COMSATS UNIVERSITY
ISLMABAD, ATTOCK CAMPUS

NAME: Hafsa Hanif (FA20-BCS-035)

Syeda Tooba Aftab (FA20-BCS-025)

SECTION: BCS 7A

SUBJECT: Compiler Construction

SUBM TO: Mr. Bilal Haider

DATE: 28-12-2023

Project Report

1. Introduction:

The project is a Windows Forms application developed in C# that seems to be designed for educational purposes, focusing on compiler-related concepts such as semantic analysis, lexical analysis, and the computation of First/Follow sets for a set of context-free grammar productions.

Components:

I. Semantic Analysis:

The PerformSemanticAnalysis method processes an arithmetic expression, checking for semantic errors (such as insufficient operands for operators) and providing a result indicating whether the expression is semantically valid.

II. Lexical Analysis:

The CreateToken method classifies input words into different token types (e.g., integers, strings, keywords, identifiers, operators, parentheses, unknown). The DisplayTokens method displays the generated tokens.

III. User Interface:

The user interface is implemented using Windows Forms, featuring buttons (ADD, Generate, Click for lexical analysis, Tokenize Button) and text boxes (Non_Terminal, Production, Start Symbol, Input, Output, Grammer Display). The buttons trigger various actions, such as semantic analysis, lexical analysis and recursive descent parser.

2. Function Details:

I. Semantic Analysis (PerformSemanticAnalysis):

Parses the input expression into tokens.
Iterates through the tokens, identifying operators and operands.
Performs a basic check for the presence of enough operands for each operator.
Returns a result indicating whether the expression is semantically valid.

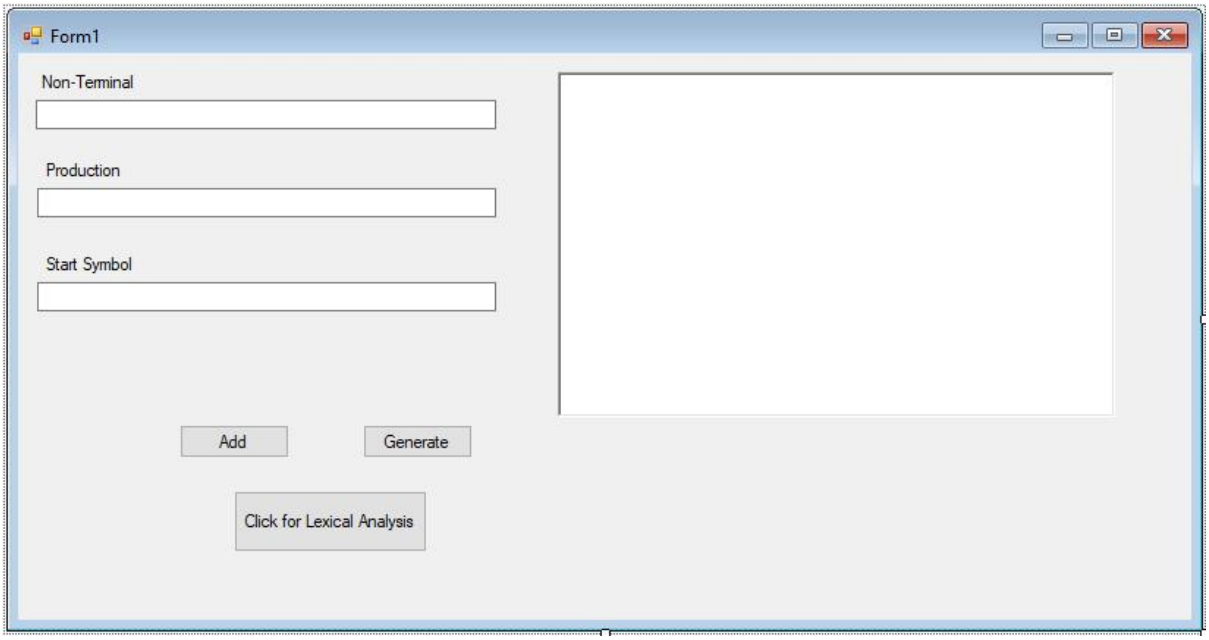
CODE:

```
public partial class Form1 : Form
{
    private Dictionary<string, string> grammarRules;

    public Form1()
    {
        InitializeComponent();
        grammarRules = new Dictionary<string, string>();
    }

    private void addBtn_Click(object sender, EventArgs e)
    {
        string nonTerminal = nonTerminalTextBox.Text.Trim();
        string production = productionTextBox.Text.Trim();
```

```
            if (!string.IsNullOrEmpty(nonTerminal)
&& !string.IsNullOrEmpty(production))
            {
                grammarRules[nonTerminal] = production;
                UpdateGrammarDisplay();
            }
            else
            {
                MessageBox.Show("Please enter both non-terminal and production.");
            }
        }

        private void generateBtn_Click(object sender, EventArgs e)
        {
            if (grammarRules.Count > 0)
            {
                string startSymbol = startSymbolTextBox.Text.Trim();

                if (!string.IsNullOrEmpty(startSymbol) &&
grammarRules.ContainsKey(startSymbol))
                {
                    string generatedParser = GenerateParser(startSymbol);
                    // Display or save the generated parser code
                    MessageBox.Show("Generated parser:\n\n" + generatedParser);
                }
                else
                {
                    MessageBox.Show("Please enter a valid start symbol.");
                }
            }
            else
            {
                MessageBox.Show("Please define grammar rules.");
            }
        }
        private string GenerateParser(string startSymbol)
```

```
        {
            // TODO: Implement parser generation logic based on grammarRules and
startSymbol
            // This is a placeholder, you need to replace this with your parser
generation logic
            StringBuilder parserCode = new StringBuilder();

            parserCode.AppendLine($"// Generated Parser for {startSymbol}");
            parserCode.AppendLine("public class Parser {");
            parserCode.AppendLine("\tpublic void Parse() {");
            parserCode.AppendLine("\t\t// Parse logic goes here");
            parserCode.AppendLine("\t}");
            parserCode.AppendLine("}");

            return parserCode.ToString();
        }

        private void UpdateGrammarDisplay()
        {
            grammarDisplayTextBox.Clear();

            foreach (var rule in grammarRules)
            {
                grammarDisplayTextBox.AppendText($"{rule.Key} -> {rule.Value}\n");
            }
        }

        private void form2_Click(object sender, EventArgs e)
        {
            this.Hide();
            LexicalAnalysis lexicalForm = new LexicalAnalysis();

            // Show LexicalAnalysis form
            lexicalForm.Show();
        }
    }
```

Output:



II. Lexical Analysis (CreateToken, DisplayTokens):

CreateToken classifies input words into different token types based on specific criteria.
DisplayTokens method displays the generated tokens in a user interface text box.

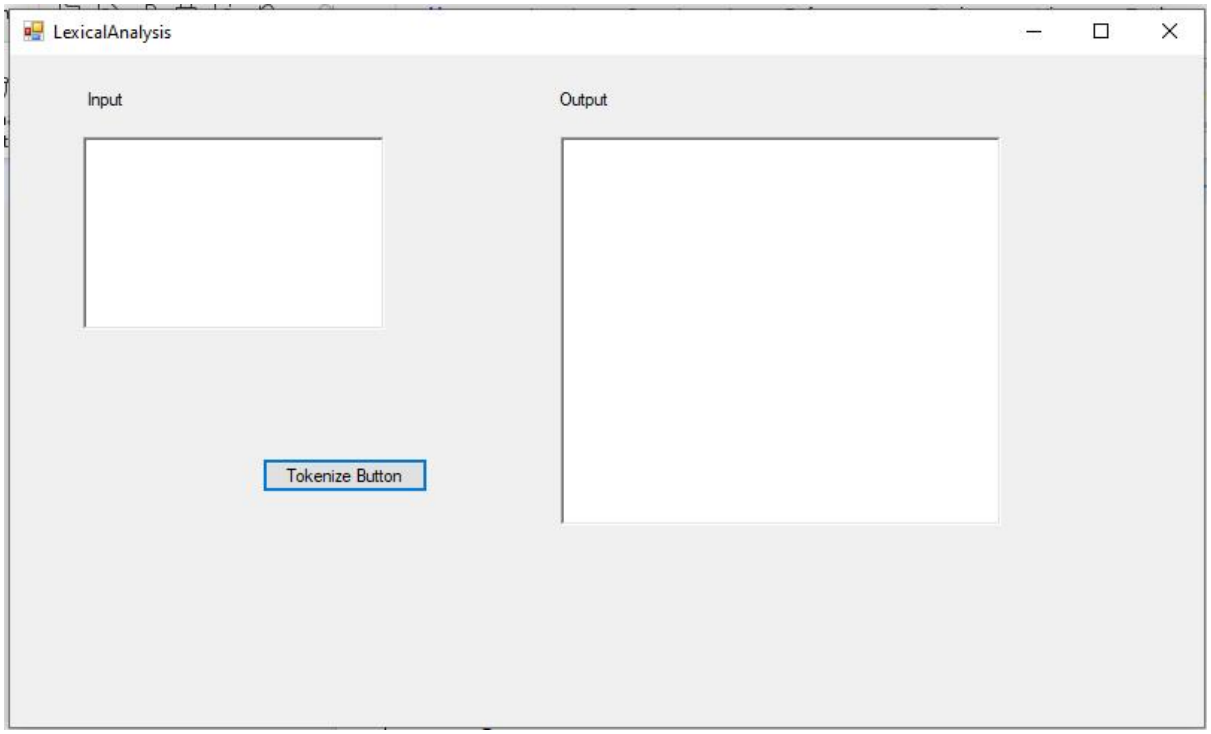
CODE:

```
public LexicalAnalysis()
{
    InitializeComponent();
    grammarRules = new Dictionary<string, string>();
}

private void TokenizeBtn_Click(object sender, EventArgs e)
{
    string inputText = inputTextBox.Text.Trim(); // Get input from parser
form

    List<Token> tokens = TokenizeInput(inputText);

    DisplayTokens(tokens);
}
```

```
        private List<Token> TokenizeInput(string input)
        {
            // Tokenizing logic (basic example)
            List<Token> tokens = new List<Token>();

            // Define regex patterns for tokens
            Regex identifierPattern = new Regex(@"[a-zA-Z_]\w*");
            Regex numberPattern = new Regex(@"\d+");

            // Split input into tokens based on patterns
            string[] words = input.Split(new[] { ' ', '\t', '\n', '\r' },
StringSplitOptions.RemoveEmptyEntries);
            foreach (var word in words)
            {
                if (identifierPattern.IsMatch(word))
                {
                    tokens.Add(new Token(TokenType.Identifier, word));
                }
                else if (numberPattern.IsMatch(word))
                {
                    tokens.Add(new Token(TokenType.Number, word));
                }
                else
                {
                    tokens.Add(new Token(TokenType.Unknown, word));
                }
            }

            return tokens;
        }

        private void DisplayTokens(List<Token> tokens)
        {
            tokenDisplayTextBox.Clear();
```

```
            foreach (var token in tokens)
            {
                tokenDisplayTextBox.AppendText($"{token.Type}: {token.Value}\n");
            }
        }
    }
    public enum TokenType
    {
        Identifier,
        Number,
        Unknown // for tokens that don't match known patterns
    }

    public class Token
    {
        public TokenType Type { get; }
        public string Value { get; }

        public Token(TokenType type, string value)
        {
            Type = type;
            Value = value;
        }
    }
```

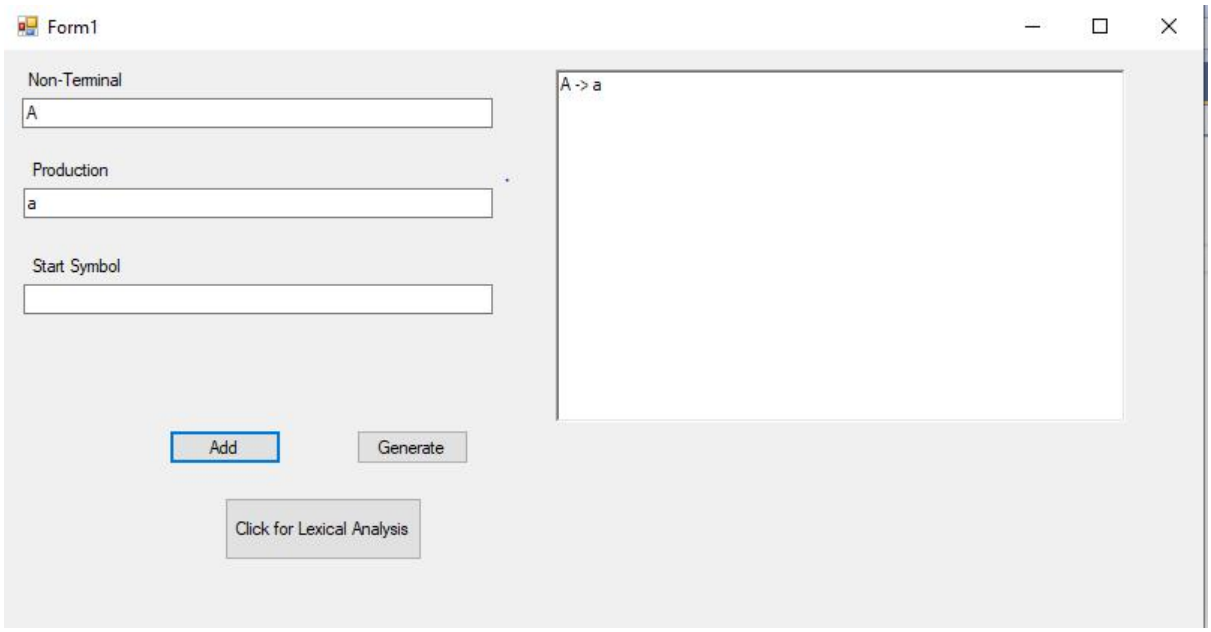
Output:



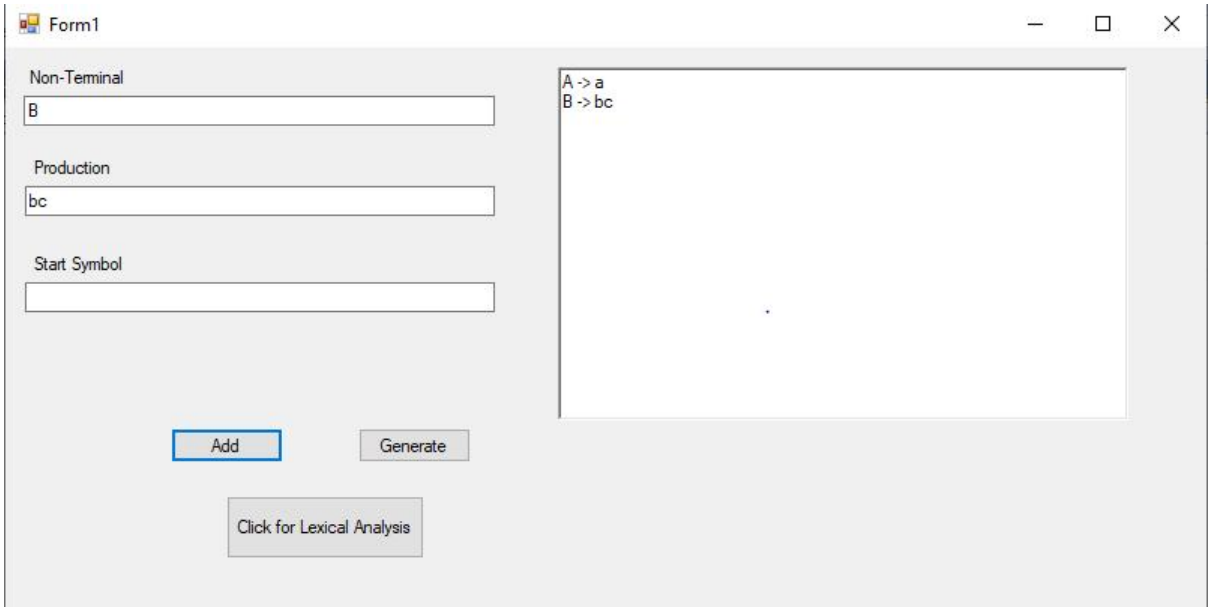
3. Input + output Screenshot

I. Semantic Analyze

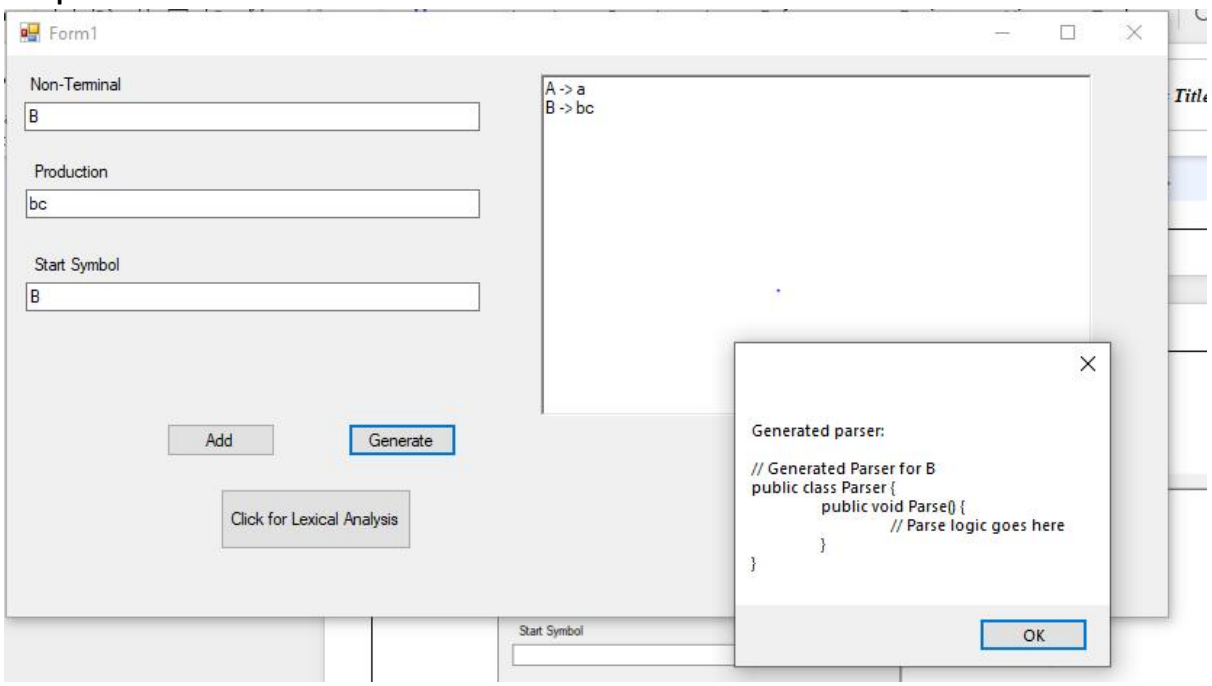
Input:



By clicking on ADD Button, the terminals and non-terminals given by user is added 1 by 1 and shown in the richTextbox.

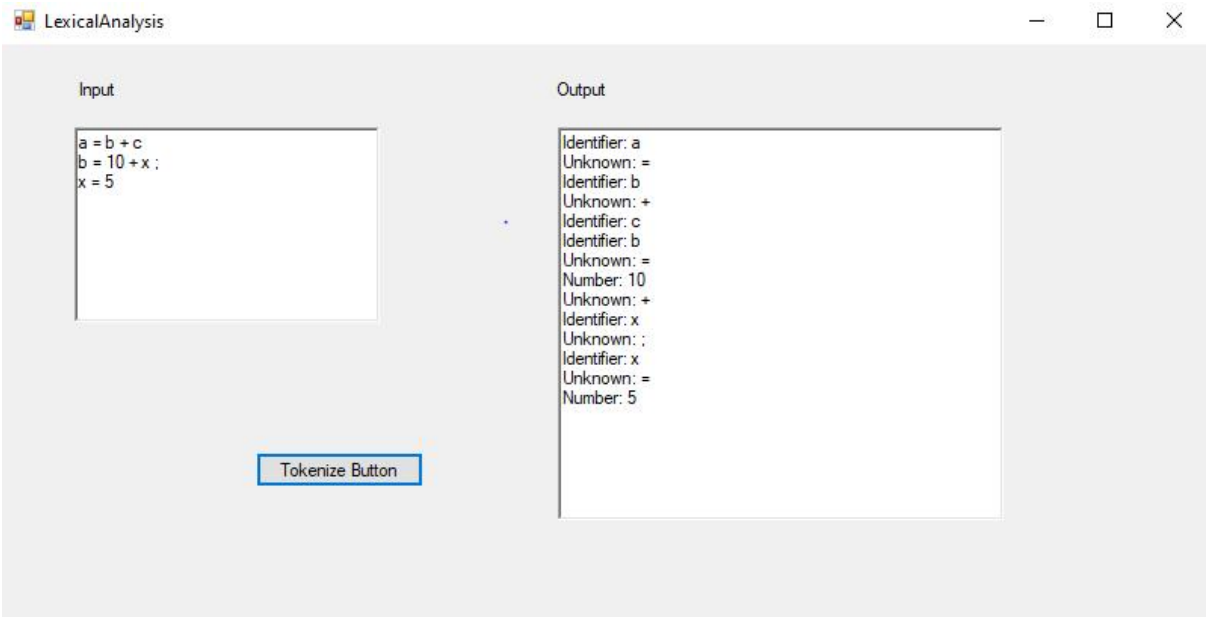


Output:



II. Lexical Analyze

Input / Output:



4. How function works step by step:

In the provided project involving a Parser Generator with Lexical Analysis functionality, the functions work together in a sequence to achieve tokenization and parser generation. Here's a step-by-step explanation of how these functions might work:

1. MainForm Interaction:

User Input: The user interacts with the `MainForm` GUI to define grammar rules and initiate parsing or lexical analysis.

2. Lexical Analysis:

a. LexicalAnalysisForm:

Tokenization Button Clicked: The user clicks a button (e.g., "Tokenize") in the `LexicalAnalysisForm`.

Invoke Tokenization Logic: The `TokenizeBtn_Click` event handler is triggered.

Input Extraction: Retrieves the input text from a text box (`inputTextBox.Text.Trim()`).

b. Tokenization Logic:

Pattern Matching: Uses regular expressions or other patterns (`Regex`) to match tokens like identifiers, numbers, etc., in the input text.

Token Generation: Iterates through the input text, identifies tokens based on patterns, and creates `Token` objects (`List<Token>`) representing different token types (Identifier, Number, Unknown).

Display Tokens: Calls the `DisplayTokens` method to show the generated tokens in a text box (`tokenDisplayTextBox`).

3. Parser Generation:

a. MainForm:

Grammar Definition: The user defines grammar rules in the `MainForm`.

Initiate Parser Generation: Initiates parser generation (not fully implemented in the provided code).

b. Parser Generation Logic (Not Fully Implemented):

Start Symbol Definition: Retrieves the user-defined start symbol from the `MainForm`.

Generate Parser Logic: Utilizes algorithms (LL, LR, recursive descent, etc.) to generate parsing logic based on the provided grammar rules and start symbol.

Output Parser Code: Constructs and outputs the parser code (not fully implemented in the provided code).

Key Points:

Interaction Flow: User interaction triggers various functions/events in different forms.

Tokenization: Tokenization logic tokenizes the input text based on defined patterns and displays the tokens.

Parser Generation: Not fully implemented in the provided code but would involve using grammar rules to generate parser logic.

The provided code showcases a basic structure and interaction flow. To complete the functionality, you would need to implement the parser generation logic based on the provided grammar rules and possibly enhance the tokenization logic for a wider range of language constructs.

5. Challenges Faced:

While the code provided is functional, there are potential challenges and considerations:

I. **Simplifications:**

The code assumes a simplified scenario with basic arithmetic expressions and a limited set of context-free grammar productions. In real-world scenarios, handling a full programming language grammar would be much more complex.

II. **Error Handling:**

The error handling in the semantic analysis is basic and may not cover all possible cases. In a complete compiler, more sophisticated error detection and reporting mechanisms would be required.

III. **User Interface Design:**

Designing an effective and user-friendly interface, especially for educational purposes, can be challenging. Clarity and simplicity are crucial, especially when dealing with complex concepts.

IV. **Grammar Handling:**

Managing and parsing context-free grammar productions can be intricate. The provided code handles a basic case, but in a real compiler, the grammar might be more extensive and diverse.

V. **Concurrency Handling:**

The code includes threading considerations (InvokeRequired) for updating the user interface from background threads. Managing concurrency is crucial in GUI applications.