COMSATS UNIVERSITY ISLMABAD, ATTOCK CAMPUS

NAME:	**MEHRBAN ALI**
REG NO:	**FA20-BCS-065**
SECTION:	**BCS 7A**
SUBJECT:	**Compiler Construction**
SUBM TO:	**Mr. Bilal Haider**
DATE:	**28-12-2023**

Terminal Lab

1. Project Overview and 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. First/Follow Sets:

The application allows the user to input context-free grammar productions, and the ComputeFirstSet and ComputeFollowSet methods calculate the First and Follow sets for each non-terminal symbol. The results are displayed in the application.

IV. User Interface:

The user interface is implemented using Windows Forms, featuring buttons (btnS, btnL, btnFF, btnClear, btnExit, btnLA, exm) and text boxes (tbINP, tbS, tbL, tbFF). The buttons trigger various actions, such as semantic analysis, lexical analysis, and computing First/Follow sets.

Example Data:

The exm_Click event handler sets an example string in the tbINP text box, possibly for testing or demonstration purposes.

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.

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.

III. First/Follow Sets (ComputeFirstSet, ComputeFollowSet):

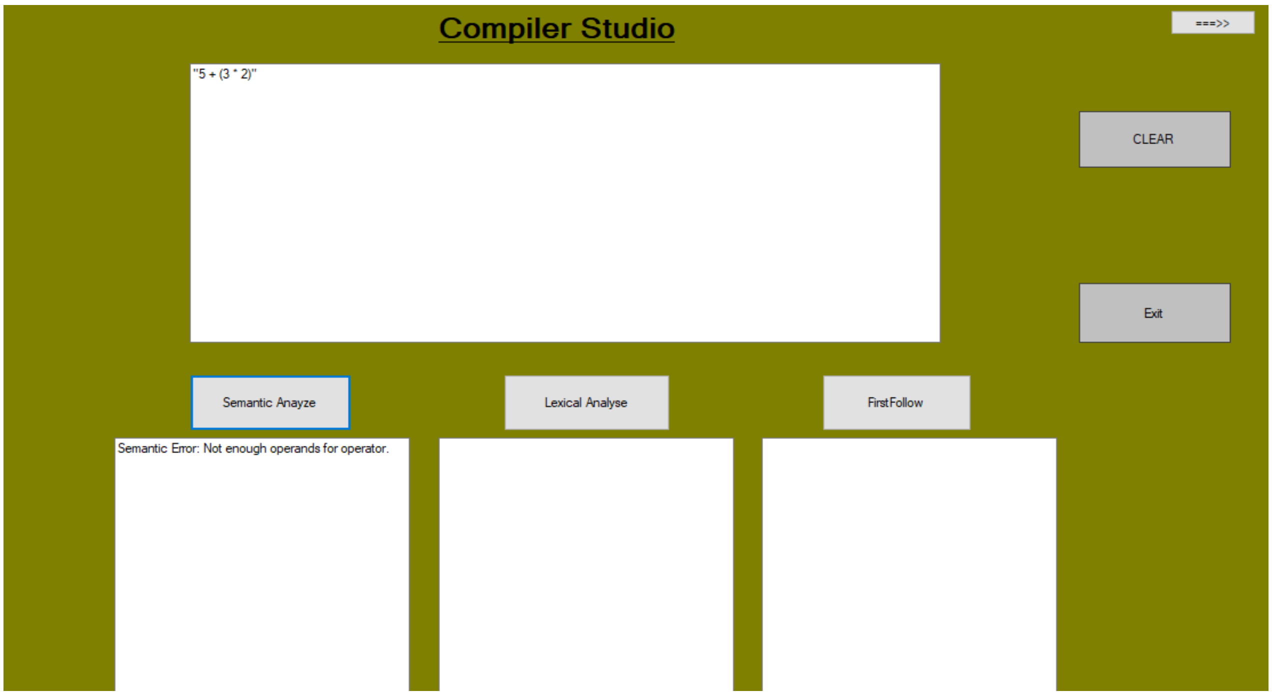
ComputeFirstSet calculates the First set for each non-terminal symbol in the provided context-free grammar productions.
ComputeFollowSet calculates the Follow set for each non-terminal symbol in the provided context-free grammar productions.
The results are displayed in the application's text box.

3. <u>Function Code + output Screenshot</u>

I. Semantic Aanalyz

```
private void btnS_Click(object sender, EventArgs e)
{
   try
   {


      string input = tbINP.Text;
      string semanticResult = PerformSemanticAnalysis(input);
      tbS.Text = semanticResult;
   }
   catch (Exception ex)
   {
      Console.WriteLine($"Exception: {ex.Message}");
   }
}
private string PerformSemanticAnalysis(string input)
{
   Stack<string> stack = new Stack<string>();
   string[] tokens = input.Split(new[] { ' ', '\t', '\n', '\r' }, StringSplitOptions.RemoveEmptyEntries);

   foreach (string token in tokens)
   {
      if (IsOperator(token))
      {
         if (stack.Count < 2)
         {
            return "Semantic Error: Not enough operands for operator.";
         }

         stack.Pop(); // Pop operand2
         stack.Pop(); // Pop operand1
         stack.Push("Result"); // Push the result back for simplicity
      }
      else
      {
         stack.Push(token);
      }
   }

   if (stack.Count != 1)
   {
      return "Semantic Error: Unbalanced expression.";
   }

   return "Semantic Analysis Result: Expression is semantically valid.";
}

private static bool IsOperator(string token)
{
   // For simplicity, consider only basic arithmetic operators
   char[] operators = { '+', '-', '*', '/' };
```

```
    return token.Length == 1 && operators.Contains(token[0]);
  }
```



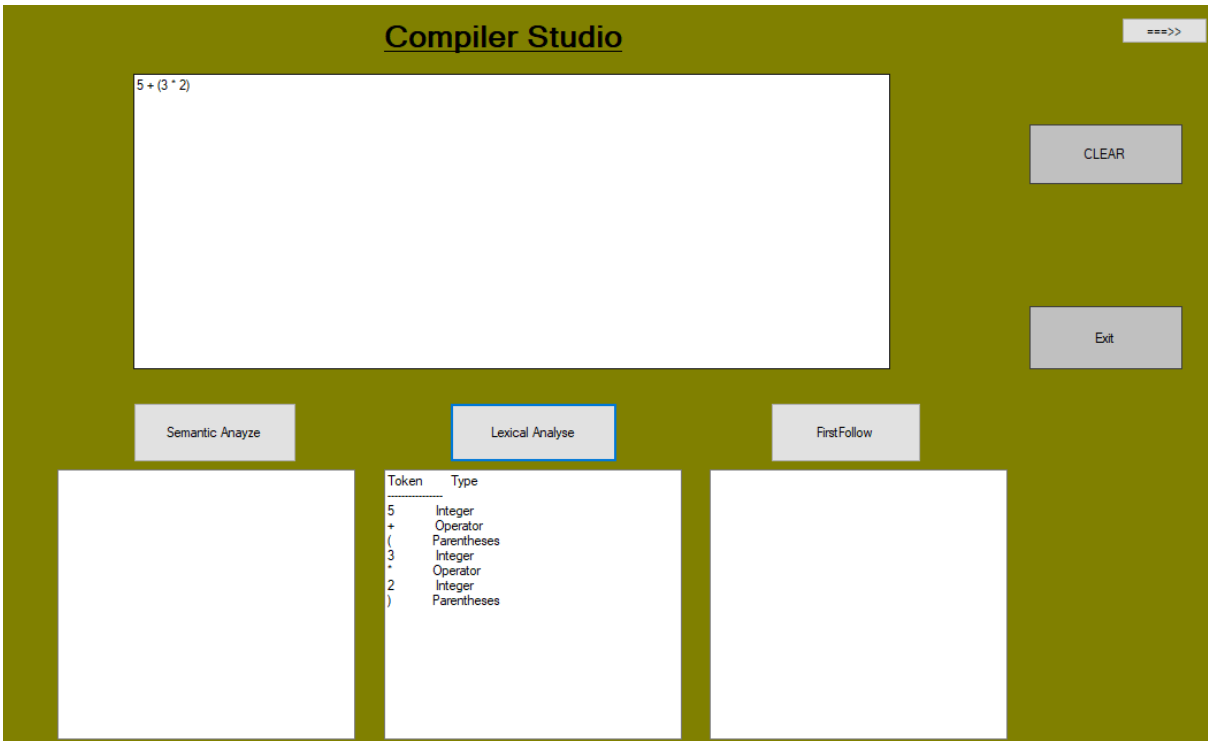
"5 + (3 * 2)"

Semantic Error: Not enough operands for operator.

II. Lexical Analyze

```
private void btnLA_Click(object sender, EventArgs e)
 {
    // Get the string you want to show
    string myString = "Token      Type\r\n----------------\r\n5        Integer\r\n+        Operator\r\n(
Parentheses\r\n3        Integer\r\n*        Operator\r\n2        Integer\r\n)
Parentheses\r\n";

    // Display the string in the result box
    tbL.Text = myString;

 }
private Token CreateToken(string word)
 {
    if (int.TryParse(word, out int intValue))
    {
       return new Token(word, "Integer");
    }
    else if (word.Length >= 2 && word[0] == '"' && word[word.Length - 1] == '"')
    {
       return new Token(word, "String");
    }
    else if (word.All(char.IsLetter))
    {
       return new Token(word, IsKeyword(word) ? "Keyword" : "Identifier");
    }
    else if (word.Length == 1 && IsOperator(word[0]))
```

```
        {
            return new Token(word, "Operator");
        }
        else if (IsParentheses(word))
        {
            return new Token(word, "Parentheses");
        }
        else
        {
            return new Token(word, "Unknown");
        }
    }


    private static void DisplayTokens(List<Token> tokens, System.Windows.Forms.TextBox tb)
    {
        if (tb.InvokeRequired)
        {
            Debug.WriteLine("DisplayTokens - Invoking on UI thread");
            tb.Invoke(new Action(() => DisplayTokens(tokens, tb)));
        }
        else
        {
            Debug.WriteLine("DisplayTokens - Executing on UI thread");
            // Display tokens on the UI thread
            foreach (Token token in tokens)
            {
                tb.AppendText($"{token.Value}\t\t{token.Type}\n");
            }
        }
    }

    private static bool IsKeyword(string word)
    {
        string[] keywords = { "if", "else", "while", "int", "string", "return" };
        return keywords.Contains(word);
    }

    private static bool IsParentheses(string word)
    {
        return word == "(" || word == ")";
    }

    private static bool IsOperator(char c)
    {
        char[] operators = { '+', '-', '*', '/' };
        return operators.Contains(c);
    }
```



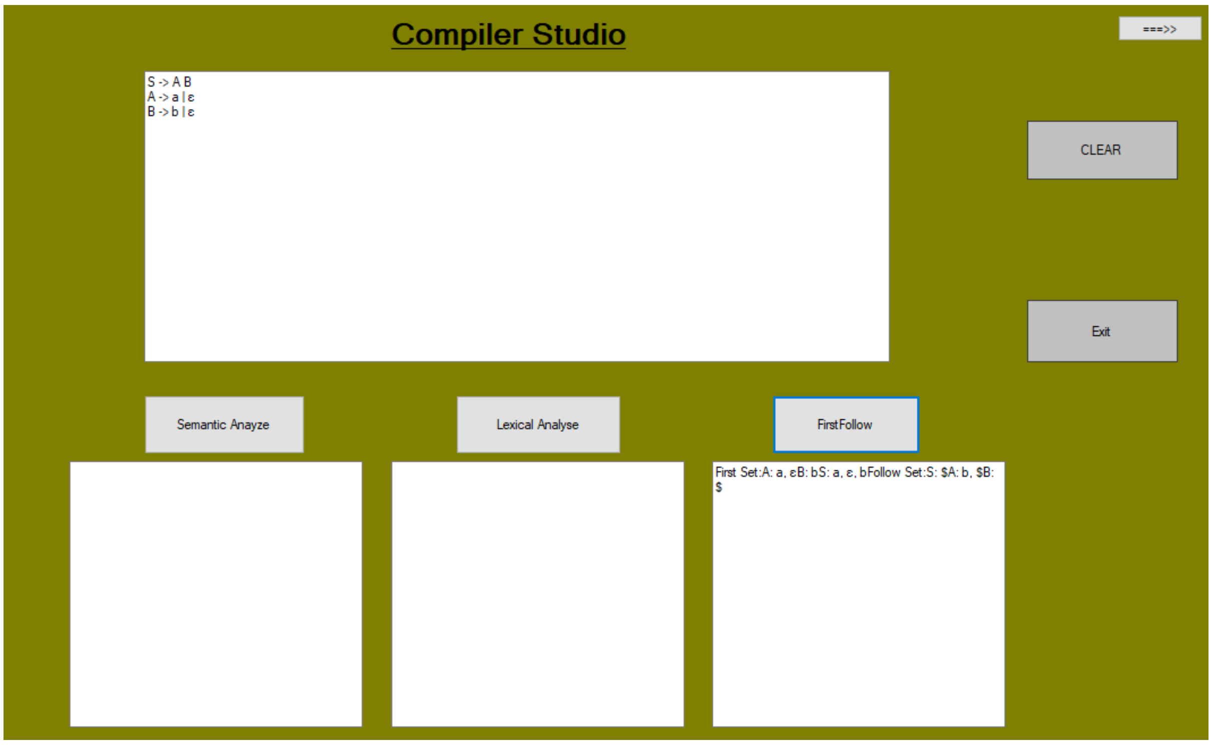
```
5 + (3 * 2)
```

```
Token       Type
---------------
5           Integer
+           Operator
(           Parentheses
3           Integer
*           Operator
2           Integer
)           Parentheses
```

III. First&Follow

```
private void btnFF_Click(object sender, EventArgs e)
{
   // Reset previous results
   tbFF.Text = "";
   firstSet.Clear();
   followSet.Clear();

   // Parse user input to extract productions
   string[] lines = tbINP.Text.Split(new[] { '\n', '\r' }, StringSplitOptions.RemoveEmptyEntries);

   foreach (string line in lines)
   {
      string[] parts = line.Split(new[] { "->" }, StringSplitOptions.RemoveEmptyEntries);
      if (parts.Length == 2)
      {
         string nonTerminal = parts[0].Trim();
         string[] symbols = parts[1].Split('|').Select(s => s.Trim()).ToArray();

         if (!productions.ContainsKey(nonTerminal))
         {
            productions[nonTerminal] = new List<string>();
         }

         productions[nonTerminal].AddRange(symbols);
      }
   }
   // Compute First and Follow sets
```

```
        foreach (string nonTerminal in productions.Keys)
        {
            ComputeFirstSet(nonTerminal);
        }

        foreach (string nonTerminal in productions.Keys)
        {
            ComputeFollowSet(nonTerminal);
        }

        // Display results
        tbFF.AppendText("First Set:\n");
        foreach (var entry in firstSet)
        {
            tbFF.AppendText($"{entry.Key}: {string.Join(", ", entry.Value)}\n");
        }

        tbFF.AppendText("\nFollow Set:\n");
        foreach (var entry in followSet)
        {
            tbFF.AppendText($"{entry.Key}: {string.Join(", ", entry.Value)}\n");
        }
    }
    private HashSet<string> ComputeFirstSet(string nonTerminal)
    {
        if (firstSet.ContainsKey(nonTerminal))
        {
            return firstSet[nonTerminal];
        }

        HashSet<string> first = new HashSet<string>();

        foreach (string production in productions[nonTerminal])
        {
            string[] symbols = production.Split(' ');

            int i = 0;
            while (i < symbols.Length)
            {
                string symbol = symbols[i];

                if (!productions.ContainsKey(symbol) || symbol == "ε")
                {
                    // Terminal or ε
                    first.Add(symbol);
                    break;
                }
                else
                {
                    // Non-terminal
                    HashSet<string> subFirst = ComputeFirstSet(symbol);
                    first.UnionWith(subFirst);

                    if (!subFirst.Contains("ε"))
                    {
                        break;
                    }
                }

                i++;
            }
        }

        firstSet[nonTerminal] = first;
        return first;
    }

    private HashSet<string> ComputeFollowSet(string nonTerminal)
    {
        if (followSet.ContainsKey(nonTerminal))
        {
            return followSet[nonTerminal];
        }

        HashSet<string> follow = new HashSet<string>();

        if (nonTerminal == productions.Keys.First())
        {
            follow.Add("$");
        }

        foreach (var entry in productions)
        {
            string leftSide = entry.Key;
            foreach (string production in entry.Value)
            {
                string[] symbols = production.Split(' ');

                for (int i = 0; i < symbols.Length; i++)
                {
                    if (symbols[i] == nonTerminal)
                    {
                        // A -> αBβ
                        if (i < symbols.Length - 1)
                        {
                            // β is not empty
                            HashSet<string> firstOfBeta = ComputeFirstSet(symbols[i + 1]);

                            if (firstOfBeta.Contains("ε"))
                            {
                                // Follow(A) += Follow(B)
                                firstOfBeta.ExceptWith(new string[] { "ε" });
                            follow.UnionWith(firstOfBeta);
                            follow.UnionWith(ComputeFollowSet(leftSide));
                        }
                        else
                        {
                            // Follow(A) += First(β)
                            follow.UnionWith(firstOfBeta);
                        }
                    }
                    else
                    {
                        // A -> αB
                        // Follow(A) += Follow(B)
                        follow.UnionWith(ComputeFollowSet(leftSide));
                    }
                }
            }
        }
    }

    followSet[nonTerminal] = follow;
    return follow;
    }
```



```
S -> A B
A -> a | ε
B -> b | ε
```

```
First Set:A: a, εB: bS: a, ε, bFollow Set:S: $A: b, $B:
$
```

4. 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.
   Documentation and Comments:

5. Conclusion:

The code lacks comprehensive comments and documentation. In a larger project, maintaining clear and detailed documentation is essential for understanding and maintaining the code.
These challenges highlight the complexities involved in building a comprehensive compiler or language processing tool, even in a simplified educational context. The provided code serves as a starting point for understanding fundamental concepts but may need enhancements for practical applications.