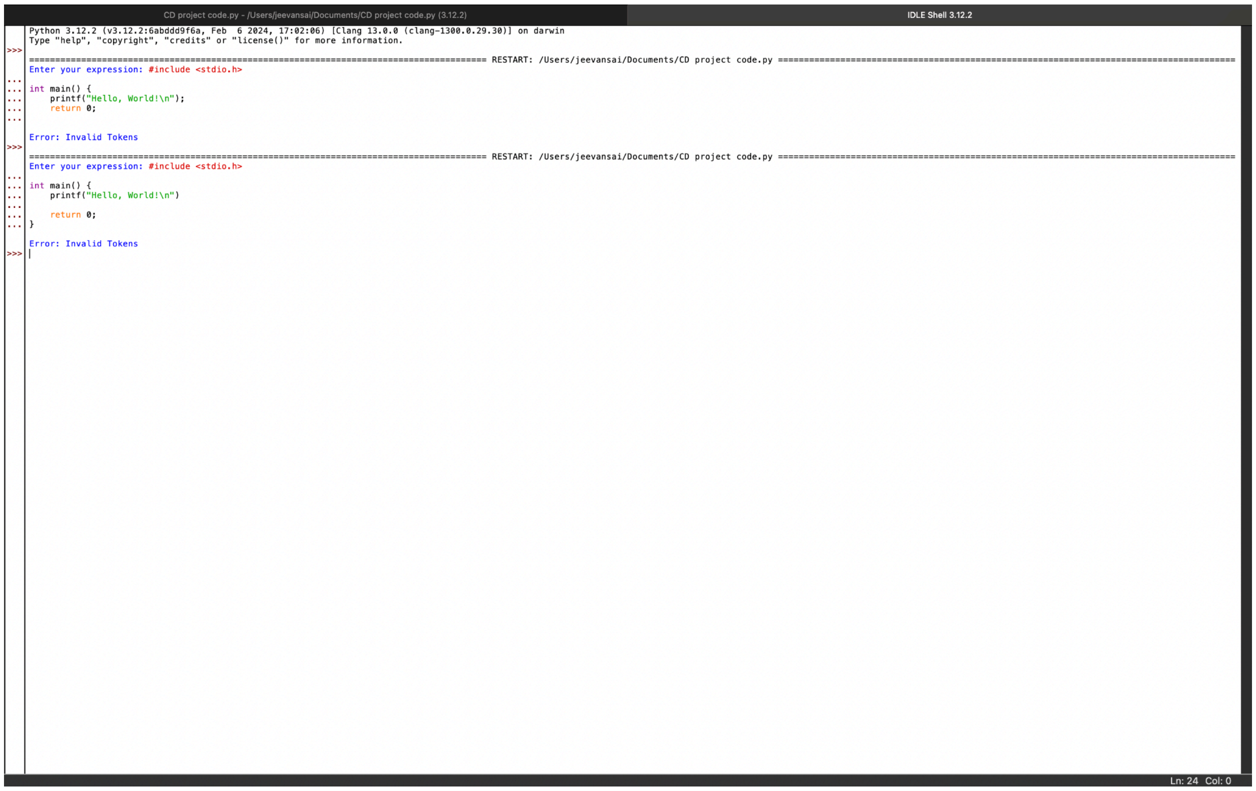
Ln: 34 Col: 0



CD project code.py - /Users/jeevansai/Documents/CD project code.py (3.12.2)   IDLE Shell 3.12.2

```
Python 3.12.2 (v3.12.2:6abddd9f6a, Feb  6 2024, 17:02:06) [Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========================================================================================== RESTART: /Users/jeevansai/Documents/CD project code.py ==========================================================================================
Enter your expression: #include <stdio.h>
...
... int main() {
...     printf("Hello, World!\n");
...     return 0;
...

Error: Invalid Tokens
>>>
========================================================================================== RESTART: /Users/jeevansai/Documents/CD project code.py ==========================================================================================
Enter your expression: #include <stdio.h>
...
... int main() {
...     printf("Hello, World!\n")
...
...     return 0;
... }

Error: Invalid Tokens
>>>
```

Ln: 24 Col: 0



```
CD project code.py - /Users/jeevansai/Documents/CD project code.py (3.12.2)          IDLE Shell 3.12.2
    Python 3.12.2 (v3.12.2:6abddd9f6a, Feb  6 2024, 17:02:06) [Clang 13.0.0 (clang-1300.0.29.30)] on darwin
    Type "help", "copyright", "credits" or "license()" for more information.
>>>
    ==================================================================================== RESTART: /Users/jeevansai/Documents/CD project code.py ====================================================================================
    Enter your expression: #include <stdio.h>
...
... int main() {
...     printf("Hello, World!\n");
...     return 0;
...

    Error: Invalid Tokens
>>>
    ==================================================================================== RESTART: /Users/jeevansai/Documents/CD project code.py ====================================================================================
    Enter your expression: #include <stdio.h>
...
... int main() {
...     printf("Hello, World!\n")
...
...     return 0;
... }

    Error: Invalid Tokens
>>> |
                                                                                                                                      Ln: 24  Col: 0
```

Conclusion:

In conclusion, the development of a syntax analyzer tailored to detect syntax errors and unexpected tokens within source code presents a significant advancement in software development practices. Through the implementation of robust error detection algorithms and user-friendly interfaces, the syntax analyzer empowers developers to identify and resolve syntax issues efficiently, thereby enhancing code quality and reliability. The integration of features such as real-time monitoring, collaboration tools, and trend analysis further augments the capabilities of the syntax analyzer, enabling users to gain valuable insights into their codebases and drive continuous improvement.

By providing a centralized platform for syntax analysis tasks and fostering collaboration among team members, the syntax analyzer contributes to streamlined development workflows and enhanced productivity. Moreover, its ability to adapt to evolving syntax standards and support multiple programming languages ensures its relevance and applicability across diverse software projects. Overall, the development of a syntax analyzer represents a pivotal step towards achieving code correctness, improving software reliability, and ultimately delivering better outcomes for developers and end-users alike