COMPILER CONSTRUCTION

LAB TERMINAL

NAME:

USAMA MANSOOR

REG. NO:

FA20-BCS-026

CLASS:

BCS-7B

SUBMITTED TO:

SIR BILAL HAIDER

DATE:

27-12-2023

SEMANTIC ANALYZER:

- **QUESTIO NO 01:**

Brief Introduction of Project

- **ANSWER:**

The project involves a basic compiler or interpreter that processes user-entered source code. The main components include a Scanner, Semantic Analyzer.

- **Semantic Analyzer:**

This component applies semantic rules to the tokens generated by the scanner. It checks for semantic errors, ensuring that the code makes logical sense. This includes checking for proper variable declaration and assignment, as well as verifying that operations are performed between compatible types.

Following are the rules that my Semantic Analyzer follows:

RULE 1: First rule checks the datatypes that used in source code (input).

RULE 2: Second rule checks operators (+, -, *, /) in source code (input).

RULE 3: Third rule checks logical operators (>=, <=, >, <) in source code (input).

The SemanticAnalyzer() function takes the list of tokens identified by the Scanner, and checks if the tokens sequence follow any of the three rules that are mentioned above using Top-Down Parsing, then it return a list of the errors found if any.

- **Scanner:**

The scanner is responsible for breaking down the source code into individual tokens. It classifies these tokens into categories such as identifiers, numbers, variables, operators, etc.

• QUESTION NO 02:

Functionalities of Project

• ANSWER:

The two main functionalities of the project are:

Lexical Analysis and Tokenization:

This involves breaking down the source code into individual tokens (words) and classifying them into different categories such as identifiers, numbers, variables, and operators.

Semantic Analysis and Error Checking:

This involves checking the code for errors that make it nonsensical, such as using undeclared variables or performing operations on incompatible types.

Code of classes:

SCANNER CLASS:

```
List<Token> Scanner( List<string> splitCode ) {
        List<Token> output = new List<Token>();
        List<string> identifiers = new List<string>( new string[] { "int", "float", "string", "double",
"bool", "char" } );
        List<string> symbols = new List<string>( new string[] { "+", "-", "/", "%", "*", "(", ")", "{", "}",
",", ";", "&&",

                        "||", "<", ">", "=", "!","++","==",">=","<=","!=" } );
        List<string> reservedWords = new List<string>( new string[] { "for", "while", "if", "do",
"return", "break", "continue", "end" } );
        bool match = false;


        for ( int i = 0; i < splitCode.Count; i++ ) {
                if ( identifiers.Contains( splitCode[i] ) && match == false ) {
                        output.Add( new Token( splitCode[i], "identifier" ) );
                        match = true;
                }
                if ( symbols.Contains( splitCode[i] ) && match == false ) {
                        output.Add( new Token( splitCode[i], "symbol" ) );
                        match = true;
                }
                if ( reservedWords.Contains( splitCode[i] ) && match == false ) {
                        output.Add( new Token( splitCode[i], "reserved word" ) );
                        match = true;
                }
                if ( float.TryParse( splitCode[i], out _ ) && match == false ) {
                        output.Add( new Token( splitCode[i], "number" ) );
                        match = true;
                }
                if ( isValidVar( splitCode[i] ) && match == false ) {

                        variable pnn = new variable();
                        pnn.name = splitCode[i];
         Lvar.Add(pnn);
                        output.Add( new Token( splitCode[i], "variable" ) );
                        match = true;
                }
                if ( splitCode[i].StartsWith( "\"" ) && splitCode[i].EndsWith( "\"" ) && match == false ) {
                        output.Add( new Token( splitCode[i], "string" ) );
                        match = true;
                }
                if ( match == false ) {
                        output.Add( new Token( splitCode[i], "unknown" ) );
                }
                match = false;
        }
        return output;

        bool isValidVar( string v ) {
                if ( v.Length >= 1 ) {
                        if ( char.IsLetter( v[0] ) || v[0] == '_' ) {
                                return true;
                        }
                        else {
                                return false;
                        }
                }
                else {
                        return false;
                }
        }
}
```

SEMANTIC ANALYZER CODE:

```
List<string> errors = new List<string>();
```

```
Token prevInput1 = new Token();
Token prevInput2 = new Token();
Token prevInput3 = new Token();

int selectedRule = 0;
for ( int i = 0; i < tokens.Count; i++ ) {
	if ( selectedRule == 0 ) {
		if ( Rule1( tokens[i] ).StartsWith( "Start" ) ) {
			selectedRule = 1;
			continue;
		}
		if ( Rule2( tokens[i] ).StartsWith( "Start" ) ) {
			selectedRule = 2;
			continue;
		}
		if ( Rule3( tokens[i] ).StartsWith( "Start" ) ) {
			selectedRule = 3;
			continue;
		}
	}

	if ( selectedRule == 1 ) {
		var state = Rule1( tokens[i] );
		if ( state.StartsWith( "Ok" ) || state.StartsWith( "Error" ) ) {
			errors.Add( state );
			selectedRule = 0;
		}
	}
	if ( selectedRule == 2 ) {
		var state = Rule2( tokens[i] );
		if ( state.StartsWith( "Ok" ) || state.StartsWith( "Error" ) ) {
			errors.Add( state );
			selectedRule = 0;
		}
	}
	if ( selectedRule == 3 ) {
		var state = Rule3( tokens[i] );
		if ( state.StartsWith( "Ok" ) || state.StartsWith( "Error" ) ) {
			errors.Add( state );
			selectedRule = 0;
		}
	}
}

if ( selectedRule == 1 ) {
	errors.Add( Rule1( new Token() ) );
}
if ( selectedRule == 2 ) {
	errors.Add( Rule2( new Token() ) );
}
if ( selectedRule == 3 ) {
        errors.Add( Rule3( new Token() ) );
}

// code is too large so I pasted some portion in this file
```

RULES:

FIRST RULE:

```
string Rule1( Token input ) {
        List<string> identifiers = new List<string>(new string[] { "int", "float", "string", "double",
"bool", "char" });
        if ( prevInput1.name == "" && input.type == "identifier" ) {
                prevInput1 = input;
                if(prevInput1.name=="int")
     {
                        f = 1;
     }
                if (prevInput1.name == "float")
                {
                        f = 2;
                }
                if (prevInput1.name == "string")
                {
                        f = 3;
                }
                if (prevInput1.name == "double")
                {
                        f = 4;
                }
                if (prevInput1.name == "bool")
                {
                        f = 5;
                }
                if (prevInput1.name == "char")
                {
                        f = 6;
                }
                return "Start Rule 1";
        }
        else if ( prevInput1.type == "identifier" ) {
                string state = Rule2( input );
                if ( state.StartsWith( "Ok" ) ) {
                        prevInput1 = new Token();
                }
                if ( state != "Error Rule 2" ) {
                        return state.Substring( 0, state.IndexOf( "Rule 2" ) - 1 ) + " Rule 1";
                }
        }
        if ( prevInput1.type == "identifier" ) {
                prevInput1 = new Token();
                return "Error Expected 'variable' Rule 1";
        }
        prevInput1 = new Token();
        return "Error Rule 1";
}
```

SECOND RULE:

```
string Rule2( Token input ) {
        List<string> operators = new List<string>( new string[] { "+", "-", "/", "%", "*" } );
        List<Int32> memoryint = new List<Int32>();


        if ( prevInput2.name == "" && input.type == "variable" ) {
                prevInput2 = input;

                return "Start Rule 2";
        }
        else if ( prevInput2.type == "variable" && input.name == ";" ) {
                prevInput2 = new Token();
                fstop = 1;
                return "Ok Rule 2";
        }
        else if ( prevInput2.type == "variable" && input.name == "=" ) {
                firstvar = prevInput2.name;
                prevvar = prevInput2.name;
                prevInput2 = input;

                return "Continue Rule 2";
        }
        else if ( prevInput2.name == "=" && input.type == "variable" ) {
                prevInput2 = input;
                prevtypevar = input.name;
                if (f == 1)
                {

                                flagoutput = 0;
                                for (int i = 0; i < Lvar.Count; i++)
                                {
                                        if (Lvar[i].name == prevInput2.name && flagoutput == 0)
                                        {
                                                for (int j = 0; j < Lvar.Count; j++)
                                                {
                                                        if (Lvar[j].name == prevvar)
                                                        {
                                                         if (fstop == 1)
                                                         {
                                                                Lvar[j].integer = Lvar[i].integer;
varsize.Add(Convert.ToString(Lvar[j].integer));

                                                                        flagoutput = 1;
                                                                        break;
                                                                  }
                                                            }

                                                         }


                                                 }

                                         }



                         }


                         return "Continue Rule 2";
        }
        else if ( prevInput2.name == "=" && input.type == "number" ) {
                         prevInput2 = input;
                         if (f == 1)
                         { //
                                   varsize.Add(prevInput2.name);
                                   Lvar[ct].integer = Convert.ToInt32(prevInput2.name);

                         }


                         return "Continue Rule 2";
        }
        else if ( prevInput2.type == "number" && input.name == ";" ) {
                         prevInput2 = new Token();
                         ct++;
                         return "Ok Rule 2";
        }
        else if ( prevInput2.type == "number" && operators.Contains( input.name ) ) {
                         if (f == 1)
                         { //
                                 number = Convert.ToInt32(prevInput2.name);
                                 number2 = Convert.ToInt32(prevInput2.name);
                                 prevtype = prevInput2.type;

                         }
                         prevInput2 = input;

                         return "Continue Rule 2";
        }
        else if ( prevInput2.type == "variable" && operators.Contains( input.name ) ) {

                         prevvar = prevInput2.name;
                         vartype = prevInput2.type;
                         fstop = 1;
                         prevInput2 = input;
                         return "Continue Rule 2";
        }
        else if ( operators.Contains( prevInput2.name ) && input.type == "number" ) {
                         if(prevInput2.name=="+")
               { //


                                 if (f == 1 && prevtype == "number" )
                                 { //

                                         Lvar[ct].integer = number + Convert.ToInt32(input.name);
                                         varsize.Add(Convert.ToString(Lvar[ct].integer));
                                         number2 = Convert.ToInt32(input.name);

                                 }
                                 if (f == 1&& vartype=="variable")
                                 {
                                         flagoutput = 0;
                                         for (int i = 0; i < Lvar.Count; i++)
                                         {
                                                 if (Lvar[i].name == prevtypevar && flagoutput == 0)
                                                 {
                                                         for (int j = 0; j < Lvar.Count; j++)
                                                         {
                                                                 if (Lvar[j].name == firstvar)
                                                                 {

                                                                        Lvar[j].integer = Lvar[i].integer +
            Convert.ToInt32(input.name);

        varsize.Add(Convert.ToString(Lvar[j].integer));

                                                                        flagoutput = 1;
                                                                        break;
                                                                 }

                                                         }
                                                         if (flagoutput == 1)
                                                         {
                                                                 break;
                                                         }


                                                 }
                         }

                     }

        }
        if (prevInput2.name == "-")
        {


                     if (f == 1&&prevtype == "number")
                     {

                         Lvar[ct].integer = number - Convert.ToInt32(input.name);
                         varsize.Add(Convert.ToString(Lvar[ct].integer));

                     }

                     if (f == 1 && vartype == "variable")
                     {
                         flagoutput = 0;
                         for (int i = 0; i < Lvar.Count; i++)
                         {
                             if (Lvar[i].name == prevtypevar && flagoutput == 0)
                             {
                                 for (int j = 0; j < Lvar.Count; j++)
                                 {
                                     if (Lvar[j].name == firstvar)
                                     {

                                         Lvar[j].integer = Lvar[i].integer -
Convert.ToInt32(input.name);

    varsize.Add(Convert.ToString(Lvar[j].integer));

                                flagoutput = 1;
                                break;
                            }

                        }
                        if (flagoutput == 1)
                        {
                            break;
                        }


                    }

                }

            }
```

```
                }
                if (prevInput2.name == "*" )
                {

                        if (f == 1 && prevtype == "number")
                        {

                            Lvar[ct].integer = number * Convert.ToInt32(input.name);
                            varsize.Add(Convert.ToString(Lvar[ct].integer));
                            number2 = Convert.ToInt32(input.name);

                        }
                        if (f == 1 && vartype == "variable")
              {
                            flagoutput = 0;
                        for (int i = 0; i < Lvar.Count; i++)
                        {
                            if (Lvar[i].name == prevtypevar && flagoutput == 0)
                            {
                                for (int j = 0; j < Lvar.Count; j++)
                                {
                                    if (Lvar[j].name == firstvar)
                                    {

                                        Lvar[j].integer = Lvar[i].integer *
Convert.ToInt32(input.name);

    varsize.Add(Convert.ToString(Lvar[j].integer));

                                        flagoutput = 1;
                                        break;
                                    }

                                }
                                if (flagoutput == 1)
                                {
                                    break;
                                }


                            }
                        }
                  }


                }
```

THIRD RULE:

```
string Rule3( Token input ) {
	List<string> comp_operators = new List<string>( new string[] { "==", "!=", "<=", "<", ">", ">=" 
} );
	List<string> bool_operators = new List<string>( new string[] { "&&", "||" } );
	if ( prevInput3.name == "" && input.name == "if" ) {
		prevInput3 = input;
		return "Start Rule 3";
	}
	else if ( prevInput3.name == "if" && input.name == "(" ) {
		prevInput3 = input;
		return "Continue Rule 3";
	}
	else if ( prevInput3.name == "(" && input.type == "variable" ) {
		prevInput3 = input;
		return "Continue Rule 3";
	}
	else if ( prevInput3.type == "variable" && comp_operators.Contains( input.name ) ) {
		prevInput3 = input;
		return "Continue Rule 3";
	}
	else if ( comp_operators.Contains( prevInput3.name ) && input.type == "number" ) {
		prevInput3 = input;
		return "Continue Rule 3";
	}
	else if ( comp_operators.Contains( prevInput3.name ) && input.type == "variable" ) {
		prevInput3 = input;
		return "Continue Rule 3";
	}
	else if ( prevInput3.type == "number" && bool_operators.Contains( input.name ) ) {
		prevInput3 = input;
		return "Continue Rule 3";
	}
	else if ( prevInput3.type == "variable" && bool_operators.Contains( input.name ) ) {
		prevInput3 = input;
		return "Continue Rule 3";
	}
	else if ( bool_operators.Contains( prevInput3.name ) && input.type == "variable" ) {
		prevInput3 = input;
		return "Continue Rule 3";
	}
	else if ( prevInput3.type == "number" && input.name == ")" ) {
		prevInput3 = input;
		return "Continue Rule 3";
	}
	else if ( prevInput3.type == "variable" && input.name == ")" ) {
		prevInput3 = input;
		return "Continue Rule 3";
	}
```

```
        else if ( prevInput3.name == ")" && input.name == "{" ) {
                prevInput3 = input;
                return "Continue Rule 3";
        }
        else if ( prevInput3.name == "{" && input.name == "}" ) {
                prevInput3 = new Token();
                return "Ok Rule 3";
        }
        else if ( prevInput3.name == "{" && input.name != "" ) {
                return "Continue Rule 3";
        }

        if ( prevInput3.name == "if" ) {
                prevInput3 = new Token();
                return "Error Expected '(' Rule 3";
        }
        if ( prevInput3.name == "(" ) {
                prevInput3 = new Token();
                return "Error Expected 'variable' Rule 3";
        }
        if ( prevInput3.type == "variable" ) {
                prevInput3 = new Token();
                return "Error Expected 'comp_operator' Or 'bool_operator' Or ')' Rule 3";
        }
        if ( comp_operators.Contains( prevInput3.name ) ) {
                prevInput3 = new Token();
                return "Error Expected 'number' Or 'variable' Rule 3";
        }
        if ( bool_operators.Contains( prevInput3.name ) ) {
                prevInput3 = new Token();
                return "Error Expected 'variable' Rule 3";
        }
        if ( prevInput3.type == "number" ) {
                prevInput3 = new Token();
                return "Error Expected 'bool_operator' Or ')' Rule 3";
        }
        if ( prevInput3.name == ")" ) {
                prevInput3 = new Token();
                return "Error Expected '{' Rule 3";
        }
        if ( prevInput3.name == "{" ) {
                prevInput3 = new Token();
                return "Error Expected '}' Rule 3";
        }

        prevInput3 = new Token();
        return "Error Rule 3";
}
```

• QUESTION NO 03:

Input with output

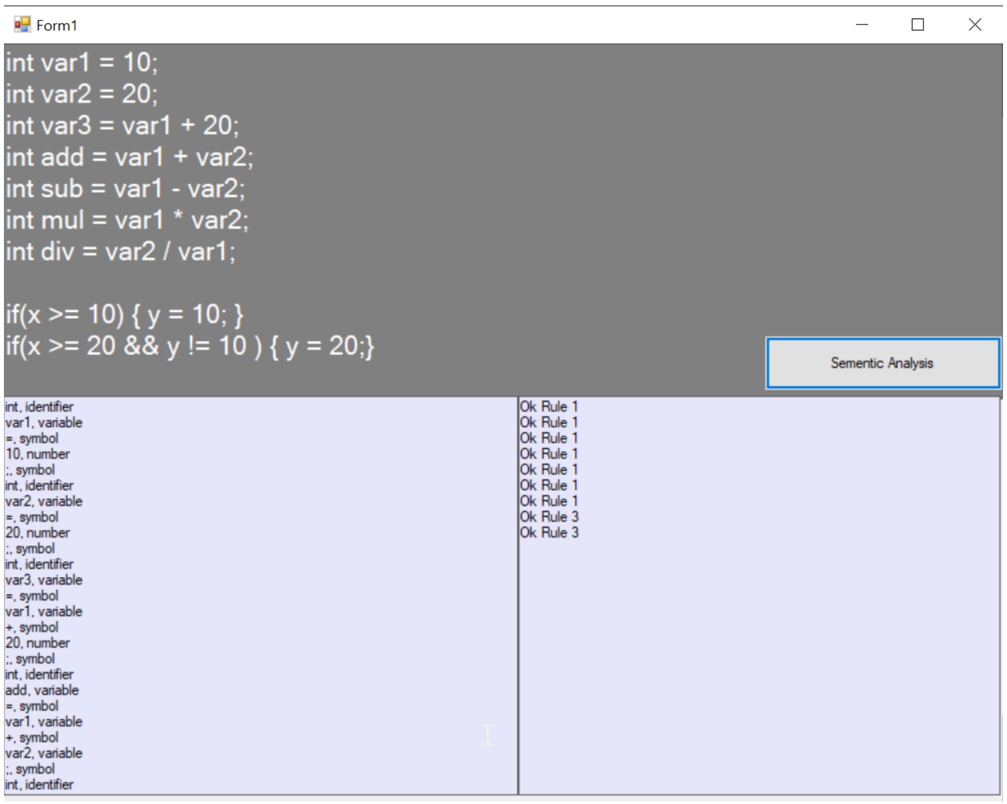
- **ANSWER:**

Process of Input execution:

1. The execution starts with the creation of Tokens from the source code (input) provided by the user.
2. Then these tokens pass through the three rules that are defined earlier to check whether it follows the rules or not.
3. Then if the tokens matched then it gives OK message otherwise it tells what the error is and what is expected to overcome the error.

INPUT WITH OUTPUT:

```
int var1 = 10;
int var2 = 20;
int var3 = var1 + 20;
int add = var1 + var2;
int sub = var1 - var2;
int mul = var1 * var2;
int div = var2 / var1;
if(x >= 10) { y = 10; }
if(x >= 20 && y != 10 ) { y = 20;}
```



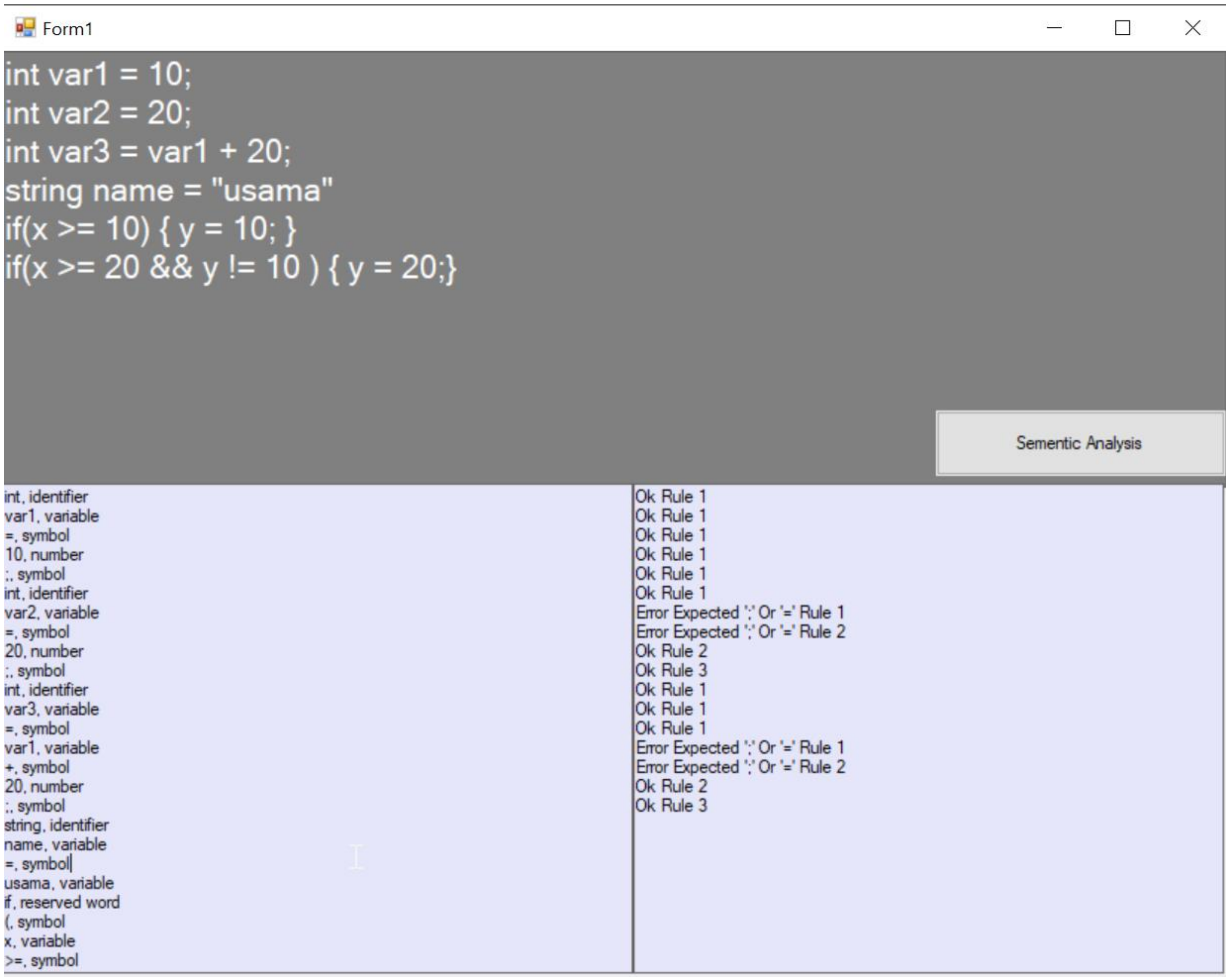
```
int var1 = 10;
int var2 = 20;
int var3 = var1 + 20;
int add = var1 + var2;
int sub = var1 - var2;
int mul = var1 * var2;
int div = var2 / var1;

if(x >= 10) { y = 10; }
if(x >= 20 && y != 10 ) { y = 20;}
```

```
int, identifier
var1, variable
=, symbol
10, number
;, symbol
int, identifier
var2, variable
=, symbol
20, number
;, symbol
int, identifier
var3, variable
=, symbol
var1, variable
+, symbol
20, number
;, symbol
int, identifier
add, variable
=, symbol
var1, variable
+, symbol
var2, variable
;, symbol
int, identifier
```

```
Ok Rule 1
Ok Rule 1
Ok Rule 1
Ok Rule 1
Ok Rule 1
Ok Rule 1
Ok Rule 1
Ok Rule 3
Ok Rule 3
```

2ND INPUT WITH OUTPUT:

```
int var1 = 10;

int var2 = 20;

int var3 = var1 + 20;

string name = "usama"

if(x >= 10) { y = 10; }

if(x >= 20 && y != 10 ) { y = 20;}
```



```
int var1 = 10;
int var2 = 20;
int var3 = var1 + 20;
string name = "usama"
if(x >= 10) { y = 10; }
if(x >= 20 && y != 10 ) { y = 20;}
```

```
int, identifier
var1, variable
=, symbol
10, number
;, symbol
int, identifier
var2, variable
=, symbol
20, number
;, symbol
int, identifier
var3, variable
=, symbol
var1, variable
+, symbol
20, number
;, symbol
string, identifier
name, variable
=, symbol
usama, variable
if, reserved word
(, symbol
x, variable
>=, symbol
```

```
Ok Rule 1
Ok Rule 1
Ok Rule 1
Ok Rule 1
Ok Rule 1
Ok Rule 1
Error Expected ';' Or '=' Rule 1
Error Expected ';' Or '=' Rule 2
Ok Rule 2
Ok Rule 3
Ok Rule 1
Ok Rule 1
Ok Rule 1
Error Expected ';' Or '=' Rule 1
Error Expected ';' Or '=' Rule 2
Ok Rule 2
Ok Rule 3
```

The error shows in right side window is missing semicolon (symbol) that violates the rule that are defined earlier.

- ## QUESTION NO 4:

How functions works step by step?

- ## ANSWER:

SemanticAnalyzer:

Following are the steps involve in SemanticAnalyzer function:

1. **Loop through Tokens:**

The code uses a for loop to iterate over the list of tokens.

2. **Rule Selection:**

Inside the loop, it checks the current token against three rules (Rule1, Rule2, and Rule3) to determine which rule to apply.

If any of the rules indicate a match with the current token (by returning a string starting with "Start"), it sets the selectedRule variable accordingly.

3. **Rule Application:**

If a rule is selected, it applies the corresponding rule to the current token.

If the rule application returns a state that starts with "Ok" or "Error," it adds the state to the errors list and resets selectedRule to 0, indicating that the next token should be processed using rule selection.

4. **Backtracking for Unmatched Token:**

If selectedRule is still set after processing all tokens, it means that the last rule is still active. In that case, it applies the corresponding rule to a new (empty) token and adds the result to the errors list.

5. **Errors:**

The final result is a list of error messages in the errors list, indicating issues found during semantic analysis.

• QUESTION NO 05:

What challenges your faces during the project?

- **ANSWER:**

Implementing complex semantic rules:

Handling semantic rules beyond basic operations can be challenging.

This includes function calls, loops, conditional statements, and type checking.

Ensuring correct code execution involves careful rule implementation.

LEXICAL ANALYZER:

QUESTION NO 01:

Give the brief explanation of the CC final project?

ANSWER:

This project involves the creation of a basic interpreter for arithmetic expressions using C#. The interpreter is designed to process and evaluate mathematical expressions entered as strings, breaking them down into individual components through a process called lexical analysis or tokenization. The main components of this interpreter include the Lexer class, the Parser class, and associated supporting structures.

In the initial stage, the Lexer class is responsible for tokenizing the input string, identifying and categorizing individual elements such as numbers, operators (addition, subtraction, multiplication, and division), and parentheses. The Lexer utilizes the Token class to represent each token, which includes properties for the token's type (TokenType) and its actual value.

Following the tokenization phase, the Parser class takes the sequence of tokens produced by the Lexer and interprets the syntax of the arithmetic expressions. The parsing process, implemented using a recursive descent parsing approach, involves breaking down expressions into terms, factors, and numbers. The Parser class includes methods such as ParseExpression, ParseTerm, and ParseFactor to navigate through the token stream and construct a syntactic representation of the input expression.

QUESTION NO 02:

Give two functionalities of the project?

ANSWER:

The two functionalities are given below:

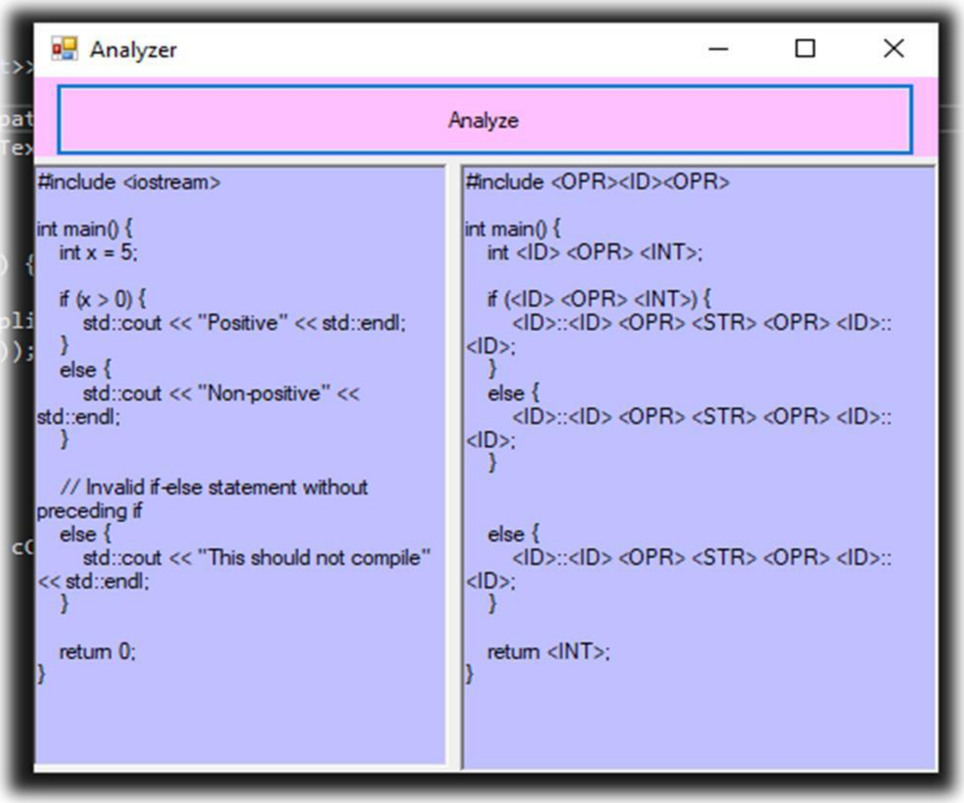
- The first essential functionality of the arithmetic expression interpreter is tokenization, implemented within the Lexer class. Tokenization involves the systematic breakdown of the input string into individual components, known as tokens. In the GetNextToken method of the Lexer class, each character of the input string is examined, and its type is determined, resulting in the creation of corresponding tokens. These tokens represent fundamental units such as numbers, operators (such as plus or minus), and parentheses. The tokenization process continues until the end of the input is reached, signified by the generation of an EndOfFile token. This initial phase is crucial for establishing a structured understanding of the arithmetic expression's composition.
- The second core functionality lies in the parsing phase, implemented in the Parser class. This component focuses on the interpretation of the syntactic

structure of arithmetic expressions using the sequence of tokens produced during tokenization. Employing a recursive descent parsing approach, the Parser class dissects the expressions into key components, such as terms and factors. The Parse method acts as the initiator of the parsing process, starting with high-level constructs like expressions and recursively delving into subcomponents. For instance, the ParseExpression method manages addition and subtraction operations, while the ParseTerm and ParseFactor methods handle multiplication, division, and individual numbers. The outcome of the parsing process is the evaluation of the arithmetic expression, providing a comprehensive understanding of its intended meaning.

QUESTION NO 03:

Working of project with output

ANSWER:



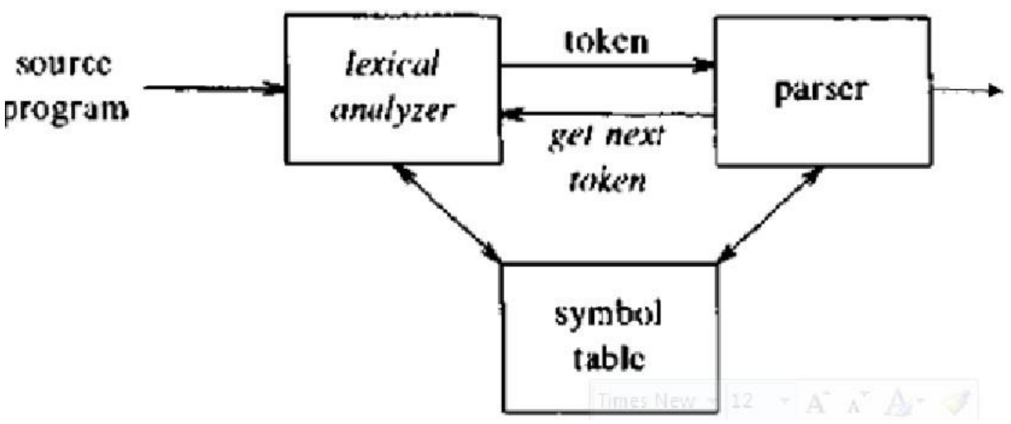
```
#include <iostream>

int main() {
   int x = 5;

   if (x > 0) {
      std::cout << "Positive" << std::endl;
   }
   else {
      std::cout << "Non-positive" <<
std::endl;
   }

   // Invalid if-else statement without
preceding if
   else {
      std::cout << "This should not compile"
<< std::endl;
   }

   return 0;
}
```

```
#include <OPR><ID><OPR>

int main() {
   int <ID> <OPR> <INT>;

   if (<ID> <OPR> <INT>) {
      <ID>::<ID> <OPR> <STR> <OPR> <ID>::
<ID>;
   }
   else {
      <ID>::<ID> <OPR> <STR> <OPR> <ID>::
<ID>;
   }


   else {
      <ID>::<ID> <OPR> <STR> <OPR> <ID>::
<ID>;
   }

   return <INT>;
}
```

QUESTION NO 04:

Class diagram of project

ANSWER:



QUESTION NO 05:

Challenges faced during this project?

ANSWER:

- **Regular Expression Complexity:**

Crafting accurate and efficient regular expressions for token recognition can be challenging. The complexity of expressions increases with the diversity of the input language, and striking the right balance between specificity and generality is crucial.

- **Ambiguities in Token Definitions:**

Defining tokens with potential ambiguities can lead to challenges. For example, distinguishing between unary and binary operators or handling numbers with decimal points requires careful consideration to avoid misinterpretations.

- **Whitespace and Comments Handling:**

Managing whitespace and comments appropriately is often overlooked but essential. Incorrect handling of these elements can affect tokenization and lead to unexpected behavior in the interpreter.