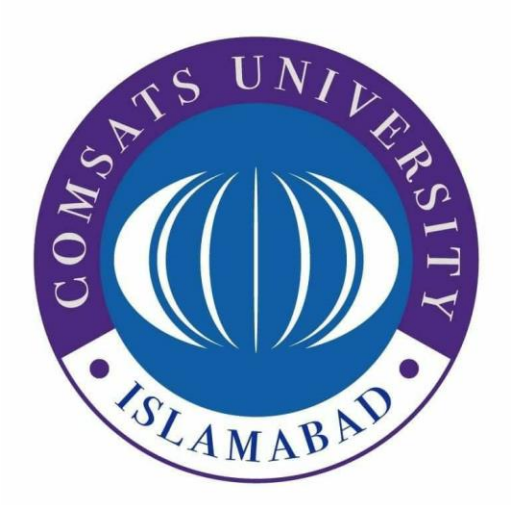
Final Project



Group Members

Name: Muhammad Ahmed

Reg No: FA20-BCS-036

Name: Alina Shah

Reg No: FA20-BCS-030

Submitted to: Sir Bilal Haider Bukhari

Dated: 26-Dec-23

Subject: Compiler Construction

COMSATS UNIVERSITY ISLAMABAD ATTOCK CAMPUS

Question No 01

Brief of the project

Input and Analysis:

- Accepts code input from a text box.
- Divides the code into tokens (individual elements).
- Implements methods to analyze tokens for different patterns:
- analyze1a, analyze1b, analyze2a, analyze2b, analyze3a, and analyze3b to handle various code structures.
- Checks syntax, validates identifiers, variables, and conditional statements (if).

Error Handling:

Incorporates error detection and handling mechanisms:

- Identifies errors like unidentified variables, incorrect syntax, or missing code blocks.
- Utilizes printErrors and MessageBox to inform users about errors.

Memory Management:

Manages memory-related data structures:

- memoryList, calcList, and finalMemoryList to store and update variable values and identifiers.
- updateValues function updates values based on the memory list.

Arithmetic Operations:

Processes arithmetic operations based on parsed statements:

- Handles addition, subtraction, multiplication, division, and modulo operations within the code.
- Updates memory values based on calculation results.

User Interaction:

Provides user feedback through messages and console logs:

- Utilizes Console.WriteLine to display information in the console.
- Shows MessageBox for compiled successfully or error messages.

User Interface Integration:

- Triggered by button click events (button3_Click and button4_Click) to initiate the compilation and analysis processes.

Question No 02

Control Flow of Project

Main Entry Point:

The mainAnalyze method seems to be the starting point of the compilation process. It splits the input code, performs lexical analysis to identify tokens, and then initiates different analysis functions based on identified tokens.

```
2 references
public bool mainAnalyze(int whichButton)
{
    string[] code = textBox1.Text.Split(' ');
    f = 1;
    error = "";
    double test;
    var regexItem = new Regex("^[a-zA-Z0-9 ]*$");

    if (code.Length >= 3)
    {

        for (int i = 0; i < code.Length; i++)
        {
            if (isIdentifier(code[i]))
            {
                analyze1a(i, code, 0);
                analyze1b(i, code, 0);
            }

            else if (isVariable(code[i]))
            {
                analyze2a(i, code, 0);
                analyze2b(i, code, 0);
            }

            else if (code[i] == "if")
            {
                analyze3a(i, code);
```

```
            else if (code[i] == "if")
            {
                analyze3a(i, code);
                analyze3b(i, code);

            }

            if (!code[i].All(char.IsLetter) || Double.TryParse(code[i], out test) || String.IsNullOrEmpty(code[i]) || !regexItem.IsMatch(code[i]))
            {
                if (i == 0)
                {
                    f = 0;
                    error = "Unexpected Error ";
                    break;
                }
                else if (i > 0)
                {
                    if ((code[i - 1] == ";" && code[i] != "}") || code[i - 1] == "}")
                    {
                        f = 0;
                        error = "Unexpected Error ";
                        break;
                    }
                }
            }


            if (f == 0) break;
```

```
        }
        else
        {
            f = 0;
            error = "Error Occurred -> Too little code to compile";
        }

        if (f == 1)
        {
            if(whichButton == 3)
            {
                //MessageBox.Show("Compiled Successfully", "Run", MessageBoxButtons.OK, MessageBoxIcon.Information);
                printErrors("Compiled Successfully, No Errors. Genius!", true);
            }
            return true;
        }
        else
        {
            //MessageBox.Show("Error Occurred " + error, "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
            printErrors("Error Occurred " + error, false);
            return false;
        }
    }
```

Semantic Analysis:

While not explicitly labeled, the code checks for semantic correctness by ensuring correct variable usage (updateValues method), performing arithmetic operations (button4_Click method), and handling memory operations (createMemoryLabels method).

```
1 reference
public void updateValues(int t)
{
    for (int j = 0; j < calcList[t].statement.Count; j++)              // Made it to change only the exact
                                                                       // statements in the first go, so n
                                                                       // it was changing all the statemen
                                                                       // already changed when it was chan
                                                                       // enter this "if" to change the va
    {
        for (int k = 0; k < memoryList.Count; k++)
        {
            if (calcList[t].statement[j] == memoryList[k].name && isVariable(calcList[t].statement[j]))
            {
                calcList[t].statement[j] = memoryList[k].value.ToString();
            }
        }
    }

}
```

Error Handling:
Throughout the code, various error messages are set (error = "...") when anomalies or violations of expected patterns are found. These messages are utilized for error reporting in the mainAnalyze method.

```
        }

        if (!code[i].All(char.IsLetter) || Double.TryParse(code[i], out test) || String.IsNullOrEmpty(code[i]) || !regexItem.IsMatch(code[i]))
        {
            if (i == 0)
            {
                f = 0;
                error = "Unexpected Error ";
                break;
            }
            else if (i > 0)
            {
                if ((code[i - 1] == ";" && code[i] != "}") || code[i - 1] == "}")
                {
                    f = 0;
                    error = "Unexpected Error ";
                    break;
                }
            }
        }
```

Memory Operations:
The code appears to handle memory-related operations (memoryList, calcList, finalMemoryList) to store and update values of variables.

```
private void button4_Click(object sender, EventArgs e)
{
    string[] code = textBox1.Text.Split(' ');
    memoryList.Clear();
    calcList.Clear();
    finalMemoryList.Clear();
    if (mainAnalyze(4) || true)
    {
        for (int i = 0; i < memoryList.Count; i++)
        {
            //MessageBox.Show("" + memoryList[i].name + " = " + memoryList[i].value, "Memory Output", MessageBoxButtons.OK, MessageBoxIco
            MemorySaver pnn = new MemorySaver();
            pnn.name = memoryList[i].name;
            pnn.value = memoryList[i].value;
            finalMemoryList.Add(pnn);
        }

        Console.WriteLine();
        Console.WriteLine();


        for (int i = 0; i < tempCalcList.Count; i++)
        {
            for(int j = 0; j < tempCalcList[i].statement.Count; j++)
            {
                //MessageBox.Show("" + tempCalcList[i].statement[j]);
            }
        }

        int value = 0;
        int f2 = 0;
        for (int i = 0; i < calcList.Count; i++)
        {
            updateValues(i);        /// <====================
            for (int j = 0; j < calcList[i].statement.Count; j++)
            {
                if (j == 1)
                {
                    try
                    {
                        if (calcList[i].statement[j] == "+")
                        {
                            value += Int32.Parse(calcList[i].statement[j - 1]) + Int32.Parse(calcList[i].statement[j + 1]);
                        }
                        else if (calcList[i].statement[j] == "-")
                        {
     {
                        try
                        {
                            if (calcList[i].statement[j] == "+")
                            {
                                value += Int32.Parse(calcList[i].statement[j - 1]) + Int32.Parse(calcList[i].statement[j + 1]);
                            }
                            else if (calcList[i].statement[j] == "-")
                            {
                                value += Int32.Parse(calcList[i].statement[j - 1]) - Int32.Parse(calcList[i].statement[j + 1]);
                            }
                            else if (calcList[i].statement[j] == "*")
                            {
                                value += Int32.Parse(calcList[i].statement[j - 1]) * Int32.Parse(calcList[i].statement[j + 1]);
                            }
                            else if (calcList[i].statement[j] == "/")
                            {
                                value += Int32.Parse(calcList[i].statement[j - 1]) / Int32.Parse(calcList[i].statement[j + 1]);
                            }
                            else if (calcList[i].statement[j] == "%")
                            {
                                value += Int32.Parse(calcList[i].statement[j - 1]) % Int32.Parse(calcList[i].statement[j + 1]);
                            }
                        }
                        catch(Exception ex)
                        {
                            printErrors(calcList[i].name + " Can't be Calculated because it includes one or more unidentified variable", false);///
                            f2 = 1;
                        }
```

```
{
    try
    {
        if (calcList[i].statement[j] == "+")
        {
            value += Int32.Parse(calcList[i].statement[j - 1]) + Int32.Parse(calcList[i].statement[j + 1]);
        }
        else if (calcList[i].statement[j] == "-")
        {
            value += Int32.Parse(calcList[i].statement[j - 1]) - Int32.Parse(calcList[i].statement[j + 1]);
        }
        else if (calcList[i].statement[j] == "*")
        {
            value += Int32.Parse(calcList[i].statement[j - 1]) * Int32.Parse(calcList[i].statement[j + 1]);
        }
        else if (calcList[i].statement[j] == "/")
        {
            value += Int32.Parse(calcList[i].statement[j - 1]) / Int32.Parse(calcList[i].statement[j + 1]);
        }
        else if (calcList[i].statement[j] == "%")
        {
            value += Int32.Parse(calcList[i].statement[j - 1]) % Int32.Parse(calcList[i].statement[j + 1]);
        }
    }
    catch(Exception ex)
    {
        printErrors(calcList[i].name + " Can't be Calculated because it includes one or more unidentified variable", false);///
        f2 = 1;
    }
}
else
{
    try
    {
        if (calcList[i].statement[j] == "+")
        {
            value += Int32.Parse(calcList[i].statement[j + 1]);
        }
        else if (calcList[i].statement[j] ==
        {
            value -= Int32.Parse(calcList[i].statement[j + 1]);
        }
        else if (calcList[i].statement[j] == "*")
        {
            value *= Int32.Parse(calcList[i].statement[j + 1]);
        }
        else if (calcList[i].statement[j] == "/")
        {
            value /= Int32.Parse(calcList[i].statement[j + 1]);
        }
        else if (calcList[i].statement[j] == "%")
        {
            value %= Int32.Parse(calcList[i].statement[j + 1]);
        }
    }
    catch (Exception ex)
    {
        printErrors(calcList[i].name + " Can't be Calculated because" + calcList[i].statement[j + 1] + "is unidentified variable",
```

memoryList
Aa Abl .* Current Document

memoryList
Aa Abl .* Current Document

List<string> MemoryCalculating.statement

```
            //MessageBox.Show("" + calcList[i].name + " = " + value, "Memory Output", MessageBoxButtons.OK,
            MemorySaver pnn = new MemorySaver();
            pnn.name = calcList[i].name;
            if (f2 == 1)
                pnn.value = "Undefined";
            else
                pnn.value = value.ToString();


            finalMemoryList.Add(pnn);
            value = 0;
        }
        // if f2 == 0     <============================
        createMemoryLabels();

        /////

        for (int i = 0; i < calcList.Count; i++)
        {
            Console.WriteLine(calcList[i].name);
            for (int j = 0; j < calcList[i].statement.Count; j++)
            {
                Console.WriteLine(calcList[i].statement[j]);
            }
        }
```

Arithmetic Operations:

The arithmetic operations are performed within the button4_Click method, calculating expressions and updating the memory values accordingly.

```
private void button4_Click(object sender, EventArgs e)
{
    string[] code = textBox1.Text.Split(' ');
    memoryList.Clear();
    calcList.Clear();
    finalMemoryList.Clear();
    if (mainAnalyze(4) || true)
    {
        for (int i = 0; i < memoryList.Count; i++)
        {
            //MessageBox.Show("" + memoryList[i].name + " = " + memoryList[i].value, "Memory Output", Mes
            MemorySaver pnn = new MemorySaver();
            pnn.name = memoryList[i].name;
            pnn.value = memoryList[i].value;
            finalMemoryList.Add(pnn);
        }

        Console.WriteLine();
        Console.WriteLine();



        for (int i = 0; i < tempCalcList.Count; i++)
        {
            for(int j = 0; j < tempCalcList[i].statement.Count; j++)
            {
                //MessageBox.Show("" + tempCalcList[i].statement[j]);
            }
        }
```

Output/Display:
There are some commented MessageBox.Show lines that might have been used for displaying specific values or debugging information.

```
private void button4_Click(object sender, EventArgs e)
{
    string[] code = textBox1.Text.Split(' ');
    memoryList.Clear();
    calcList.Clear();
    finalMemoryList.Clear();
    if (mainAnalyze(4) || true)
    {
        for (int i = 0; i < memoryList.Count; i++)
        {
            //MessageBox.Show("" + memoryList[i].name + " = " + memoryList[i].value, "Memory Output", MessageBoxButtons.OK, MessageBoxIcon.Informa
            MemorySaver pnn = new MemorySaver();
            pnn.name = memoryList[i].name;
            pnn.value = memoryList[i].value;
            finalMemoryList.Add(pnn);
        }

        Console.WriteLine();
        Console.WriteLine();



        for (int i = 0; i < tempCalcList.Count; i++)
        {
            for(int j = 0; j < tempCalcList[i].statement.Count; j++)
```

Question No 03

Data Flow of Project

Input Data:

The input data comes from a text box (textBox1.Text), likely containing code written in a custom programming language.

```
2 references
public bool mainAnalyze(int whichButton)
{
    string[] code = textBox1.Text.Split(' ');
    f = 1;
    error = "";
    double test;
    var regexItem = new Regex("^[a-zA-Z0-9 ]*$");

    if (code.Length >= 3)
    {

        for (int i = 0; i < code.Length; i++)
        {
            if (isIdentifier(code[i]))
            {
                analyze1a(i, code, 0);
                analyze1b(i, code, 0);
            }

            else if (isVariable(code[i]))
            {
                analyze2a(i, code, 0);
```

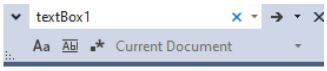
```
if (isIdentifier(code[i]))
{
    analyze1a(i, code, 0);
    analyze1b(i, code, 0);
}

else if (isVariable(code[i]))
{
    analyze2a(i, code, 0);
    analyze2b(i, code, 0);
}

else if (code[i] == "if")
{
    analyze3a(i, code);
    analyze3b(i, code);

}

if (!code[i].All(char.IsLetter) || Double.TryParse(code[i], out test) || String.IsNullOrEmpty(code[i]) || !regexItem.IsMatch(code[i]))
{
    if (i == 0)
    {
        f = 0;
        error = "Unexpected Error ";
        break;
    }
    else if (i > 0)
    {
```



```
            {
                if (i == 0)
                {
                    f = 0;
                    error = "Unexpected Error ";
                    break;
                }
                else if (i > 0)
                {
                    if ((code[i - 1] == ";" && code[i] != "}") || code[i - 1] == "}")
                    {
                        f = 0;
                        error = "Unexpected Error ";
                        break;
                    }
                }
            }

            if (f == 0) break;
        }


    }
    else
    {
        f = 0;
        error = "Error Occurred -> Too little code to compile";
    }
```

```
    }
    else
    {
        f = 0;
        error = "Error Occurred -> Too little code to compile";
    }

    if (f == 1)
    {
        if(whichButton == 3)
        {
            //MessageBox.Show("Compiled Successfully", "Run", MessageBoxButtons.OK, MessageBoxIcon.Information);
            printErrors("Compiled Successfully, No Errors. Genius!", true);
        }
        return true;
    }
    else
    {
        //MessageBox.Show("Error Occurred " + error, "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
        printErrors("Error Occurred " + error, false);
        return false;
    }
}
```

1 reference

Tokenization:

The input code is split into tokens using Split(' '), creating an array of strings (code[]). These tokens are used for further analysis.

Syntax and Semantic Analysis:

The code contains multiple methods (analyze1a, analyze1b, analyze2a, analyze2b, analyze3a, analyze3b, etc.) to check for syntactic and semantic correctness. These methods ensure that specific patterns or constructs in the code (such as variable assignments, if conditions) are correctly formed.

Memory and Calculation Lists:

There are data structures (memoryList, calcList, finalMemoryList) used to maintain information about variables, expressions, and their values. These structures store and manage data related to identifiers, values, and calculations.

Arithmetic Operations and Value Updates:

The updateValues method iterates through statements in the calculation list and updates their values based on the memory list. Additionally, the button4_Click method performs arithmetic operations (addition, subtraction, multiplication, division, modulo) based on the provided expressions and updates the memory list with the calculated values.

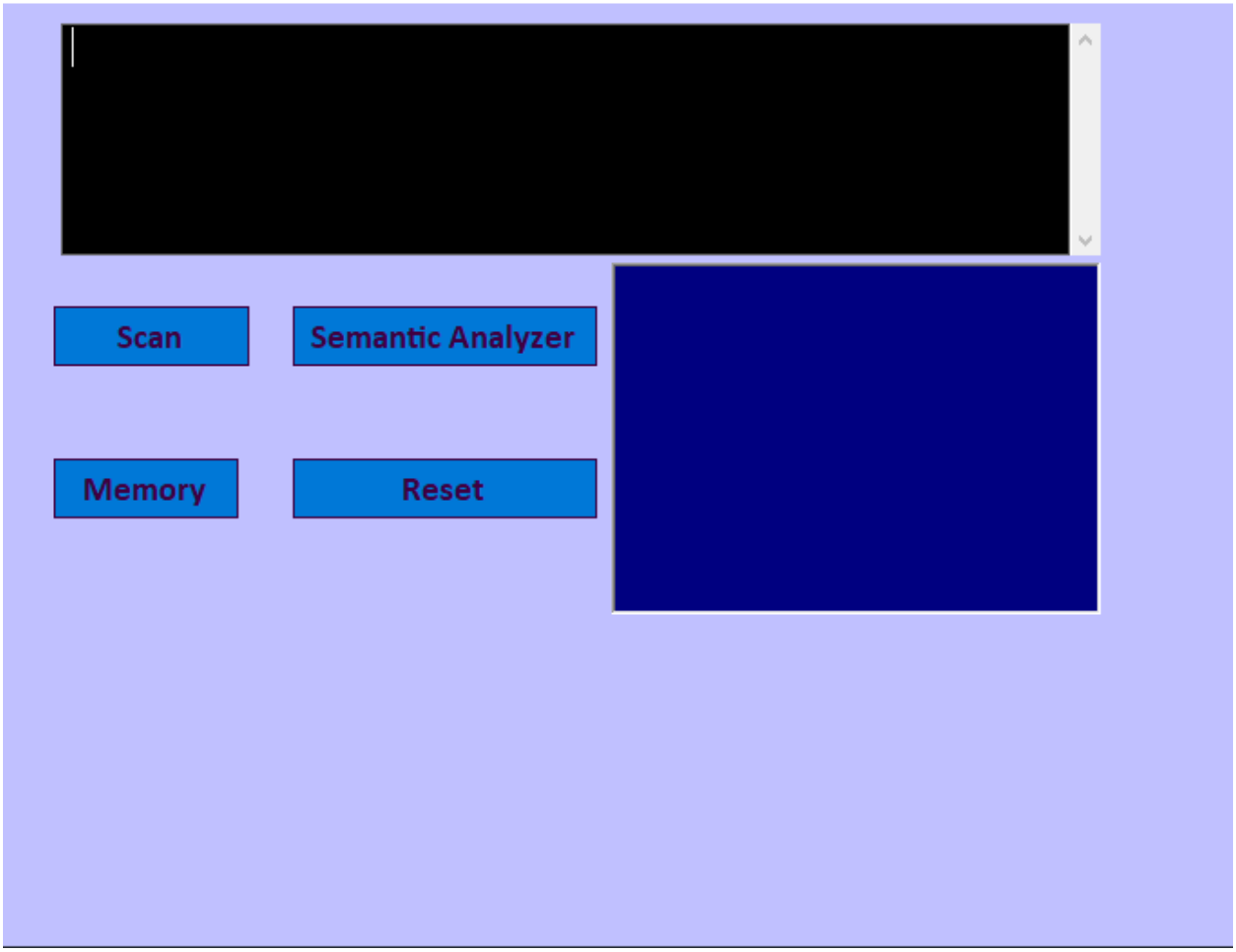
Error Handling:

Throughout the code, various error checks are performed to ensure that unexpected or incorrect inputs are captured. Error messages are set (error = "...") when anomalies are encountered, aiding in error reporting for the user.

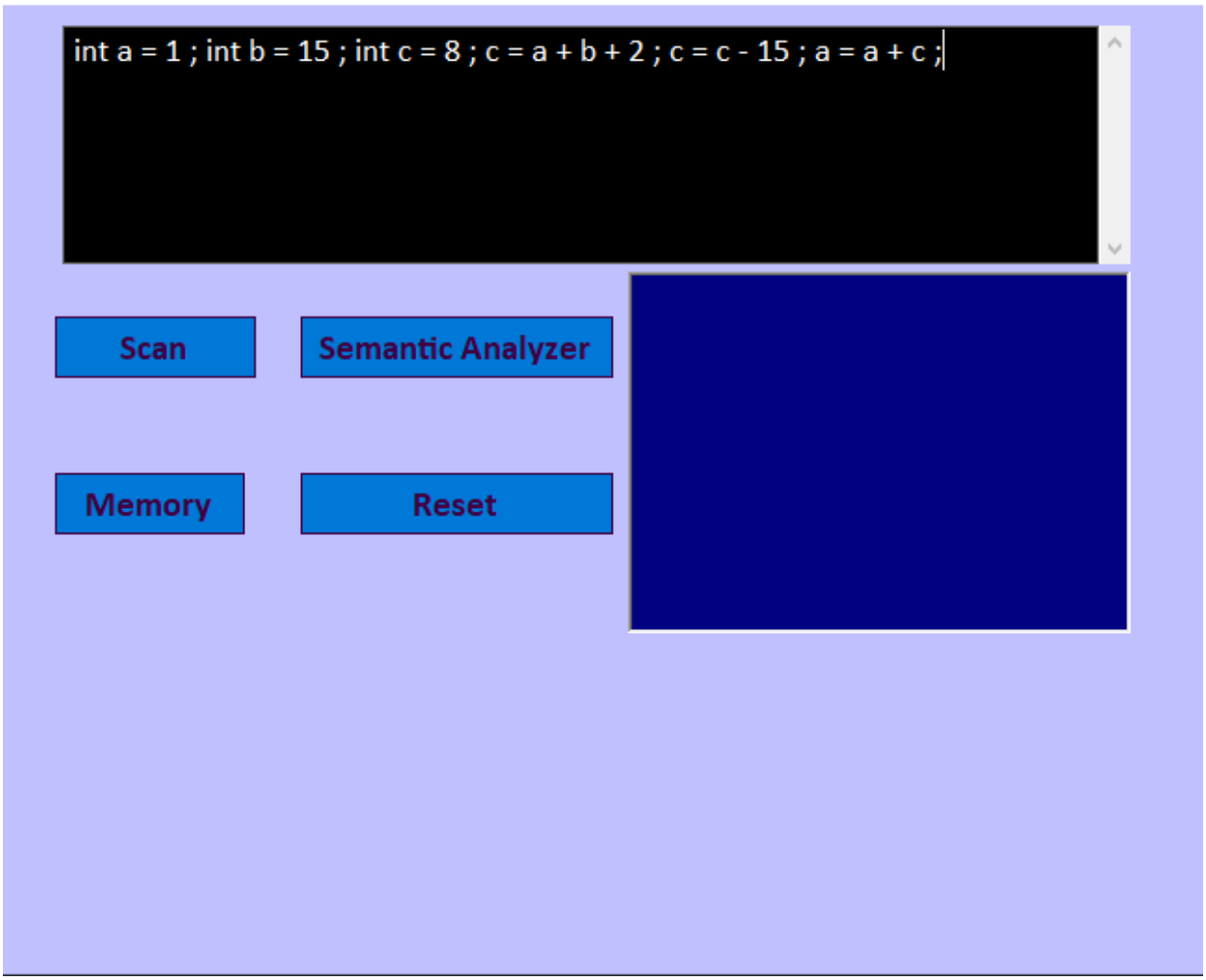
Output/Display:

The code may involve displaying messages, warnings, or final results through message boxes (MessageBox.Show). However, the actual output might vary based on the application's design and might not always be displayed in message boxes.

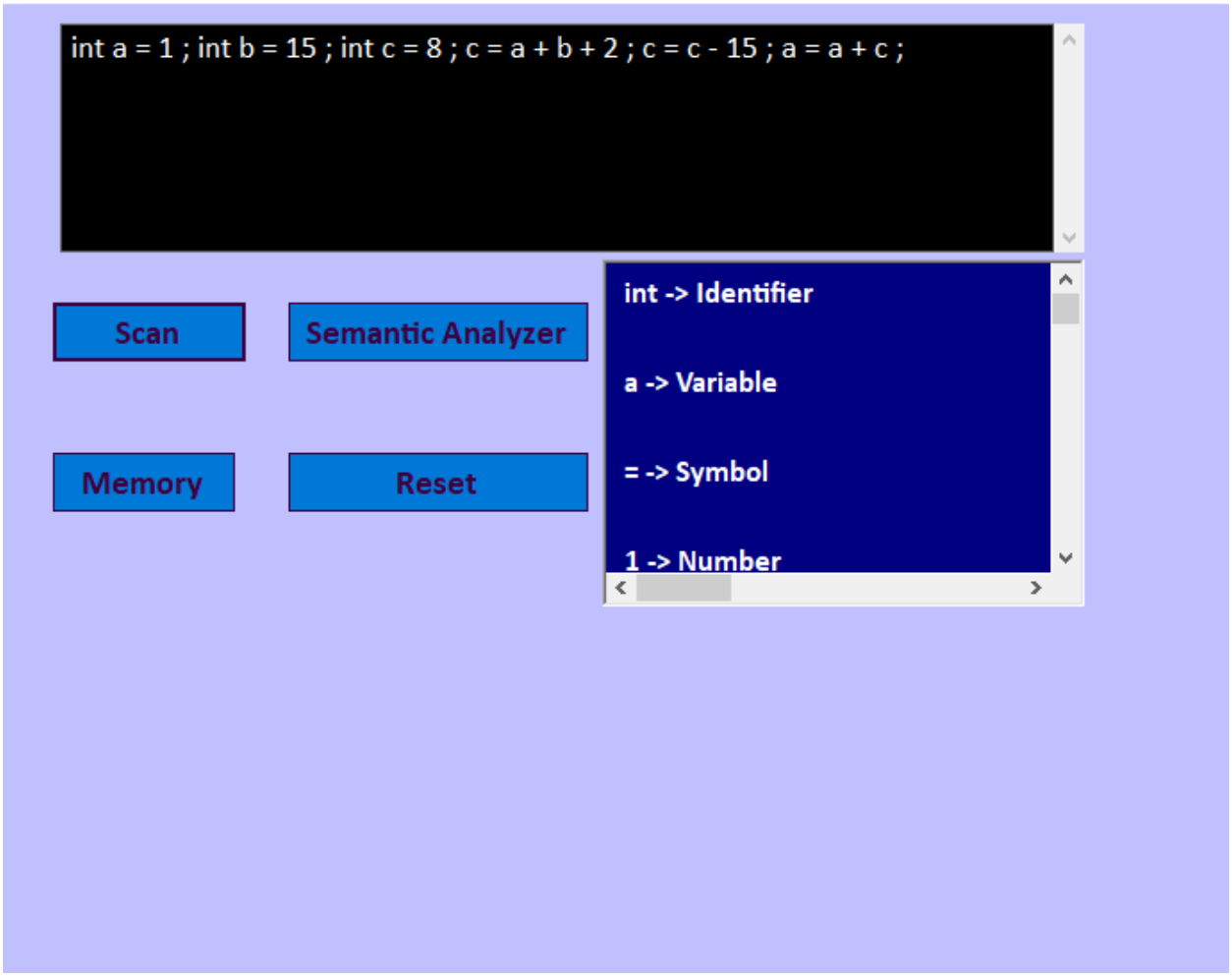
Lab Terminal

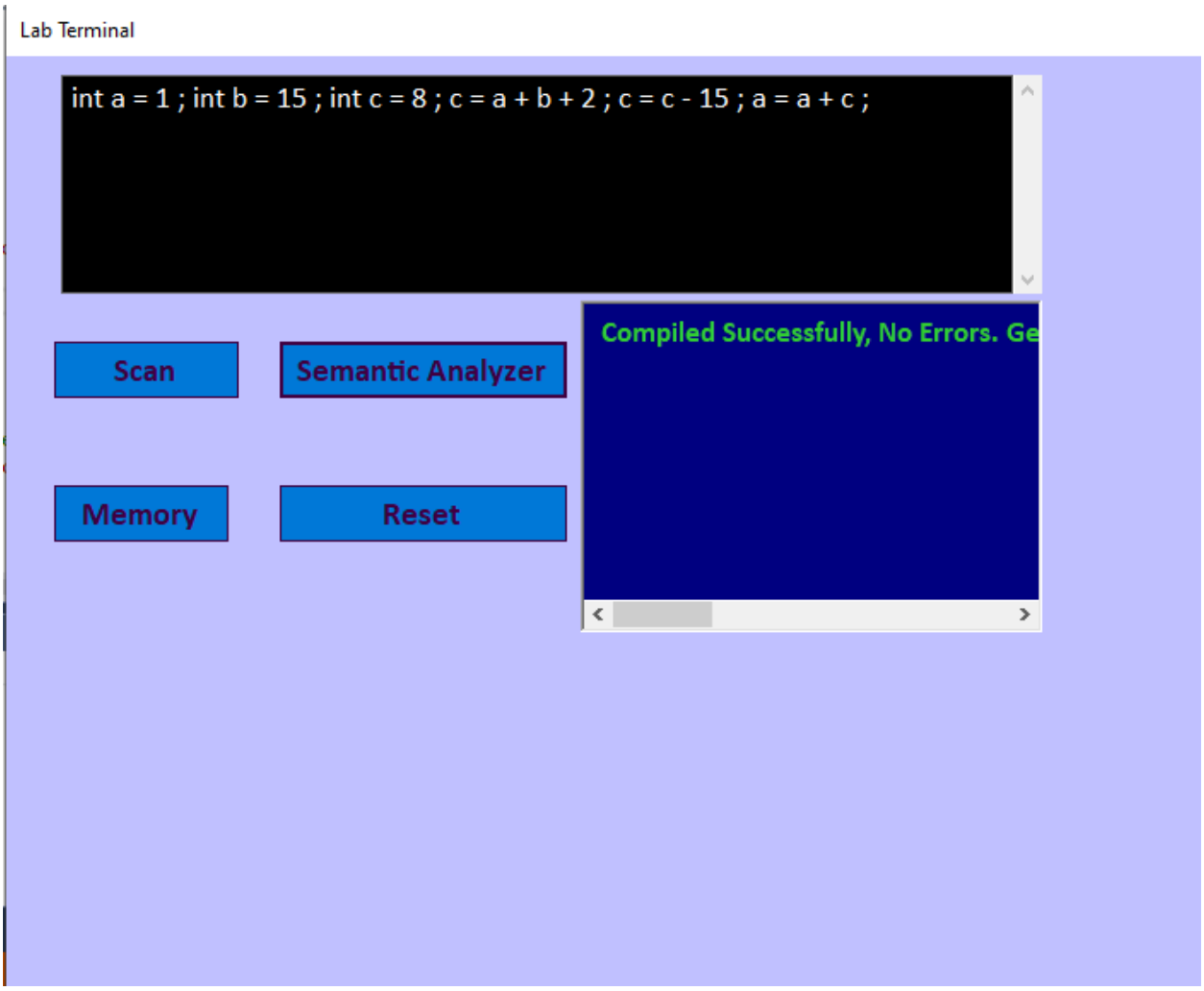


Lab Terminal

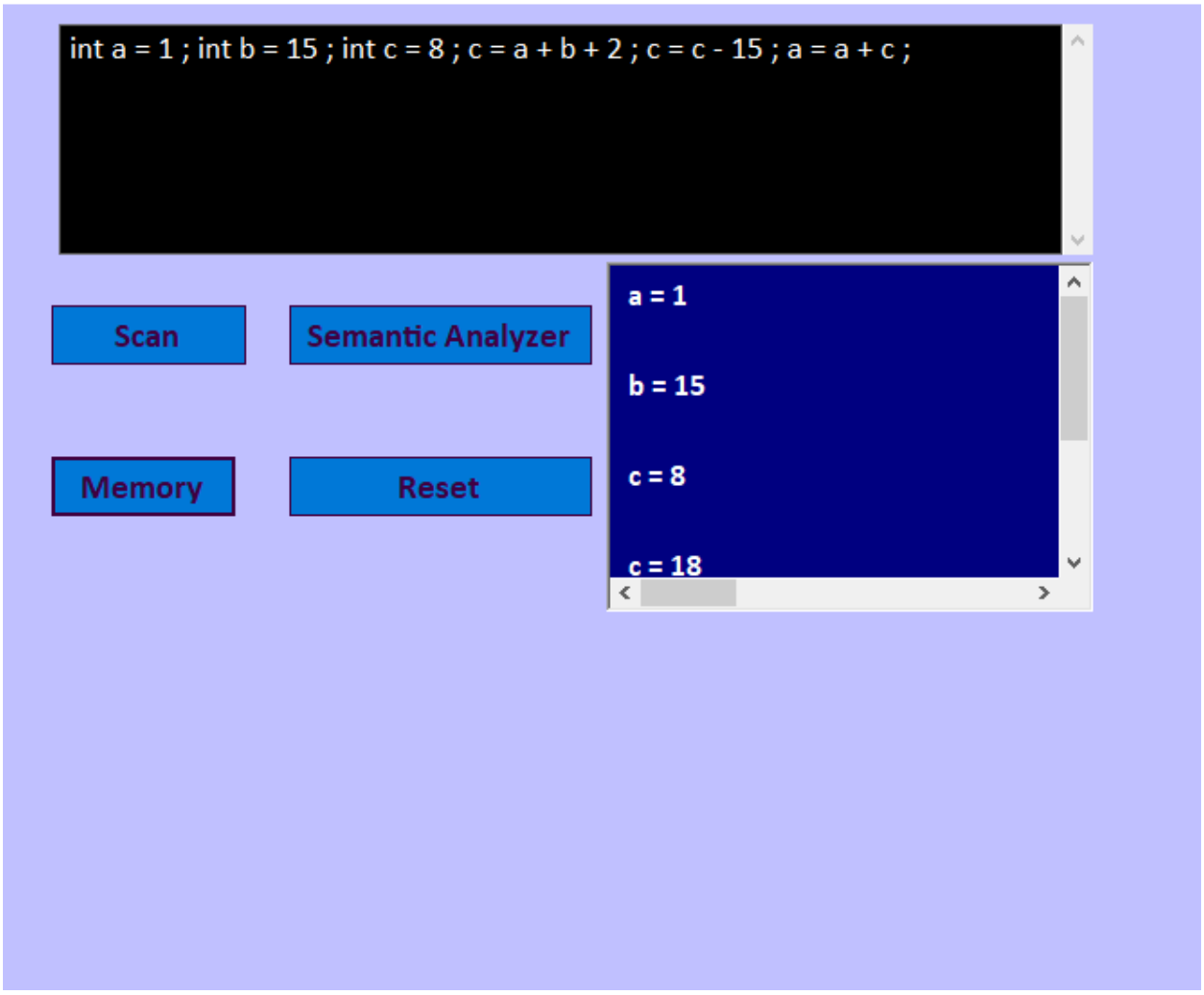


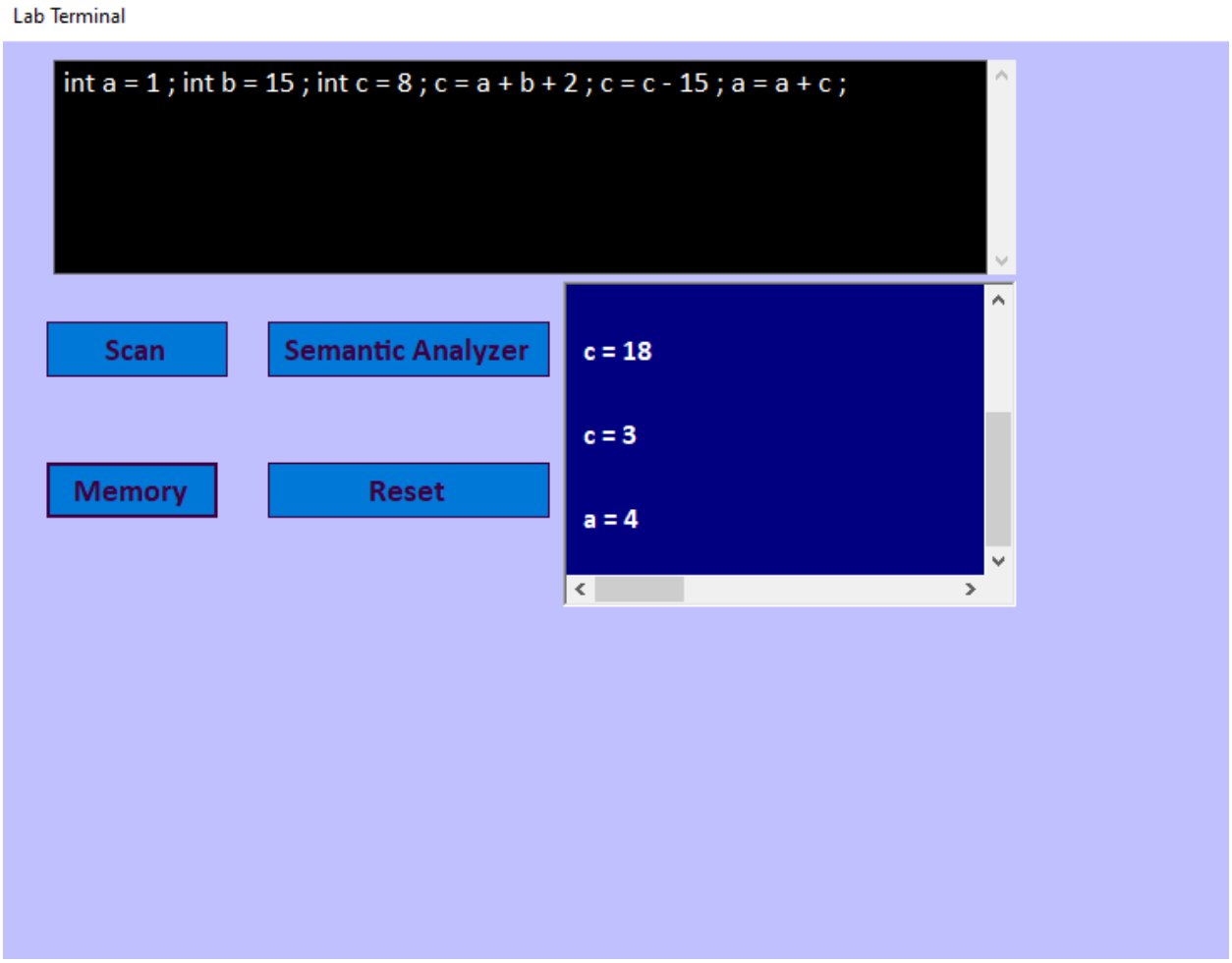
Lab Terminal




Lab Terminal
int a = 1 ; int b = 15 ; int c = 8 ; c = a + b + 2 ; c = c - 15 ; a = a + c ;
Scan
Semantic Analyzer
Compiled Successfully, No Errors. Ge
Memory
Reset

Lab Terminal





Question No 04

Function Structure and Invocation:

- **Function Declaration:**

Functions are declared with a specified signature, including the return type (bool, void), function name (analyze1a, mainAnalyze, etc.), and parameters (e.g., int i, string[] code).

- **Function Parameters:**

Parameters are variables that the function expects to receive when it's called. They act as placeholders for data to be passed into the function.

- **Function Invocation:**

Functions are called/invoked from other parts of the code using their names and, if required, passing arguments (values or variables) that match the parameters expected by the function.

Execution Flow:

- **Function Body:**

The body of a function contains the code that will be executed when the function is called.

- **Variable Scope:**

Variables declared within a function (local variables) have a scope limited to that function. They are not accessible outside of it.

- **Flow Control:**

Functions may include flow control structures like loops (for, while) and conditional statements (if, else) to control the execution flow based on specific conditions.

- **Error Handling:**

Functions can handle errors using try-catch blocks to catch exceptions that might occur during their execution. Errors caught within these blocks can be handled or reported.

- **Return Statement:**

Functions that have a return type (bool, void, etc.) use the return statement to provide a result or exit the function. When a return statement is encountered, the function stops executing and control returns to the calling point.

Calling Functions:

- **Function Arguments:**

When calling a function, arguments are passed within parentheses () if the function expects parameters. These arguments must match the data type and order of the function parameters.

- **Execution of the Called Function:**

When a function is called, the program's execution jumps to the function's code, and the function starts executing its defined logic.

- **Passing Control Back:**

After completing execution or encountering a return statement, the function passes control back to the point where it was called. This can include returning a value (in the case of functions with return types) or simply resuming the main program flow.

Question No 05

What challenges your faces during the project?

Complexity of Language Parsing: Parsing the syntax and grammar of a programming language can be intricate, especially if dealing with complex constructs, nested expressions, and various language features.

Error Handling: Managing and handling errors, such as syntax errors, semantic errors, or runtime errors, can be challenging. Ensuring informative and accurate error messages for debugging purposes is crucial.

Optimization: Implementing efficient algorithms and data structures to optimize the compilation process, memory usage, and runtime performance can be a challenge.

Semantic Analysis: Ensuring that the compiler understands the meaning behind the code (semantic analysis) and correctly interprets variables, functions, and their interactions.

Memory Management: Efficiently managing memory allocation and deallocation, especially when dealing with dynamic memory usage.
