Final Project

Compiler Construction

Name: Junaid Ali , Muhammad Yaseen

Regno: FA20-BCS-008, FA20-BCS-076

Submitted to: Sir Bilal Haider Bukhari

Dated: 26-Dec-23

Subject: Compiler Construction

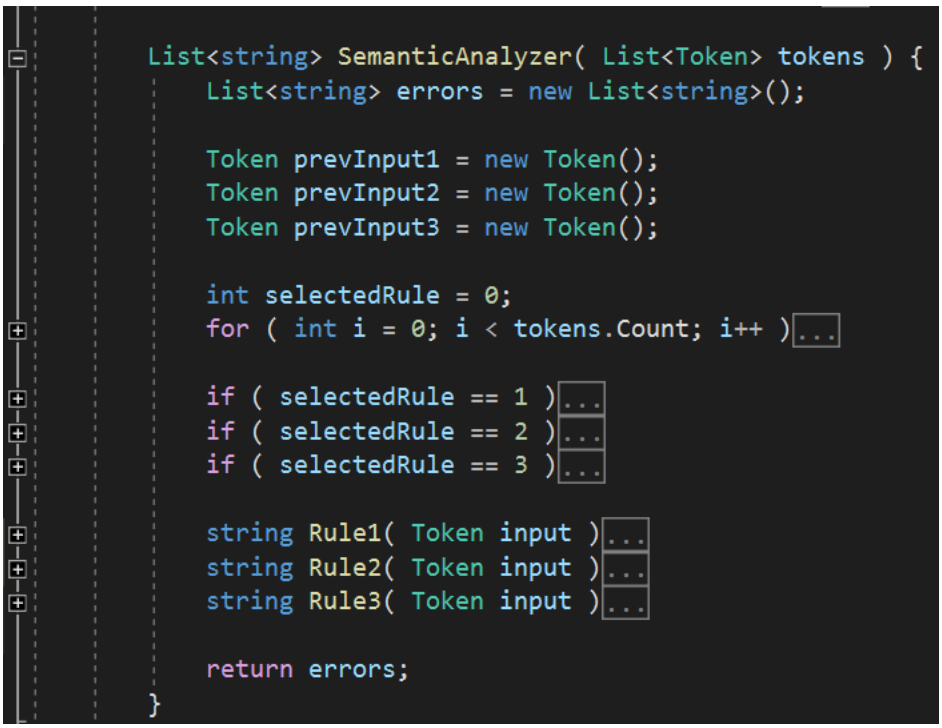
COMSATS UNIVERSITY ISLAMABAD ATTOCK CAMPUS

Brief of the Project

In this Lab terminal project, we have a semantic analyzer is developed so that errors in the code block could be identified, the SemanticAnalyzer() function takes the list of tokens identified by the Scanner, and checks if the tokens sequence follow any of the three rules provided in the project description using Top-Down Parsing, then it return a list of the errors found.

The three rules:

An overview of the SemanticAnalyzer() function with collapsed code:



The SemanticAnalyzer() function has a main “for” loop that iterates over the given list of tokens, it test which of the three rules will start accepting the i-th token, once a rule starts accepting, this rule becomes the selected rule, it will remain selected until it either returns an error or returns an “Ok”, which means the last sequence of tokens did follow the rule correctly:

```
List<string> SemanticAnalyzer( List<Token> tokens ) {
    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;
        }
    }
}
```

In Each rule takes a token from each loop iteration and stores the previous token it has seen inside previous Input. Inside each rule, the upper half contains the conditions that begin, ends and continues the application of that rule, the lower half catches the errors and returns what the rule expected instead.

Role 1,2,3:

```
string Rule1( Token input ) {
    if ( prevInput1.name == "" && input.type == "identifier" )...
    else if ( prevInput1.type == "identifier" ) {
        string state = Rule2( input );
        if ( state.StartsWith( "Ok" ) )...
        if ( state != "Error Rule 2" )...
    }
    if ( prevInput1.type == "identifier" )...
    prevInput1 = new Token();
    return "Error Rule 1";
}
```

```
string Rule2( Token input ) {
    List<string> operators = new List<string>( new string[] { "+", "-", "/", "%", "*" } );
    if ( prevInput2.name == "" && input.type == "variable" )...
    else if ( prevInput2.type == "variable" && input.name == ";" )...
    else if ( prevInput2.type == "variable" && input.name == "=" )...
    else if ( prevInput2.name == "=" && input.type == "variable" )...
    else if ( prevInput2.name == "=" && input.type == "number" )...
    else if ( prevInput2.type == "number" && input.name == ";" )...
    else if ( prevInput2.type == "number" && operators.Contains( input.name ) )...
    else if ( prevInput2.type == "variable" && operators.Contains( input.name ) )...
    else if ( operators.Contains( prevInput2.name ) && input.type == "number" )...
    else if ( operators.Contains( prevInput2.name ) && input.type == "variable" )...

    if ( prevInput2.type == "variable" )...
    if ( prevInput2.name == "=" )...
    if ( prevInput2.type == "variable" )...
    if ( prevInput2.type == "number" )...
    if ( operators.Contains( prevInput2.name ) )...
    prevInput2 = new Token();
    return "Error Rule 2";
}
```

```
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" )...
    else if ( prevInput3.name == "if" && input.name == "(" )...
    else if ( prevInput3.name == "(" && input.type == "variable" )...
    else if ( prevInput3.type == "variable" && comp_operators.Contains( input.name ) )...
    else if ( comp_operators.Contains( prevInput3.name ) && input.type == "number" )...
    else if ( comp_operators.Contains( prevInput3.name ) && input.type == "variable" )...
    else if ( prevInput3.type == "number" && bool_operators.Contains( input.name ) )...
    else if ( prevInput3.type == "variable" && bool_operators.Contains( input.name ) )...
    else if ( bool_operators.Contains( prevInput3.name ) && input.type == "variable" )...
    else if ( prevInput3.type == "number" && input.name == ")" )...
    else if ( prevInput3.type == "variable" && input.name == ")" )...
    else if ( prevInput3.name == ")" && input.name == "{" )...
    else if ( prevInput3.name == "{" && input.name == "}" )...
    else if ( prevInput3.name == "{" && input.name != "" )...

    if ( prevInput3.name == "if" )...
    if ( prevInput3.name == "(" )...
    if ( prevInput3.type == "variable" )...
    if ( comp_operators.Contains( prevInput3.name ) )...
    if ( bool_operators.Contains( prevInput3.name ) )...
    if ( prevInput3.type == "number" )...
    if ( prevInput3.name == ")" )...
    if ( prevInput3.name == "{" )...

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

Function Explanations Along with Screenshot:

```
List<string> Splitter( string sourceCode ) {

    List<string> splitSourceCode = new List<string>();

    Regex RE = new Regex( @"\d+\.\d+|\'.*\'|\w+|\(|\)|\++|-+|\*|%|,|;|&+|\|+|<=|<|>=|>|==|=|!=|!|\{|\}|\/" );

    foreach ( Match m in RE.Matches( sourceCode ) ) {
        splitSourceCode.Add( m.Value );
    }

    return splitSourceCode;
}
```

As this Function is used to process of converting a sequence of characters (Code Block) input by user into a sequence of tokens.

Instead of DFA or split, we have decided to use regular expression match method because it allows you to specify the tokens you are looking for like the image and it is very simple.

Splitter : return list of matches strings.

Create class Token to store name and type of token that we will use later.

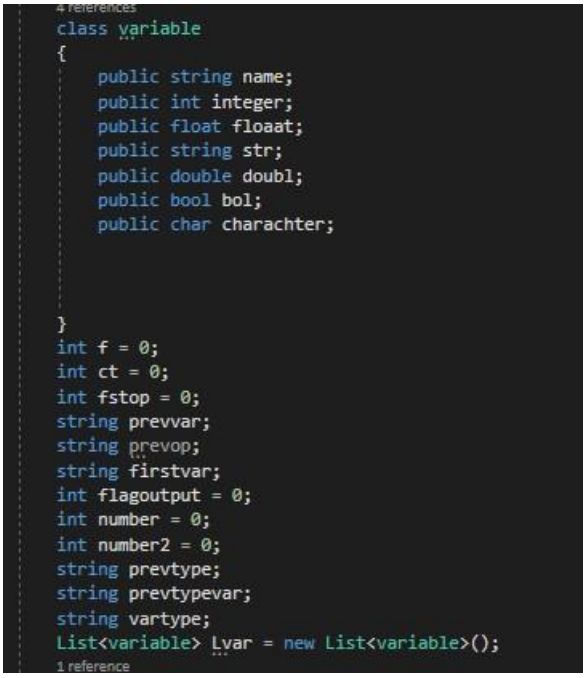
```
class Token
{
    public string name = "";
    public string type = "";


    public Token( string name, string type ) {
        this.name = name;
        this.type = type;
    }
}
```

```
List<Token> Scanner( List<string> splitCode ) {

    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 ) {
            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;
}
```

How Function Works step by step:

In this phase we save the variable value in the memory and every variable shows on the data grid view.

Our class variable and list of class variables.



```
4 references
class variable
{
    public string name;
    public int integer;
    public float floaat;
    public string str;
    public double doubl;
    public bool bol;
    public char charachter;



}
int f = 0;
int ct = 0;
int fstop = 0;
string prevvar;
string prevop;
string firstvar;
int flagoutput = 0;
int number = 0;
int number2 = 0;
string prevtype;
string prevtypevar;
string vartype;
List<variable> Lvar = new List<variable>();
1 reference
```

First in rule1 function we put a flag (f=1) to know that the identifier is integer

```
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;
        }
```

In rule 2 function We made first var to catch the variable before “=”

prevvar to catch the first variable after “=”

```
    else if ( prevInput2.type == "variable" && input.name == "=" ) {
        firstvar = prevInput2.name;
        prevvar = prevInput2.name;
        prevInput2 = input;

        return "Continue Rule 2";
    }
```

In this condition the variable becomes any integer value

```
else if ( prevInput2.name == "=" && input.type == "number" ) {
    prevInput2 = input;
    if (f == 1)
    {

         varsize.Add(prevInput2.name);
          Lvar[ct].integer = Convert.ToInt32(prevInput2.name);









    }
```

Ex: int x=3;

In this condition integer variable becomes another integer variable

Ex: int x= y;

```
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;
                        }
                        }

                    }


                }

            }



    }
```

In this condition we catch the number before the operator.

Number becomes this number.

```
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";
}
```

In this condition we catch the variable before the operator.

Prevvar becomes this variable.

```
else if ( prevInput2.type == "variable" && operators.Contains( input.name ) ) {

    prevvar = prevInput2.name;
    vartype = prevInput2.type;
    fstop = 1;
    prevInput2 = input;
    return "Continue Rule 2";
}
```

In this condition we take the values of “+” operator when

Num+ num

And

Integer variable + num

```
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;
                    }


                }

            }

        }

    }
```

In this condition we take the values of “-” operator when

Num-num

And

Integer variable-num

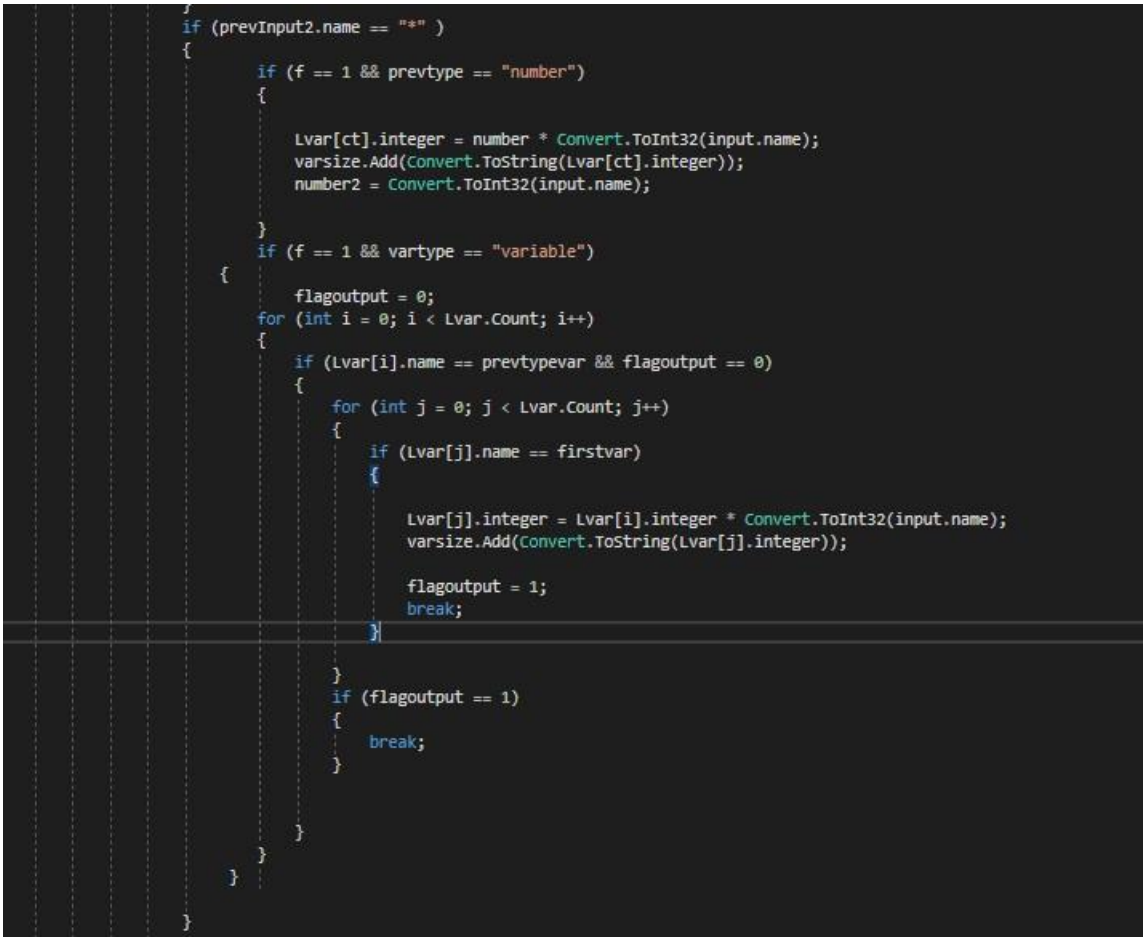
```
if (prevInput2.name == "-")
{
    if (f == 1&&prevtype == "number")
    {
        Lvar[ct].integer = number - Convert.ToInt32(input.name);
        varsize.Add(Convert.ToString(Lvar[ct].integer));

    }

    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;
                }


            }

        }

    }

}
```

In this condition we take the values of “*” operator when

Num*num

And

Integer variable*num



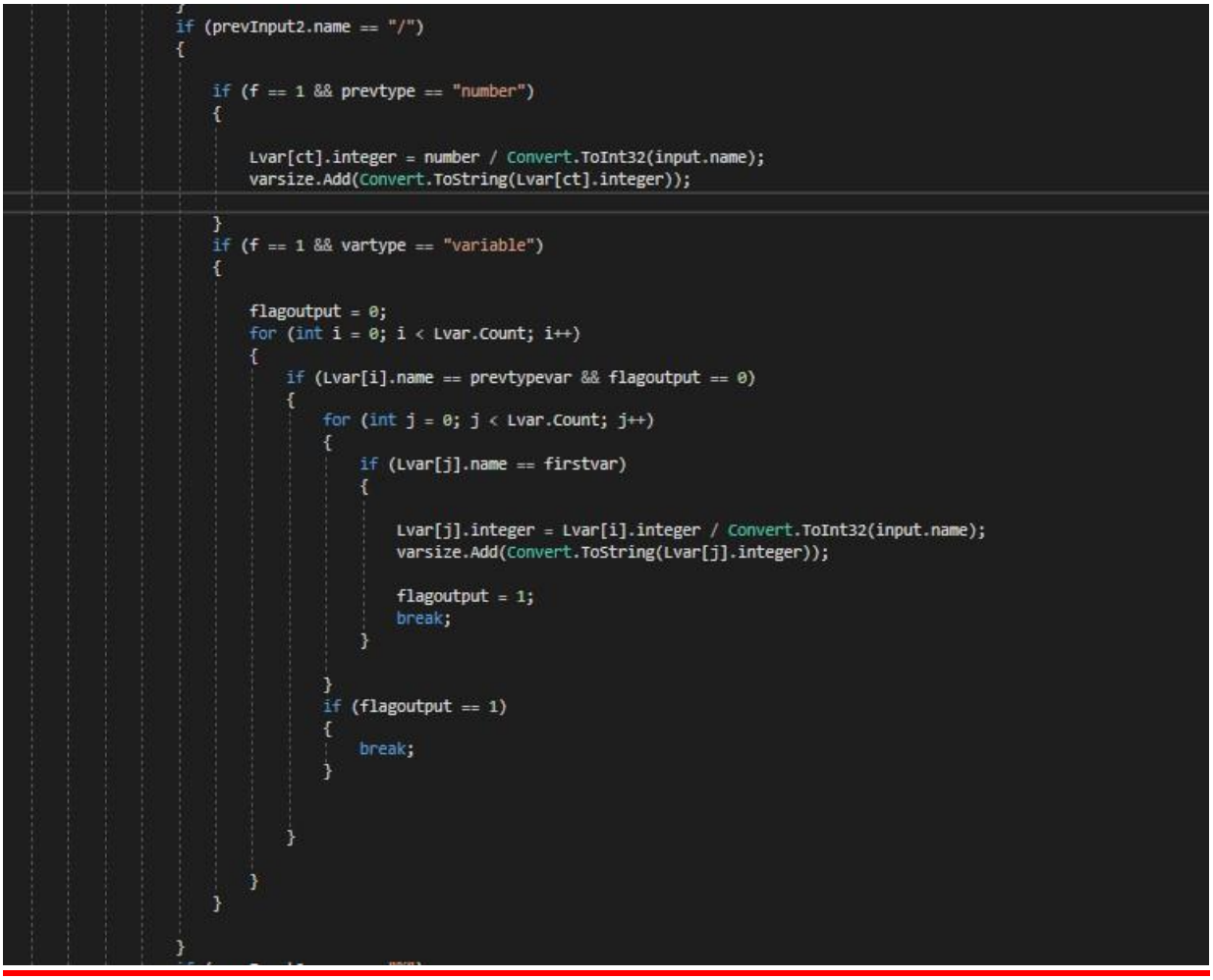
```
if (prevInput2.name == "*" )
{
        if (f == 1 && prevtype == "number")
        {

            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;
                }


            }
        }
     }

}
```

In this condition we take the values of “/” operator when

Num/num

And

Integer variable/num



```
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;
                }


            }

        }
    }

}
```

In this condition we take the values of “%” operator when

Num % num

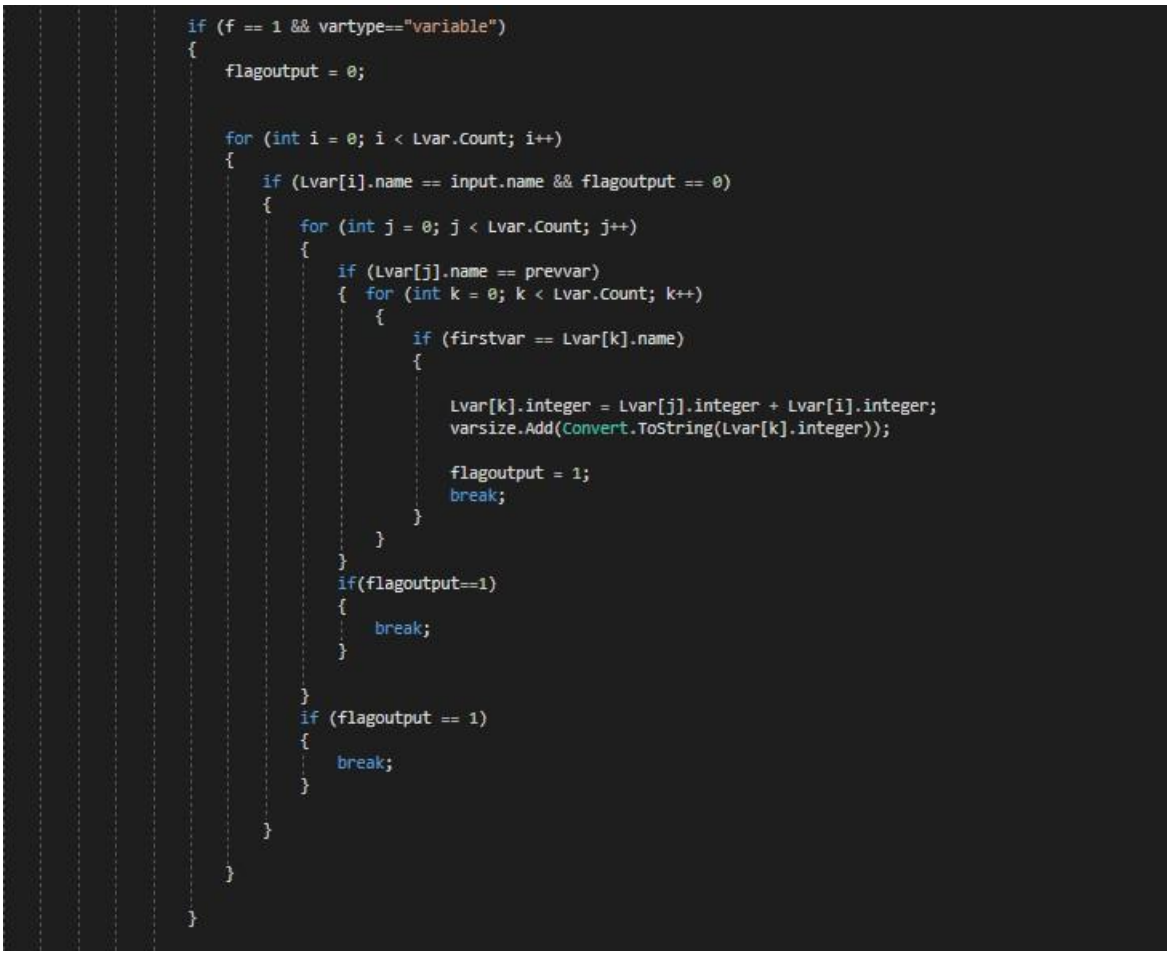
And

Integer variable % num

```
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;
                }


            }

        }
    }

}
```

In this condition we take the values of “+” operator when

Integer variable+ Integer variable



```
if (f == 1 && vartype=="variable")
{
    flagoutput = 0;


    for (int i = 0; i < Lvar.Count; i++)
    {
        if (Lvar[i].name == input.name && flagoutput == 0)
        {
            for (int j = 0; j < Lvar.Count; j++)
            {
                if (Lvar[j].name == prevvar)
                {  for (int k = 0; k < Lvar.Count; k++)
                    {
                        if (firstvar == Lvar[k].name)
                        {

                            Lvar[k].integer = Lvar[j].integer + Lvar[i].integer;
                            varsize.Add(Convert.ToString(Lvar[k].integer));

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

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

        }

    }

}
```

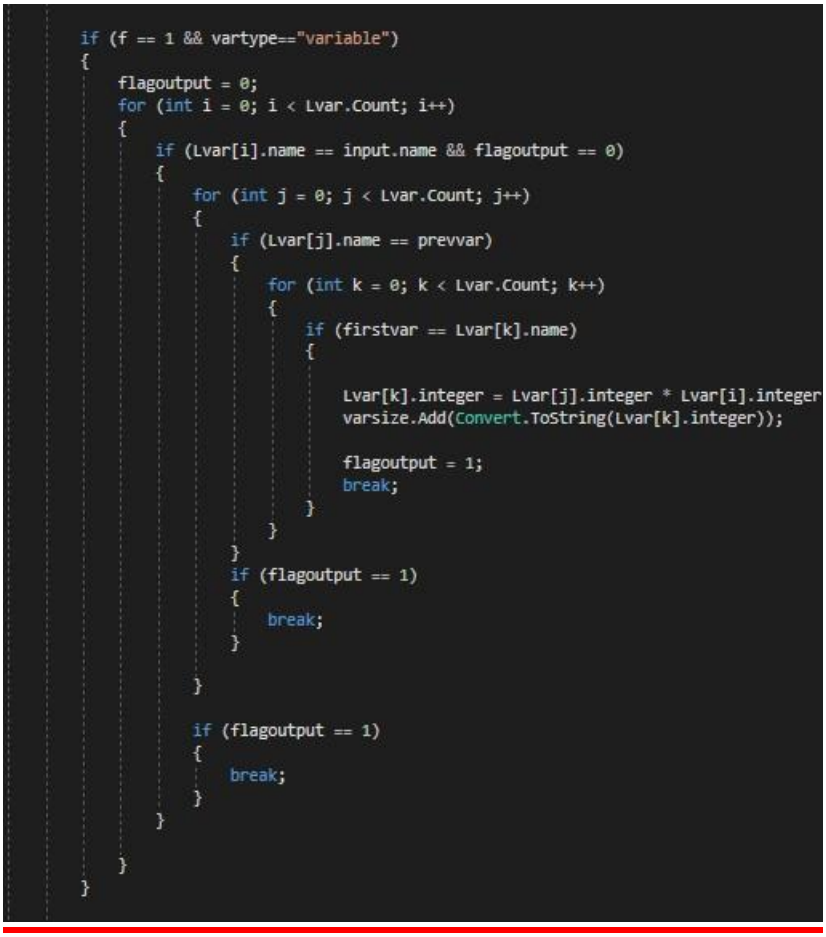
In this condition we take the values of “-” operator when

Integer variable- Integer variable

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

                            Lvar[k].integer = Lvar[j].integer - Lvar[i].integer;
                            varsize.Add(Convert.ToString(Lvar[k].integer));

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

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


        }

    }
}
```

In this condition we take the values of “*” operator when

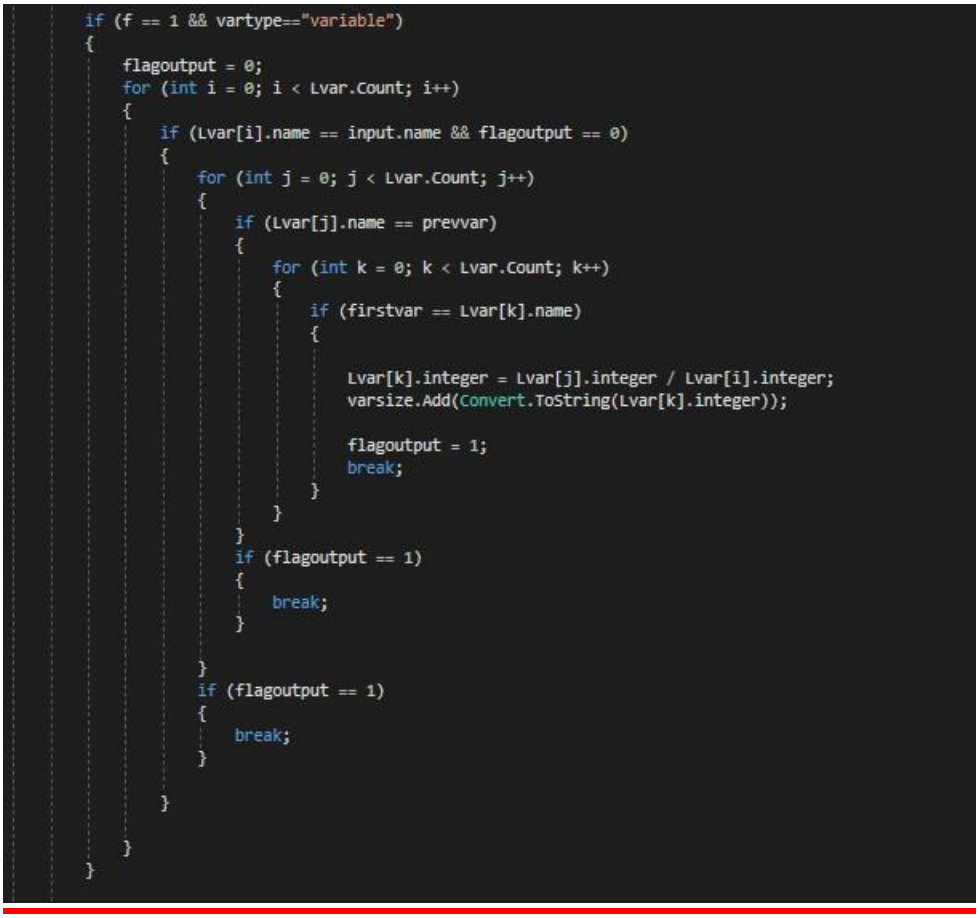
Integer variable*Integer variable



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

                            Lvar[k].integer = Lvar[j].integer * Lvar[i].integer
                            varsize.Add(Convert.ToString(Lvar[k].integer));

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

            }

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

    }
}
```

In this condition we take the values of “/” operator when

Integer variable/Integer variable



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

                            Lvar[k].integer = Lvar[j].integer / Lvar[i].integer;
                            varsize.Add(Convert.ToString(Lvar[k].integer));

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

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

        }

    }
}
```

In this condition we take the values of “%” operator when

Integer variable% Integer variable

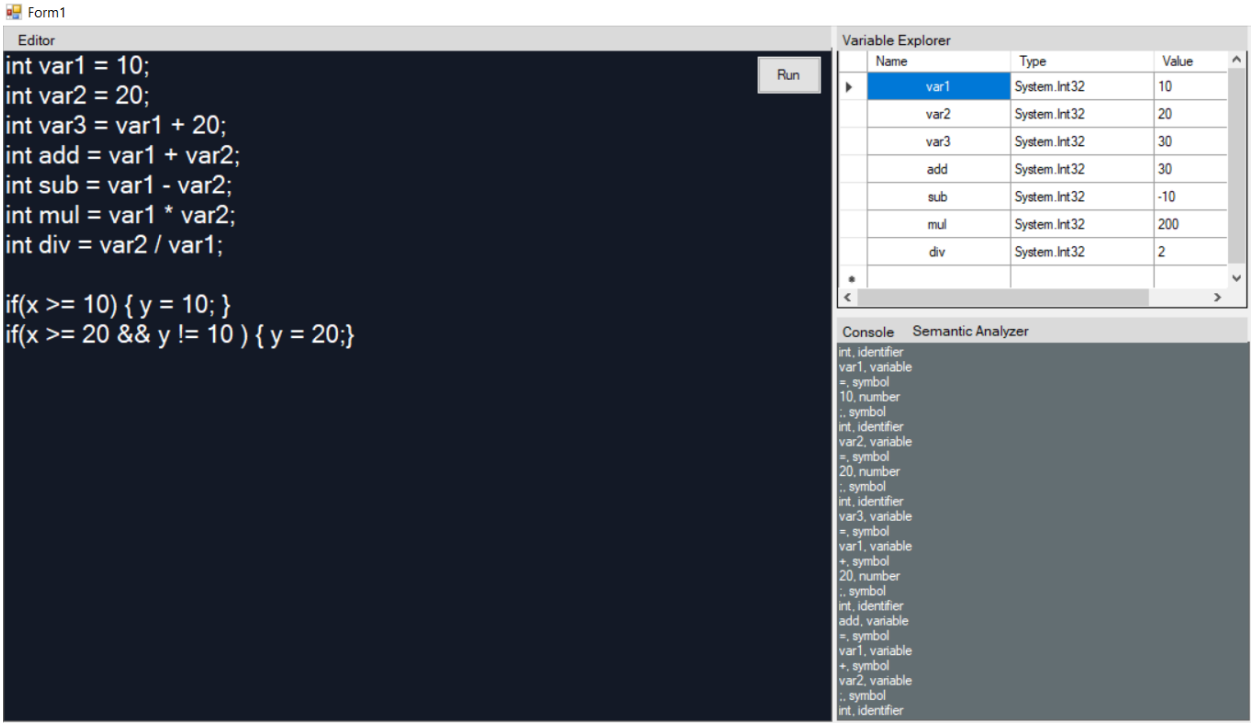
```
if (f == 1 && vartype=="variable")
{
    flagoutput = 0;
    for (int i = 0; i < Lvar.Count; i++)
    {
        if (Lvar[i].name == input.name && flagoutput == 0)
        {
            for (int j = 0; j < Lvar.Count; j++)
            {
                if (Lvar[j].name == prevvar)
               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

                            Lvar[k].integer = Lvar[j].integer % Lvar[i].integer;
                            varsize.Add(Convert.ToString(Lvar[k].integer));

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

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

        }

    }
}
```

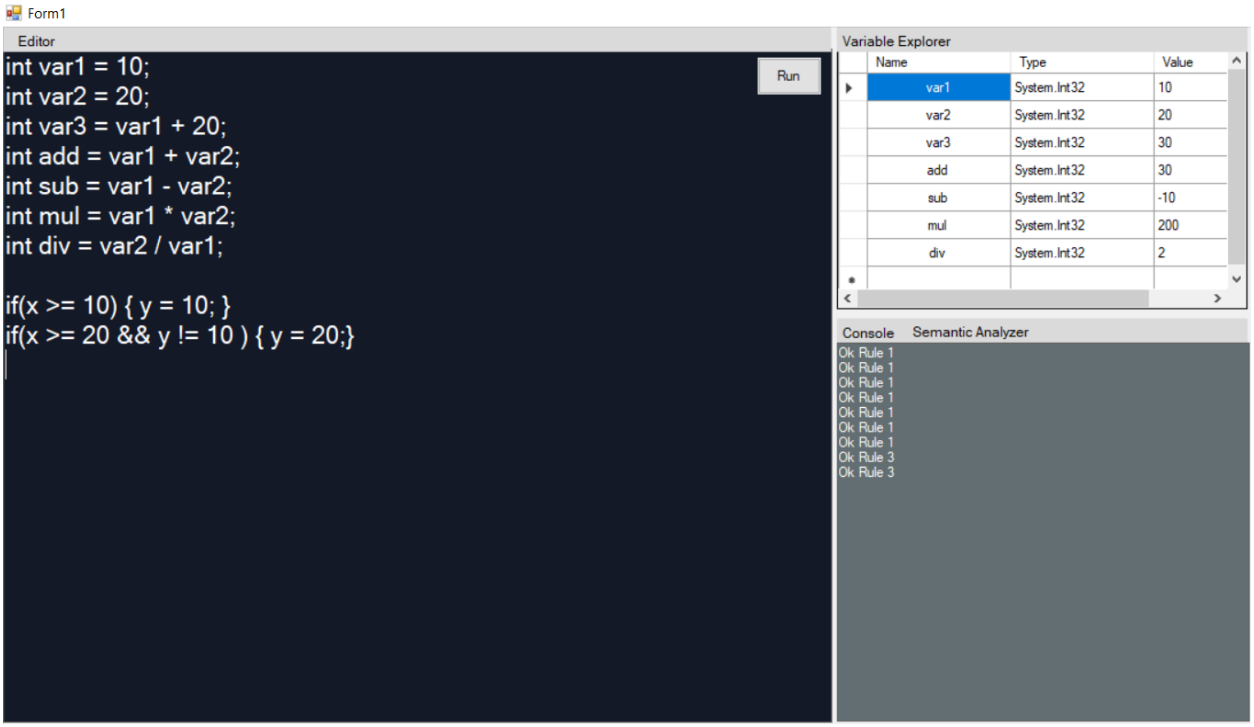
Test Case :

```
int x=90;
int z=20;
int y=20-z;
y=x/10;
```

Name	Type	Value
x	System.Int32	90
z	System.Int32	20
y	System.Int32	9

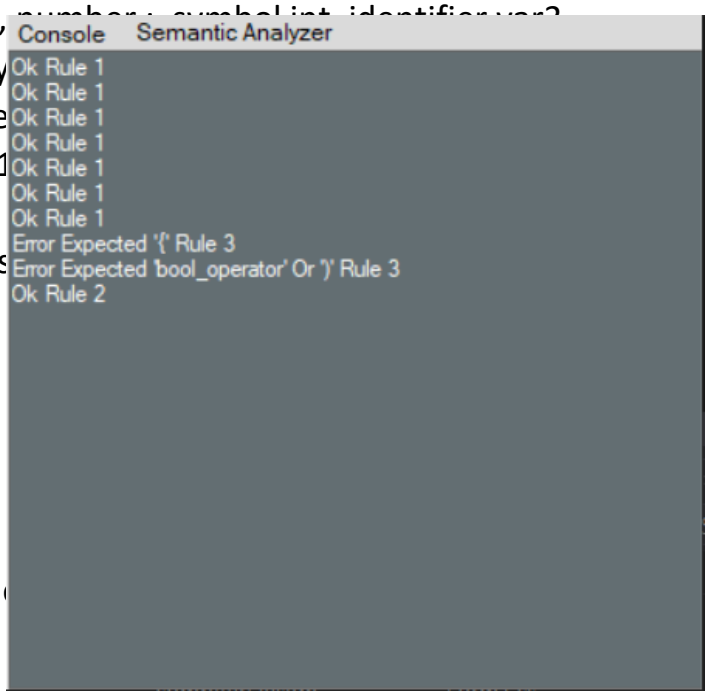
User Input Phase:





Output(generated by compiler) :

Console : 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, number ;, symbol identifier sub, variable =, symbol var1, variable -, symbol var2, number ;, symbol int, identifier mul, variable =, symbol var1, variable *, symbol var2, number ;, symbol int, identifier div, variable =, symbol var2, variable /, symbol var1, number ;, symbol if, reserved word (, symbol



Semantic Analyzer

Checking the lines with roles

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

If we replace by this input then,

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

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

with

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

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

the analyzer output :

What Challenges we Faced during the project :

While creating this project which contain semantic analyzer in which we give input from the user and scans the entire code and informs us weather the given input is correct or not and also tell us a step by step explanations of a given inputs

Statements.

The challenges we faced while creating this project is to dealing with the syntax and logical errors and also understanding the Language grammar, and doing their lexical analysis is difficult job for us so firstly we understood these all and right after that we finally constructed the semantic analyzer and project runs successfully.
