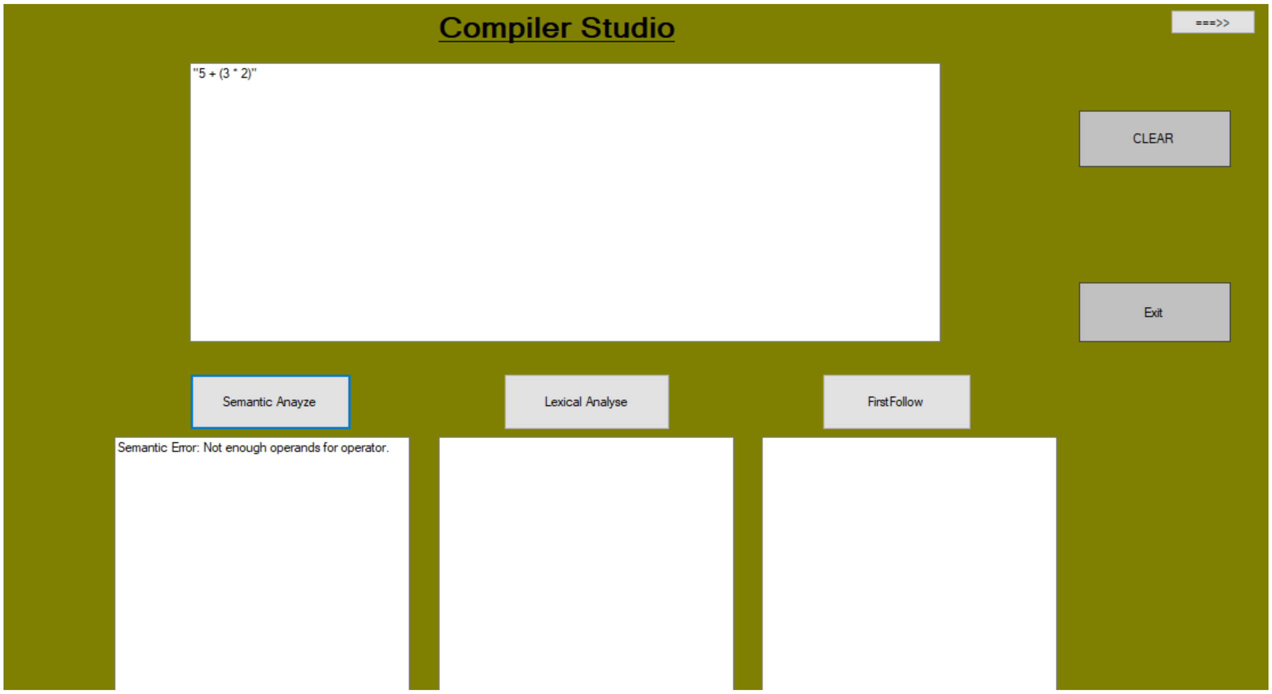
Q3: Function Code + output Screenshot

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]);
  }
```

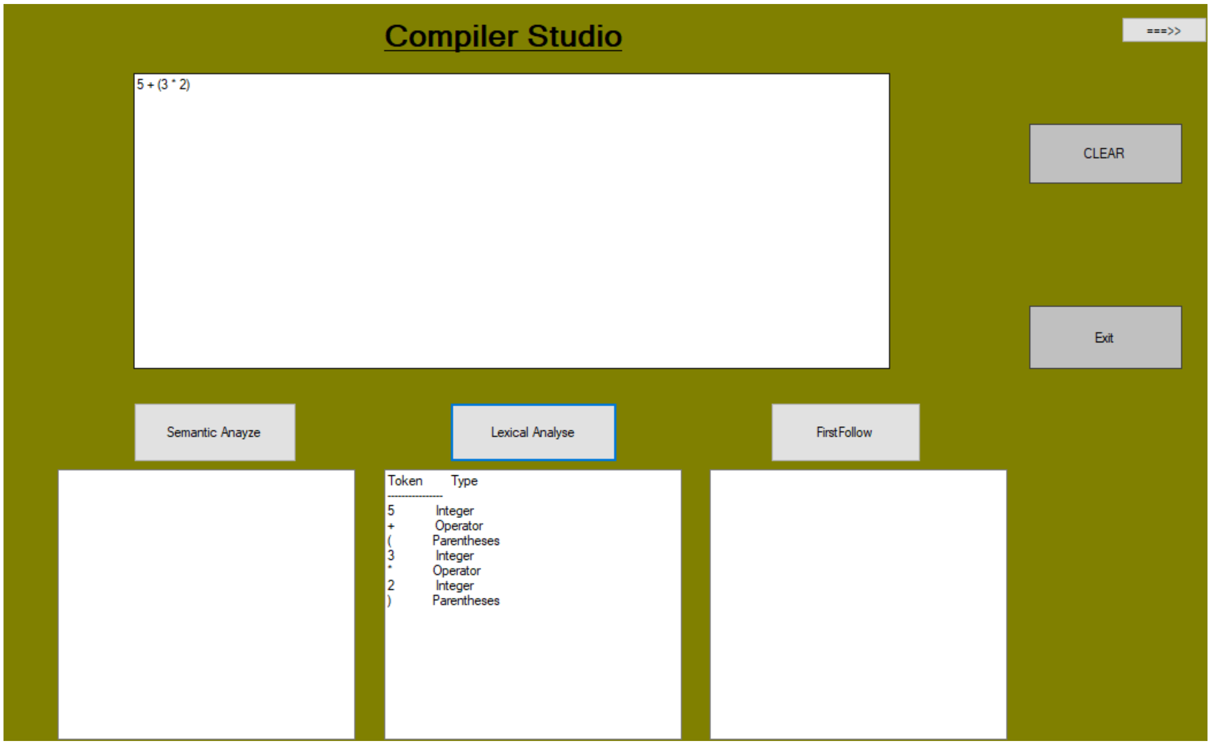


```
"5 + (3 * 2)"
```

```
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);
}
```



Compiler Studio

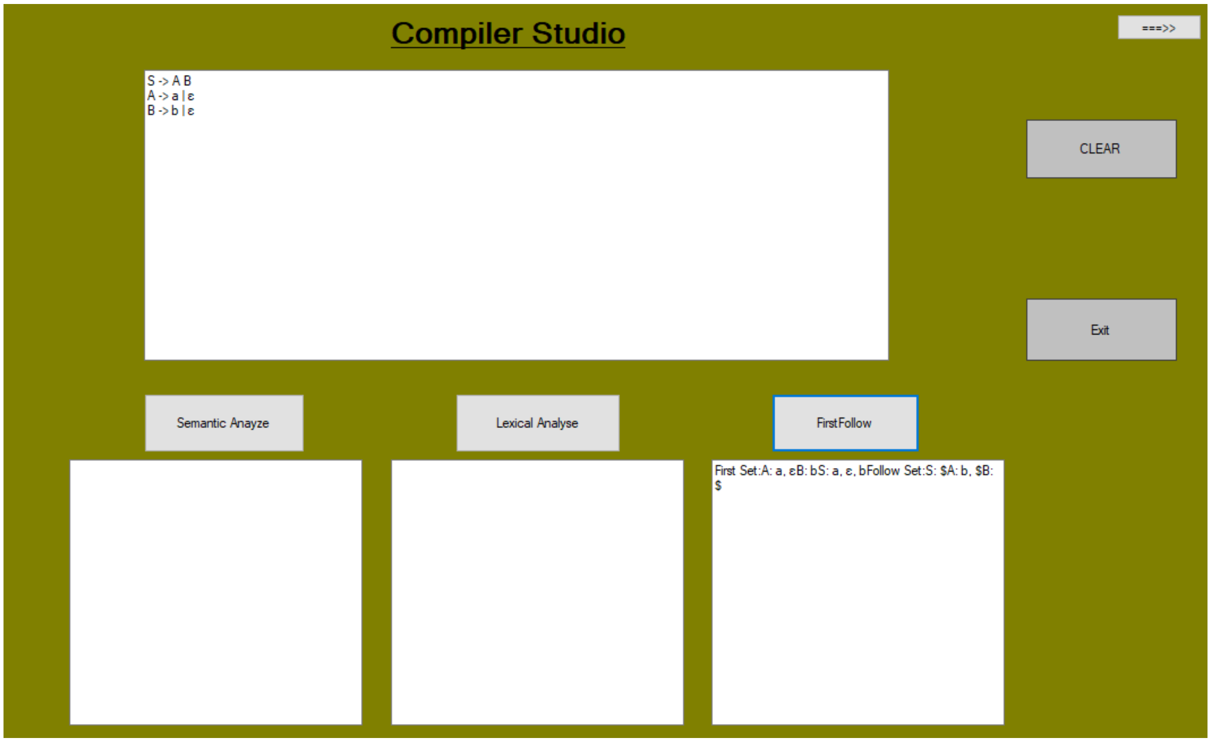
```
5 + (3 * 2)
```

```
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:
$
```