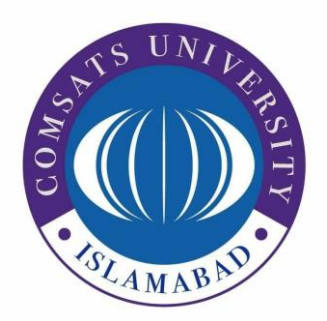


COMPILER CONSTRUCTION

LAB TERMINAL

NAME:

Anusha Amin (FA20-BCS-032)

CLASS:

BCS-7B

Submitted To:

Mr. Bilal Haider

Date:

27-December -2023

QUESTION NO.1

Brief Of Project:

We have developed a mini-compiler that works for a Java source code and has following functionalities:

- **Scanner (Lexical Analysis):**
  The scanner is the first phase of the compiler that reads the source code written in Java and breaks it down into tokens. Tokens are the smallest units of the code, such as keywords, identifiers, literals, operators, and punctuation marks. The scanner ensures that the code is well-formed by identifying and categorizing these tokens according to the rules of the Java language. It eliminates comments and whitespace, focusing on extracting the essential elements of the code.
- **Semantic Analysis:**

Semantic analysis is the second phase of the compiler and involves checking the meaning and logic of the code. This phase ensures that the code adheres to the semantic rules of the Java language. It involves checking for proper variable declarations, type compatibility, and adherence to the language's syntactic and semantic rules. Semantic analysis identifies and reports errors related to the misuse of variables, incompatible data types, undeclared identifiers, etc.

- **Memory Analyzer:**

The memory analyzer is a specialized component that focuses on memory-related aspects of the code. It checks for memory allocation and deallocation issues, such as memory leaks or attempts to access unallocated memory. This phase may involve tracking variables, ensuring they are properly initialized and released, and managing dynamic memory allocation if applicable. The memory analyzer contributes to the overall reliability and efficiency of the generated code.

- **Simplified flow of your mini compiler:**

Input: Java source code.

Scanner: Tokenizes the code and removes comments and whitespace.

Semantic Analysis: Checks the code for adherence to language rules, proper variable usage, and type compatibility.

Memory Analyzer: Focuses on memory-related aspects, checking for allocation and deallocation issues.

Output: Reports any errors or warnings found during the scanning, semantic analysis, and memory analysis phases.

QUESTION NO.2

Scanner Functionality:

The scanner, also known as the lexical analyzer, is responsible for breaking down the source code into tokens, which are the smallest units of meaning in the programming language. It performs tasks such as recognizing keywords, identifiers, operators, and literals. The scanner plays a crucial role in the initial phase of the compilation process by converting the source code into a stream of tokens that can be further processed by the parser.

Semantic Analysis Functionality:

Semantic analysis is a critical stage in the compilation process that goes beyond syntax and checks the meaning of the source code. It involves verifying that the program adheres to the language's semantics and rules. The semantic analyzer performs tasks such as type checking, ensuring that variables are used correctly, and building a symbol table to keep track of identifiers and their attributes. It plays a vital role in catching errors related to the logical structure of the program and ensures that the generated code will behave as intended.

Scanner Function:

```
private void button1_Click(object sender, EventArgs e)
{
    string[] code = textBox1.Text.Split(' ');

    for(int i = 0; i < labelsList.Count; i++)
    {
        flowLayoutPanel1.Controls.Remove(labelsList[i]);
    }

    for (int j = 0; j < memoryLabels.Count; j++)
    {
        flowLayoutPanel1.Controls.Remove(memoryLabels[j]);
    }

    flowLayoutPanel1.Controls.Remove(errLabel);

    for (int i = 0; i < code.Length; i++)
    {
        Label label = new Label();
        labelsList.Add(label);
    }

    if (!String.IsNullOrEmpty(textBox1.Text) && code[code.Length - 1] != "")
    {
        this.Size = new Size(1304, 1087);       //559 + (code.Length * 16)
        var regexItem = new Regex("^[a-zA-Z0-9 ]*$");
        for (int i = 0; i < code.Length; i++)
        {
```

```
        double test;
        if (isIdentifier(code[i]))
        {
            labelsList[i].Font = new System.Drawing.Font("Calibri", 12.75F, System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            labelsList[i].ForeColor = System.Drawing.Color.White;
            labelsList[i].Name = "newLabel" + i;
            labelsList[i].Size = new System.Drawing.Size(1000, 36);
            labelsList[i].Text = code[i] + " -> Identifier";
            labelsList[i].Margin = new System.Windows.Forms.Padding(6, 6, 6, 8);
            flowLayoutPanel1.Controls.Add(labelsList[i]);
        }

        else if (isSymbol(code[i]))
        {
            labelsList[i].Font = new System.Drawing.Font("Calibri", 12.75F, System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            labelsList[i].ForeColor = System.Drawing.Color.White;
            labelsList[i].Name = "newLabel" + i;
            labelsList[i].Size = new System.Drawing.Size(1000, 36);
            labelsList[i].Text = code[i] + " -> Symbol";
            labelsList[i].Margin = new System.Windows.Forms.Padding(6, 6, 6, 8);
            flowLayoutPanel1.Controls.Add(labelsList[i]);
        }

        else if (isReversedWord(code[i]))
        {
            labelsList[i].Font = new System.Drawing.Font("Calibri", 12.75F, System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            labelsList[i].ForeColor = System.Drawing.Color.White;
            labelsList[i].Name = "newLabel" + i;
```

```
            labelsList[i].Size = new System.Drawing.Size(1000, 36);
            labelsList[i].Text = code[i] + " -> Reversed Word";
            labelsList[i].Margin = new System.Windows.Forms.Padding(6, 6, 6, 8);
            flowLayoutPanel1.Controls.Add(labelsList[i]);
        }
        else if (!isIdentifier(code[i]) && !isSymbol(code[i]) && !isReversedWord(code[i]) && !code[i].All(char.IsDigit) && !Double.TryParse(code[i], out test) && (regexItem.IsMatch(code[i])))
        {
            labelsList[i].Font = new System.Drawing.Font("Calibri", 12.75F, System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            labelsList[i].ForeColor = System.Drawing.Color.White;
            labelsList[i].Name = "newLabel" + i;
            labelsList[i].Size = new System.Drawing.Size(1000, 36);
            labelsList[i].Text = code[i] + " -> Variable";
            labelsList[i].Margin = new System.Windows.Forms.Padding(6, 6, 6, 8);
            flowLayoutPanel1.Controls.Add(labelsList[i]);
        }
        else if (!isIdentifier(code[i]) && !isSymbol(code[i]) && !isReversedWord(code[i]) && (code[i].All(char.IsDigit) || Double.TryParse(code[i], out test)) && !String.IsNullOrEmpty(code[i]))
        {
            labelsList[i].Font = new System.Drawing.Font("Calibri", 12.75F, System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
            labelsList[i].ForeColor = System.Drawing.Color.White;
            labelsList[i].Name = "newLabel" + i;
            labelsList[i].Size = new System.Drawing.Size(1000, 36);
            labelsList[i].Text = code[i] + " -> Number";
            labelsList[i].Margin = new System.Windows.Forms.Padding(6, 6, 6, 8);
            flowLayoutPanel1.Controls.Add(labelsList[i]);


        }
```

```
        else
        {
            if (code[i][0] != '' && code[i][code[i].Length - 1] == '')
            {
                labelsList[i].Font = new System.Drawing.Font("Calibri", 12.75F, System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
                labelsList[i].ForeColor = System.Drawing.Color.White;
                labelsList[i].Name = "newLabel" + i;
                labelsList[i].Size = new System.Drawing.Size(1000, 36);
                labelsList[i].Text = code[i] + " -> Pointer";
                labelsList[i].Margin = new System.Windows.Forms.Padding(6, 6, 6, 8);
                flowLayoutPanel1.Controls.Add(labelsList[i]);
            }
            else
            {
                labelsList[i].Font = new System.Drawing.Font("Calibri", 12.75F, System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point, ((byte)(0)));
                labelsList[i].ForeColor = System.Drawing.Color.White;
                labelsList[i].Name = "newLabel" + i;
                labelsList[i].Size = new System.Drawing.Size(1000, 36);
                labelsList[i].Text = code[i] + " -> Error";
                labelsList[i].Margin = new System.Windows.Forms.Padding(6, 6, 6, 8);
                flowLayoutPanel1.Controls.Add(labelsList[i]);
            }
        }

    }
  }

  }
```

Semantics Analysis Function:

```
private void button3_Click(object sender, EventArgs e)
 {
    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);
            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. Perfect", true);
        }
        return true;
    }
    else
    {
        //MessageBox.Show("Error Occurred " + error, "Error", MessageBoxButtons.OK,
MessageBoxIcon.Error);
        printErrors("Error Occurred " + error, false);
        return false;
    }
 }
 }
```

Memory Analysis Function:

```
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.Information);
           MemorySaver identifier = new MemorySaver();
           identifier.name = memoryList[i].name;
           identifier.value = memoryList[i].value;
           finalMemoryList.Add(identifier);
        }

        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] == "-")
                {
                    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", false);///
                f2 = 1;
            }
        }
    }
    for (int t = 0; t < memoryList.Count; t++)
    {
        if (memoryList[t].name == calcList[i].name)
        {
            memoryList[t].value = value.ToString(); // <====================
            break;
        }
    }
```

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



            finalMemoryList.Add(identifier);
            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]);
            }
        }
    }
}
```

QUESTION NO.3

Process Of Execution:

1. The process starts with the acquisition of source code written by a programmer in a specific programming language.

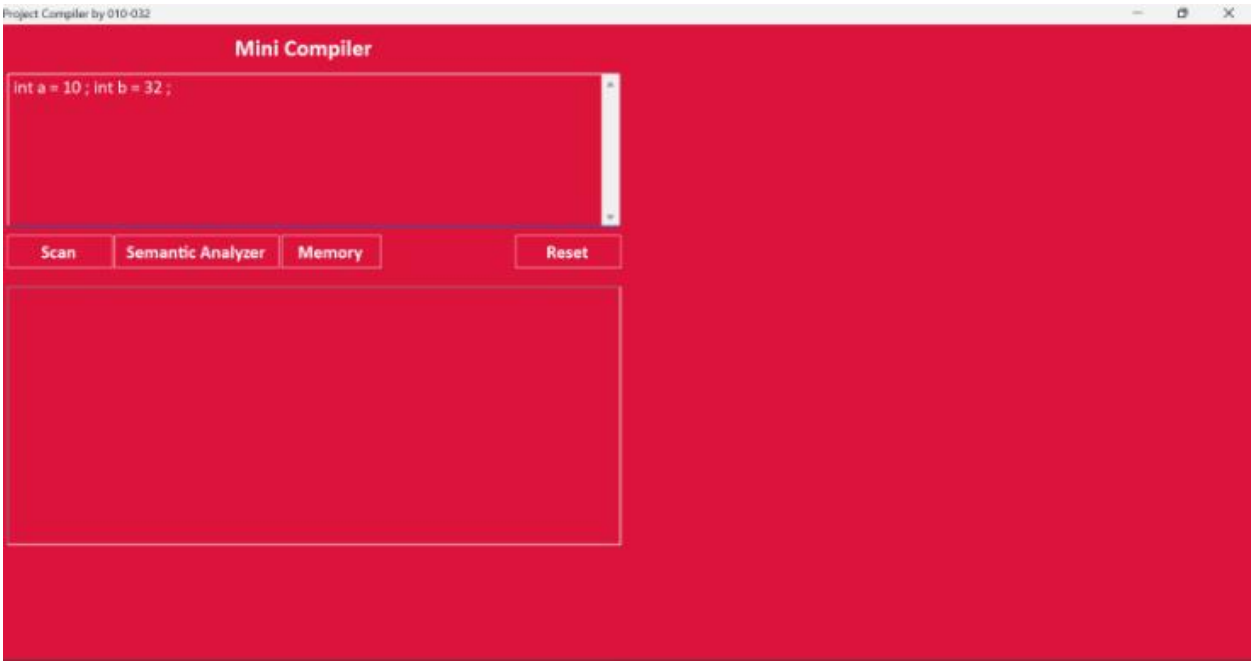
2. The source code undergoes tokenization, breaking it down into meaningful units called tokens. This involves identifying keywords, operators, identifiers, and literals.

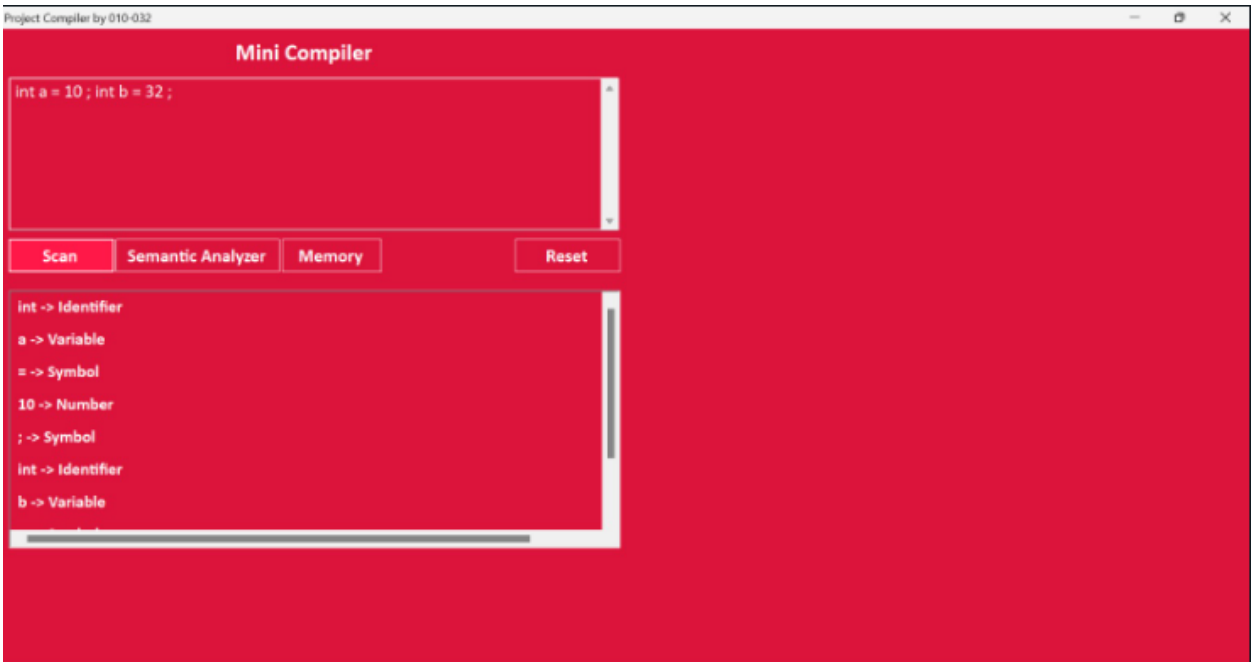
3. After tokenization, semantic analysis is performed to ensure that the code adheres to the language's semantics. This includes type checking, symbol table creation, and other checks to verify the correctness of the code's meaning.

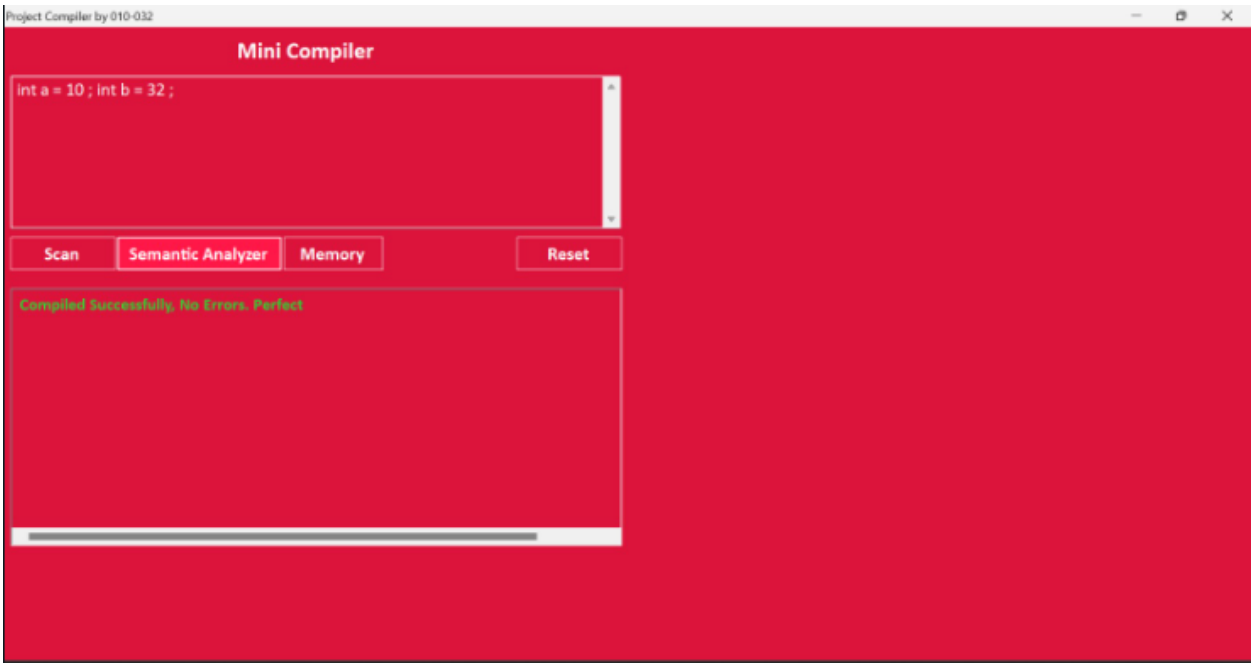
4. The program allocates memory for variables, data structures, and other elements needed for execution. Memory management involves dynamic allocation and deallocation as the program runs.

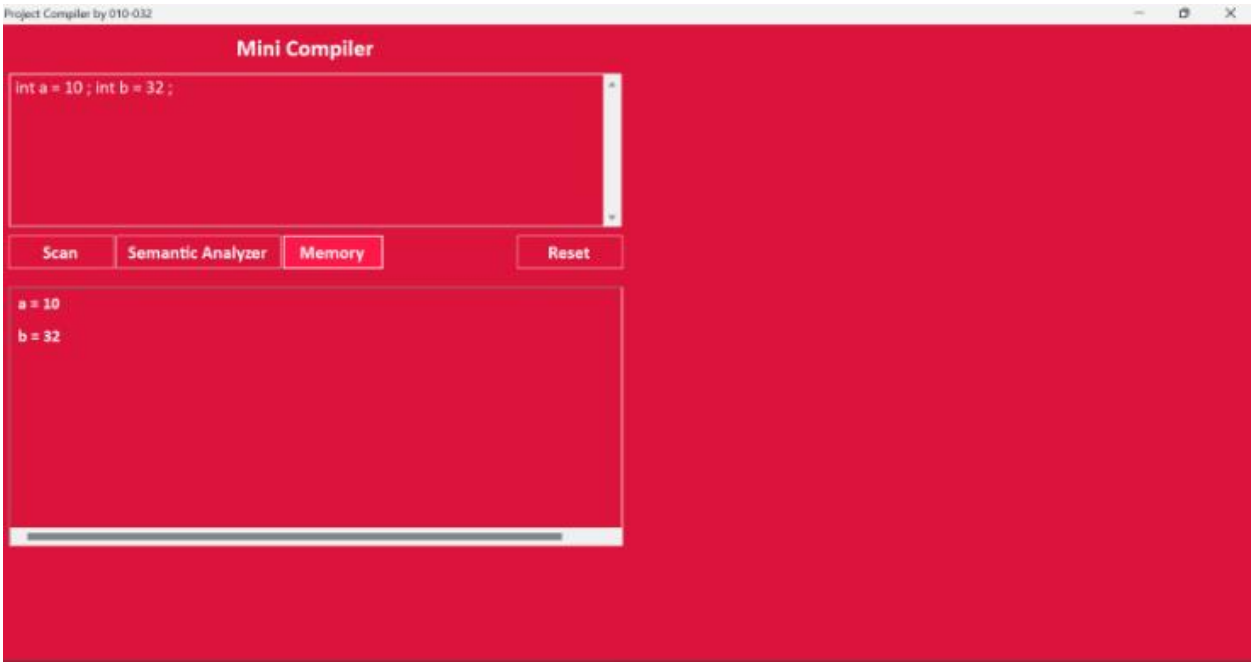
5. Report any errors or warnings found during the scanning, semantic analysis, and memory analysis phases.

Screenshots:




Project Compiler by 010-032
Mini Compiler
int a = 10 ; int b = 32 ;
Scan
Semantic Analyzer
Memory
Reset
int -> Identifier
a -> Variable
= -> Symbol
10 -> Number
; -> Symbol
int -> Identifier
b -> Variable


Project Compiler by 010-032
Mini Compiler
int a = 10 ; int b = 32 ;
Scan
Semantic Analyzer
Memory
Reset
Compiled Successfully, No Errors. Perfect



QUESTION NO.4

Step-by-Step Process of the Mini Compiler:

1. Input:

Input: Java source code.

2. Scanner (Lexical Analysis):

Tokenize the code and remove comments and whitespace.

Steps:

- Read the input Java source code.
- Break down the code into tokens (keywords, identifiers, literals, operators, etc.).
- Eliminate comments and whitespace to obtain a stream of relevant tokens.
- Identify and categorize each token according to the rules of the Java language.

3. Semantic Analysis:

Check the code for adherence to language rules, proper variable usage, and type compatibility.

Steps:

- Receive the stream of tokens from the Scanner.
- Analyze the structure of the code to ensure it follows the syntactic rules of Java.
- Perform semantic checks, including:
- Verify proper variable declarations and usage.
- Check for type compatibility.
- Ensure that identifiers are declared before use.
- Report errors for any violations of semantic rules.

4. Memory Analyzer:

Focus on memory-related aspects, checking for allocation and deallocation issues.

Steps:

- Receive the analyzed code from the Semantic Analysis phase.
- Perform memory-related checks, including:
- Track variable usage and ensure proper initialization.
- Check for memory leaks by verifying proper deallocation.
- Manage dynamic memory allocation (if applicable).
- Report errors or warnings related to memory issues.

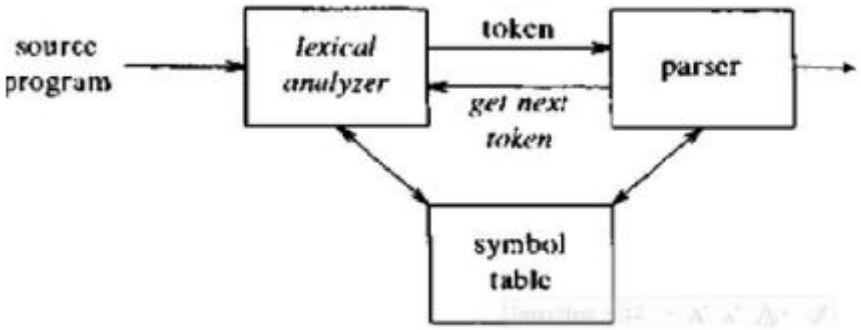
5. Output:

Report any errors or warnings found during the scanning, semantic analysis, and memory analysis phases.

Steps:

- Generate a comprehensive report based on the findings of the Scanner, Semantic Analysis, and Memory Analyzer.
- Output any errors, warnings, or messages indicating the status of the input Java code.
- Provide a summary of the code's correctness, adherence to language rules, and memory-related issues.

Class Diagram:



QUESTION NO.5

Following were the difficulties faced in the project:

- **Regular Expression Complexity**

  Crafting accurate and efficient regular expressions for token recognition presents a notable challenge in our project. The complexity of expressions escalates with the diversity of the input language, demanding a delicate balance between specificity and generality.

- **Ambiguities in Token Definitions:**

  Defining tokens with potential ambiguities poses challenges. For instance, distinguishing between unary and binary operators or handling numbers with decimal points necessitates careful consideration to avoid misinterpretations.

- **Whitespace and Comments Handling:**

  Properly managing whitespace and comments is often overlooked but essential. Incorrect handling can impact tokenization and lead to unexpected behavior in the interpreter.

- **Error Handling and Reporting:**

  Designing a robust error-handling mechanism is crucial for providing meaningful feedback to users when the lexer encounters invalid input. Identifying the location and nature of errors and reporting them clearly can be challenging.

- **Efficiency and Performance:**

  Balancing efficiency and performance while processing large input strings is a common concern. Optimizing regular expressions or adopting more efficient algorithms may be necessary to ensure a responsive lexical analysis.

- **Reserved Words and Keywords:**

  Dealing with reserved words and keywords in the language adds complexity. Ensuring that specific character sequences are recognized as keywords and not misinterpreted as identifiers or other entities requires careful handling.

- **Unicode and Character Encoding:**

  Support for different character encodings and Unicode characters introduces complexity. The lexer should handle a variety of character sets and ensure the correct interpretation of characters across different language specifications.

- **Flexibility for Language Extensions:**

  Creating a lexer that can be easily extended to support different languages or language variations is a challenge. The design should accommodate future language additions without requiring significant modifications.