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:

- **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";
}
```

LEXICAL ANALYZER:

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.

CODES:

LEXICAL ANALYZER (.cs file):

```
using System;
using System.Collections.Generic;
namespace Analyzer {
   class Analyze {
      List<List<int>> rules = new List<List<int>>();
      private void loadTransitionTable(string path) {
         string text = System.IO.File.ReadAllText(path);
         if(text.Length < 2) {
            throw new Exception();
         }
         foreach(var item in text.Split('\n')) {
            var temp = new List<int>();
            foreach(var itm in item.Trim().Split(' ')) {
               temp.Add(Convert.ToInt32(itm));
            }
            rules.Add(temp);
         }
      }

      private int getNextState(int iState,char cChar) {
         if(char.IsLetter(cChar))
            return rules[iState][1];
         else if(char.IsDigit(cChar))
            return rules[iState][2];
         else if(cChar == '.')
            return rules[iState][3];
         else if(cChar == '"')
            return rules[iState][4];
         else if(cChar == '\'')
            return rules[iState][5];
         else if(cChar == '_')
            return rules[iState][6];
         else if(cChar == '+')
            return rules[iState][7];
         else if(cChar == '=')
            return rules[iState][8];
         else if(cChar == '-')
            return rules[iState][9];
         else if(cChar == '%')
            return rules[iState][10];
         else if(cChar == '!')
            return rules[iState][11];
         else if(cChar == '>')
            return rules[iState][12];
         else if(cChar == '<')
            return rules[iState][13];
         else if(cChar == '/')
            return rules[iState][14];
         return rules[iState][0];
      }

      private bool isKeyword(string sToken) {
         if((sToken).Length > 16 || (sToken).Length == 0)
            return false;
         var sKeywords = new List<string>(){

"using","import","include","asm","auto","bool","break","case","catch","char","class","const","const_cast"
,
            "continue","default","delete","do","double","dynamic_cast","else","enum","explicit",
            "export","extern","false","float","for","friend","goto","if","inline","int","long",
            "main","mutable","namespace","new","operator","private","protected","public",
            "register","reinterpret_cast","return","short","signed","sizeof","static",
            "static_cast","struct","switch","template","this","throw","true","try","typedef",
            "typeid","typename","union","unsigned","using","virtual","void","volatile","wchar_t","while"};
         return sKeywords.Exists(element => (sToken.ToLower()) == element);
      }
      public string Result(string txt,string tt = @"matrix.txt") {
         try {
            loadTransitionTable(tt);
         }
         catch(Exception) {
            return "Unable to open the input file.\nPress any key to exit.\n";
         }
         if(txt.Length == 0)
            return "";
         var result = "";
         int txtIndex = 0,iState=0;
         char cTemp = txt[txtIndex], cChar = ' ';
         string sToken = "";
         bool flag = true;
         ///////
         txt += " ";
         while(txtIndex != txt.Length) {
            if(flag) {
               cChar = cTemp;
               if(txt.Length - 1 == txtIndex)
                  return result + cChar;
               cTemp = txt[++txtIndex];
            }
            else
              flag = true;
```

```
#region Filter out comments
//CMNT
if(cChar == '/' && cTemp == '/') {
   if(txt.Length - 1 == txtIndex)
      return result;
   while(txt[++txtIndex] != '\n') {
      if(txt.Length == txtIndex)
         return result;
   }
   result += '\r';
   if(txt.Length - 1 != txtIndex)
      cTemp = txt[++txtIndex];
   continue;
}
if(cChar == '/' && cTemp == '*') {
   if(txt.Length - 1 == txtIndex)
      return result;
   cTemp = txt[++txtIndex];
   do {
      cChar = cTemp;
      if(txt.Length - 1 == txtIndex)
         return result + cChar;
      cTemp = txt[++txtIndex];
   } while(cChar != '*' && cTemp != '/');
   result += '\r';
   if(txt.Length - 1 != txtIndex)
      cTemp = txt[++txtIndex];
   continue;
}
#endregion
iState = getNextState(iState,cChar);
switch(iState) {
   case 0:
      result += cChar;
      iState = 0;
      sToken = "";
      break;
   case 1:
   case 3:
   case 5:
   case 7:
   case 10:
   case 14:
   case 18:
   case 25:
   case 26:
      sToken += cChar;
      break;
   case 2:
      if(isKeyword(sToken))
         result += sToken;
```

```
        else
            result += "<ID>";
        iState = 0;
        flag=false;
        sToken = "";
        break;
    case 4:
        result += "<INT>";
        iState = 0;
        flag=false;
        sToken = "";
        break;
    case 6:
        result += "<FLOAT>";
        iState = 0;
        flag=false;
        sToken = "";
        break;
    case 8:
        result += "<STR>";
        iState = 0;
        sToken = "";
        break;
    case 9:
    case 11:
    case 12:
    case 13:
    case 15:
    case 16:
    case 17:
    case 19:
    case 20:
    case 21:
    case 22:
    case 23:
    case 24:
    case 27:
    case 28:
        result += "<OPR>";
        if(cChar != '+' && cChar != '-' && cChar != '/'
            && cChar != '>' && cChar != '<' && cChar != '=')
            flag=false;
        iState = 0;
        sToken = "";
        break;
    case 30:
    case 33:
        iState = 0;
        sToken = "";
        break;
}
```

```
            }
            return result;
        }
    }
}
```