

Name: Aqsa Ishfaq

Reg No: Fa20-bcs-002

Submitted to: Sir Bilal Bukhari

Lab Terminal

Lab Terminal

**Question No # 1**

Write Brief Overview About Project?

**Answer:**

**Project Overview: Compiler Analysis with GUI**

**Objective:**

The project aims to create a simple compiler analysis tool with a graphical user interface (GUI) using Windows Forms in Visual Studio. The primary focus is on implementing lexical analysis, semantic analysis, and providing a user-friendly interface for input and output.

**Key Components:**

**Lexical Analysis (Tokenization):**

The code includes a method to tokenize input expressions using regular expressions. This step breaks down the input expression into a sequence of tokens.

**Semantic Analysis (Partial Implementation):**

The code begins the implementation of semantic analysis. However, the logic for parsing and analyzing expressions is incomplete. Further development is needed to handle the recursive descent parsing for the specified grammar.

**First and Follow Sets:**

The project incorporates the definition and display of First and Follow sets for non-terminals in a context-free grammar. These sets play a crucial role in parsing algorithms.

**GUI Design:**

The GUI is created using Windows Forms in Visual Studio. It includes controls such as TextBox for input, a Button for triggering the analysis, and a Label for displaying results.

**Input:**

Users can enter mathematical expressions in the provided TextBox.

**Analysis:**

Clicking the "Analyze" button triggers the analysis process, which includes tokenization, semantic analysis (incomplete), and the display of First and Follow sets.

**Output:**

Results, including First and Follow sets and analysis success/failure messages, are displayed in the Label.

**Project Status:**

The project is in progress, with the need for further development in the semantic analysis part to complete the parsing logic for the specified grammar.

**Next Steps:**

Implement the missing logic in the AnalyzeExpression, AnalyzeEPrime, AnalyzeT, AnalyzeTPrime, and AnalyzeF methods to achieve a fully functional parser.

**Potential Enhancements:**

Expand the project to handle more complex grammars and support additional compiler phases (e.g., syntax analysis, code generation).

**Question No # 2**

How Function works Step by Step?

**Answer:**

* **private void InitializeFirstAndFollowSets()**

This function initializes the firstSets and followSets dictionaries with rules for each non-terminal in your grammar. For example, it defines that the E non-terminal has the first set of {"(", "num"} and the follow set of {"$", ")"}.

* **private void analyzeButton\_Click(object sender, EventArgs e)**

This function is an event handler for the "Analyze" button click. It gets triggered when the user clicks the button. Inside this function, you:

Set **currentPositio**n to 0, indicating the start of the input.

Call **DisplayFirstAndFollowSets** to show the First and Follow sets in the result label.

Call **AnalyzeExpression** to start the analysis of the mathematical expression.

* **private void DisplayFirstAndFollowSets()**

This function displays the First and Follow sets for each non-terminal in the result label. It iterates through the firstSets and followSets dictionaries and constructs a string representation to display in the GUI.

* **private void AnalyzeExpression()**

This function is a starting point for the analysis of the mathematical expression. It gets called from the analyzeButton\_Click method. However, the logic inside this method is not implemented yet (it throws a NotImplementedException). This is the part where you need to complete the recursive descent parsing logic to analyze the expression based on the grammar rules**.**

* **private void AnalyzeEPrime()**

This function represents the analysis of the non-terminal E' in your grammar. It checks for the presence of the "+" symbol, indicating addition. If present, it recursively analyzes the expression after the addition. If not, it checks for the end of input or closing parenthesis.

* **private void AnalyzeT()**

This function represents the analysis of the non-terminal T in your grammar. It calls AnalyzeF to analyze the factor and then calls AnalyzeTPrime to handle additional factors and multiplication.

* **private void AnalyzeTPrime()**

This function represents the analysis of the non-terminal T' in your grammar. It checks for the "\*" symbol, indicating multiplication. If present, it recursively analyzes the expression after the multiplication. If not, it checks for the end of input, closing parenthesis, or addition.

* **private void AnalyzeF()**

This function represents the analysis of the non-terminal F in your grammar. It checks whether the current token is an opening parenthesis, a number, or any other unexpected token. It then recursively analyzes the expression inside the parentheses or consumes the number accordingly.

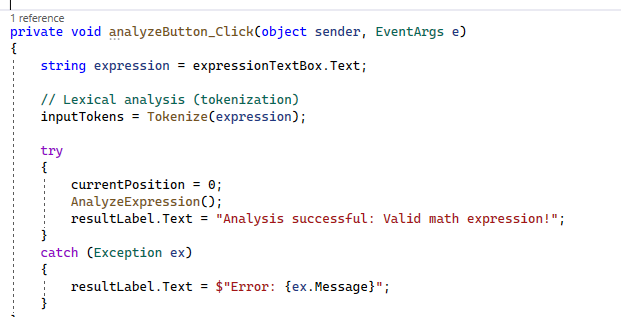
**Conclusion**

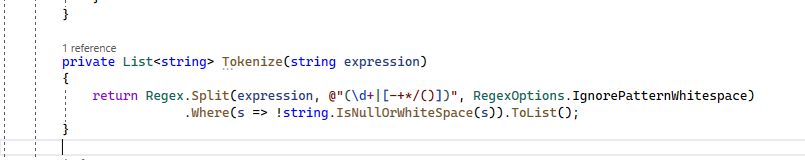
The key idea here is that the functions work together in a recursive descent parsing manner, reflecting the grammar rules. Each function is responsible for analyzing a specific non-terminal or part of the grammar.

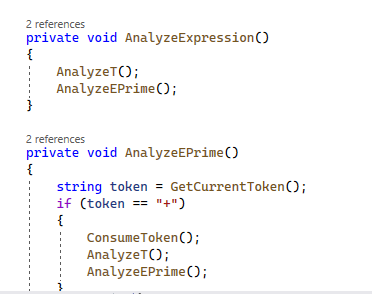
**Question No # 3**

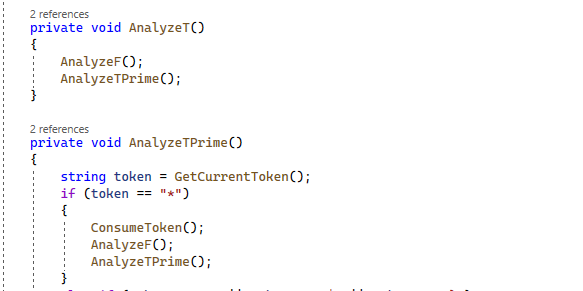
**Functionalities along with screenshots (function code +output)?**

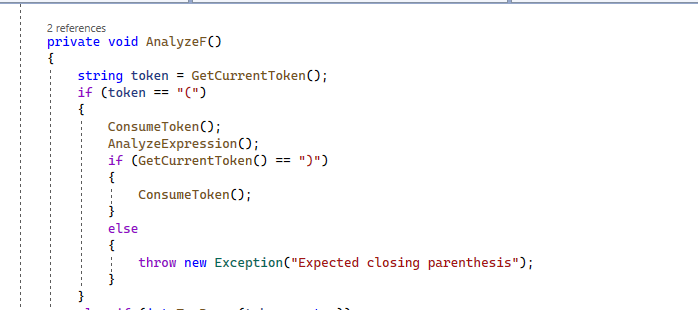
**Answer:**





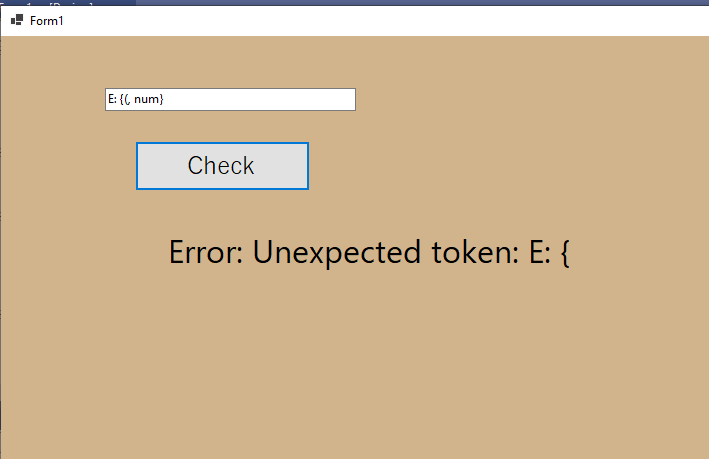


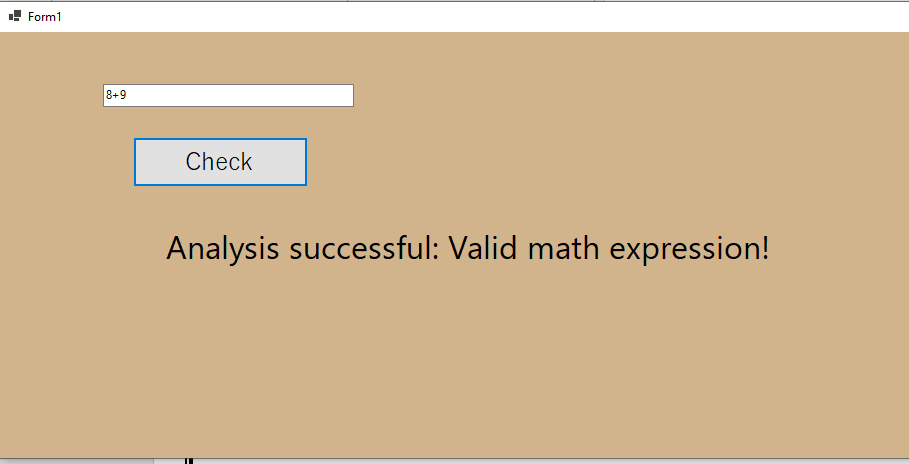


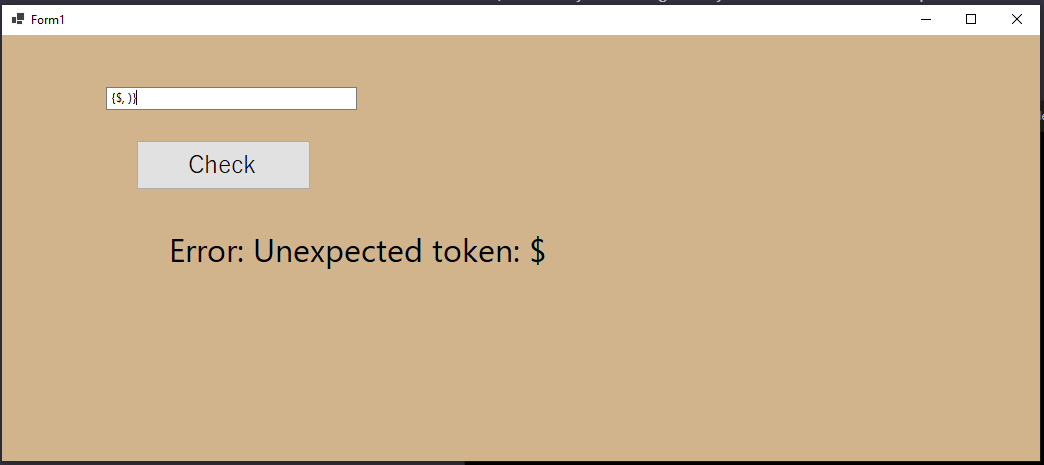


**Question No #4**

Output:







**Question No # 5**

**Problems Facing During Project?**

**Answer:**

* **Challenges:**

Ambiguity in error messages.

Handling multiple errors efficiently.

Detecting and reporting semantic errors.

* **Strategies:**

Provide detailed and clear error messages.

Implement graceful error recovery mechanisms.

Focus on user-friendly error output.

* **Integration of Compiler Phases:**
* **Challenges:**

Efficient data flow between phases.

Consistency in code representation.

Handling changes in one phase affecting others.

* **Strategies:**

Use standardized data structures for code representation.

Define well-defined interfaces between phases.

Rigorously test interfaces to ensure seamless integration.