GOVERNMENT COLLEGE UNIVERSITY, LAHORE

COMPILER CONSTRUCTION ASSIGNMENT # 01

Written and Submitted By:

Muhammad Jawad Amin **0175-BSCS-2020**

Submitted To:

Sir Atif Ishaq **(Compiler Teacher)**

Section and Semester:

Section (A1) **Five (V)**

INTRODUCTION:

In this document I have explained the working of a **Basic Lexical Analyzer** which I have constructed myself using the **C++** Language. Basically, a Lexical Analyzer is **First Phase** of a compiler in which it takes stream of characters as input and convert it into tokens. Main task of Lexical Analyzer is to read the source file as a stream of characters which contains the code of your language and identify each lexeme and convert it into relevant tokens (A pair value which has a **Token Name** and **Attribute Value**). Following are the points which I have explained in this document:

- **How it works?**
- **How it identifies each lexeme?**
- **How it separates each lexeme token by token?**

STEP 01:

Language Selection: I have Selected **C++** Language to solve this problem.

CODE DESCRIPTION

CLASS T0KEN:

This class contains two field first is Lexeme Name and second is Token Name. It also contains a **toString()** function which return a string and some **Getter** and **Setter** functions related to the fields which I have discussed above.

CLASS BUFFER:

This class is for reading input from the Input file and store it on the string variable by concatenating each line it reads from the Input file. This class contains an array of type string which contains 20 basic keywords such as "int", "float", "double", "short", and "char" etc. It also contains lexeme begin and lexeme forward pointers. **Lexeme Begin** Pointer points the beginning position of each lexeme, and the **Lexeme Forward** Pointer points end of the lexeme when whole lexeme is read. There are also some functions which are:

- **InitializeBuffer ():** This function is used for reading input from text file line by line and storing it into a string variable.
- **getChar ():** This function is used for getting input character by character from the string variable which store the input file.
- **TokenFail ():** This function is used when character is not matched to one machine and it makes Forward pointer equal to Begin pointer and return first state of the next state number of next machine.
- **Retract ():** Used to decrement value of Forward pointer when forward read the extra character which is not the part of pattern.
- **TokenSuccess ():** When the token is successfully created then there is need to read the next lexeme, so this function shifts the begin pointer to place where Forward pointer is pointing.
- **CheckKeyword(string String):** Used to check whether lexeme is an identifier or keyword by traversing through the Keywords array.
- **Stop ():** Used to finish the process when whole input is tokenized.

CLASS TGS:

This class is the main class which contains the codes for all the machines depending on the types of patterns which are checked by these machines. This class also has the following functions:

- **RelationalOp(State):** Contains the code for Relational **(**< , > , <> , = , <= , >= **)** Operators Machine.
- **IdAndKeyword(State):** Contains the code for Identifiers and Keywords Machine.
- **Digits(State):** Contains the code for Digits Machine.
- **ArithematicOp(State):** Contains the code for Arithmetic **(**+ , - , / , * , %**)** Operators Machine.
- **LogicOp(State):** Contains the code for Logical **(**&& , || , !**)** Operators Machine.
- **Delimiters(State):** Contains the code for Delimiters **(**tabs , spaces , new line**)** Machine.
- **Puntuaters(State):** Contains the code for Punctuator **(,** , **;** , **:)** Machine.
- **Brackets(State):** Contains the code for Brackets**(**(,) , [,] , { , }**)** Machine.

- **NotInLanguage(State):** Contains the code for character not exists in the language.
- **DisplaySymbolTable():** Used to display the symbol table of the tokens.

CLASS NODE:

I have written this class to implement the Symbol Table. I have implemented the symbol table as a linked list of tokens. It has two fields which are **Token** and **NextToken** pointer. It also has the **Getter** and **Setter** methods for these fields.

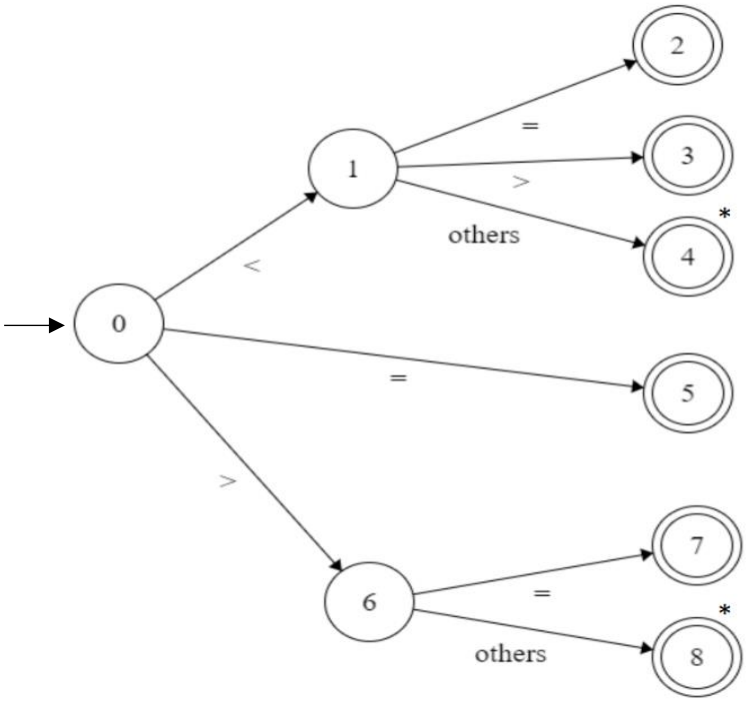
CLASS SYMBOLTABLE:

This class is used to create a Linked List of the Lexemes which have been identified as the tokens. There are the following fields in this class which are **HeadNode ,CurrentNode** and **TraverseNode** which points the linked list at the following Nodes and a **Size** variable to store the size of the Linked List. It also contains the following functions which are as follows:

- **Size():** This function is used to return the size of the Linked List.
- **Empty():** This function is used to check whether the Linked List is empty.
- **Add():** This function is add to a new token in the Lined List.
- **Display():** This function is used to display the Linked List.
- **Get(Index):** This function is used to get a token from the Linked List.
- **StoreInFile(FileName):** This function is used to store the Linked List in an output file.

PATTERN MACHINES

RELATIONAL OPERATOR MACHINE



RLEVENT C++ CODE

```
int RelationalOp(int State)
                {
                        this->Loop = true;
                        this->State = State;
                        int Temp;
                        char Char;

                        while(this->Loop)
                        {
                                switch(this->State)
                                {
                                        case -1:
                                                if(this->Buffer.Stop())
                                                {
                                                        return this->State = 0;
```

```
                }
                else
                {
                        this->Loop = false;
                }
        break;

        case 0:
                this->Buffer.setTGCounter(State);
                Char = this->Buffer.getChar();
                if(Char == '<')this->State=1;
                else if(Char == '=')this->State=5;
                else if(Char == '>')this->State=6;
                else return this->State=this->Buffer.TokenFail();
        break;

        case 1:
                Char = this->Buffer.getChar();
                if(Char == '=')this->State=2;
                else if(Char == '>')this->State=3;
                else this->State=4;
        break;

        case 2:
                RetToken.setLexemeName("RelOp");
                RetToken.setTokenName("LE");
                SymbolTable.Add(RetToken);
                this->Buffer.TokenSuccess();
                this->State=-1;
        break;
```

```
case 3:
        RetToken.setLexemeName("RelOp");
        RetToken.setTokenName("NE");
        SymbolTable.Add(RetToken);
        this->Buffer.TokenSuccess();
        this->State=-1;
break;

case 4:
        this->Buffer.Retract();
        RetToken.setLexemeName("RelOp");
        RetToken.setTokenName("LT");
        SymbolTable.Add(RetToken);
        this->Buffer.TokenSuccess();
        this->State=-1;
break;

case 5:
        RetToken.setLexemeName("RelOp");
        RetToken.setTokenName("EQ");
        SymbolTable.Add(RetToken);
        this->Buffer.TokenSuccess();
        this->State=-1;
break;

case 6:
        Char = this->Buffer.getChar();
        if(Char == '=')this->State=7;
        else this->State=8;
break;
```

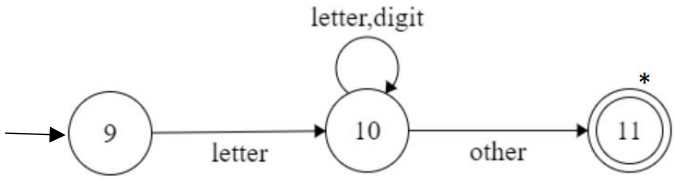
```
                    case 7:
                        RetToken.setLexemeName("RelOp");
                        RetToken.setTokenName("GE");
                        SymbolTable.Add(RetToken);
                        this->Buffer.TokenSuccess();
                        this->State=-1;
                    break;

                    case 8:
                        this->Buffer.Retract();
                        RetToken.setLexemeName("RelOp");
                        RetToken.setTokenName("GT");
                        SymbolTable.Add(RetToken);
                        this->Buffer.TokenSuccess();
                        this->State=-1;
                    break;
                }
            }
            return -1;
    }
```

Explanation of Relational Operator Machine:

Each case in switch is the state of machine. In this machine state 0 is starting state of machine. If control is at 0 state and read '<' operator it will move on state 1 and at state 1 if it read '=' then it will move on state 2 and return the name and value of token (RelOP,LE). If it reads '>' at state 1 then it will move on state 3 and return the token name and value (RelOP,NE). If it reads any other character, then it will move on state 4 and retract and return the name and value (RelOP,LT). If it reads '=' at state 0 then it will return name and value (RelOP,EQ). If it reads '>' at state 0 it will move on state 6 and at state 6 if it reads '=' then it will move on state 7 and it will return name and value of token (RelOP,GE). If it reads any other character,

then it will move on state 8 and retract and return the name and value of token (RelOP,GT). If no operator in lexeme it will move on other automata.

IDENTIFIERS MACHINE



RELEVENT C++ CODE

```
int IdKeyword(int State)
                {
                        this->Loop = true;
                        this->State = State;
                        int Temp;
                        char Char;

                        while(this->Loop)
                        {
                                switch(this->State)
                                {
                                        case -1:
                                                if(this->Buffer.Stop())
                                                {
                                                        return this->State = 0;
                                                }
                                                else
                                                {
                                                        this->Loop = false;
                                                }
```

```
                                        break;

                                        case 9:
                                                this->String = "";
                                                Char = this->Buffer.getChar();
                                                Temp = int(Char);
                                                if((Temp>=65 && Temp<=90) ||  (Temp>=97 &&
Temp<=122))
                                                {
                                                        this->String = Char;
                                                        this->State=10;
                                                }
                                                else return this->State=this->Buffer.TokenFail();
                                        break;

                                        case 10:
                                                Char = this->Buffer.getChar();
                                                Temp = int(Char);
                                                if((Temp>=65 && Temp<=90) ||  (Temp>=97 && Temp<=122)
|| (Temp>=48 && Temp<=57))
                                                {
                                                        this->String += Char;
                                                        this->State=10;
                                                }
                                                else this->State = 11;
                                        break;

                                        case 11:
                                                if(this->Buffer.CheckKeyword(this->String) == true)
                                                {
                                                        this->State = 12;
```

```
                }
                else
                {
                        this->Buffer.Retract();
                        RetToken.setLexemeName("Identifier");
                        RetToken.setTokenName(this->String);
                        SymbolTable.Add(RetToken);
                        this->Buffer.TokenSuccess();
                        this->State = -1;
                }
                break;

        case 12:
                this->Buffer.Retract();
                RetToken.setLexemeName("Keyword");
                RetToken.setTokenName(this->String);
                SymbolTable.Add(RetToken);
                this->Buffer.TokenSuccess();
                this->State = -1;
        break;
            }
        }
        return -1;
}
```

Explanation of Identifiers Machine:

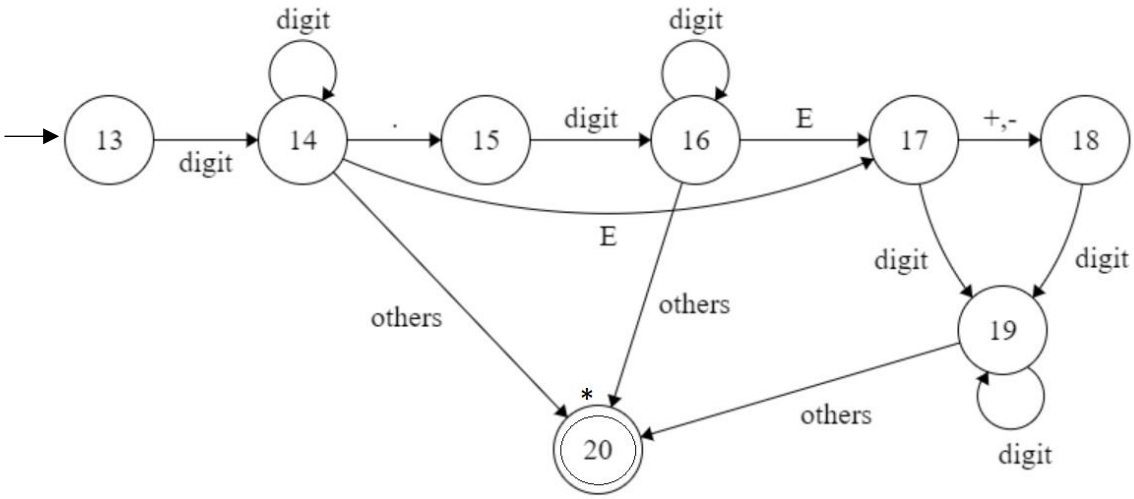
Starting state of Identifier Machine is state 9.

➔ If it reads any letter ((a-z) or (A-Z)) it will move on state 10.

➔ At state 10 there is loop of letter and digit ((letter, digit)*). If it reads any letter or digit it will remain on state 10 until other character is read.

➔ If it reads any other character at state 10 then it will move on state 11 and then it will check whether it is keyword or not if yes then it will move on state 12 and return the name and value of token (KeyWord, val) if not then it will return the name and value (Id,val). If at starting state there is no letter then it will move on next automata.

NUMBERS MACHINE



RELEVENT C++ CODE

```
int Digits(int State)
                {
                        this->Loop = true;
                        this->State = State;
                        int Temp;
                        char Char;

                        while(this->Loop)
                        {
                                switch(this->State)
                                {
                                        case -1:
                                                if(this->Buffer.Stop())
                                                {
                                                        return this->State = 0;
                                                }
                                                else
                                                {
                                                        this->Loop = false;
```

```
                }
            break;

            case 13:
                this->String = "";
                Char = this->Buffer.getChar();
                this->String = Char;
                Temp = int(Char);
                if(Temp>=48 && Temp<=57)this->State = 14;
                else if(Char == '.')this->State = 15;
                else return this->State  = this->Buffer.TokenFail();
            break;

            case 14:
                Char = this->Buffer.getChar();
                Temp = int(Char);
                if(Temp>=48 && Temp<=57){this->String +=
Char;State = 14;}
                else if(Char == '.'){this->String += Char;this->State =
15;}
                else if(Char == 'E'){this->String += Char;this->State =
17;}
                else this->State  = 20;
            break;

            case 15:
                Char = this->Buffer.getChar();
                Temp = int(Char);
                if(Temp>=48 && Temp<=57){this->String +=
Char;this->State = 16;}
                //Missing Case Require
            break;

            case 16:
                Char = this->Buffer.getChar();
                Temp = int(Char);
                if(Temp>=48 && Temp<=57){this->String +=
Char;this->State = 16;}
                else if(Char == 'E'){this->String += Char;this->State =
17;}
                else this->State = 20;
            break;

            case 17:
                Char = this->Buffer.getChar();
                Temp = int(Char);
                if(Char == '-' || Char == '+'){this->String += Char;this-
>State = 18;}
```

```
                                        else if(Temp>=48 && Temp<=57){this->String +=
Char;this->State = 19;}
                                        //Missing Case Require
                                    break;

                                    case 18:
                                        Char = this->Buffer.getChar();
                                        Temp = int(Char);
                                        if(Temp>=48 && Temp<=57){this->String +=
Char;this->State = 19;}
                                        //Missing Case Require
                                    break;

                                    case 19:
                                        Char = this->Buffer.getChar();
                                        Temp = int(Char);
                                        if(Temp>=48 && Temp<=57){this->String +=
Char;this->State = 19;}
                                        else this->State = 20;
                                    break;

                                    case 20:
                                        this->Buffer.Retract();
                                        RetToken.setLexemeName("Number");
                                        RetToken.setTokenName(this->String);
                                        SymbolTable.Add(RetToken);
                                        this->Buffer.TokenSuccess();
                                        this->State = -1;
                                    break;
                                }
                            }
                            return -1;
                        }
```

Explanation of Digits Machine:

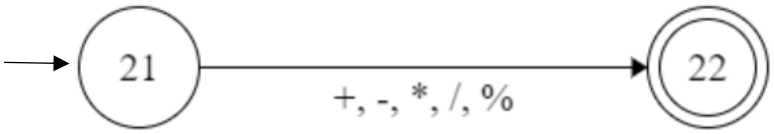
Starting State of Digits machine is state 13.

- ➔ If it reads digit (0-9) at state 13 it will move on state 14.
- ➔ At state 14 when it reads digit it will remain on 14 if reads '.' (dot) it will move on state 15 if it reads E it will move on state 17 other then digit or '.' or E It will move on state 20 and retract and return the name and value of token (Number, val).
- ➔ At state 15 if it reads digit it will move on state 16.
- ➔ At state 16 if it reads digit it will remain at state 16. If it reads E it will move in state 17. If any other character, it will move on state 20. It retracts and return name and value of token (Number,val).
- ➔ At state 17 if it reads '+' or '–' it will move on state 18 and if digit it will move on state 19.

➔ At state 18 if it reads digit it will move on 19.

➔ At state 19 when it reads digit it will remain at 19 otherwise it will move on state 20 and then retract and return name and value of token (Number, val).

ARITHEMATIC OPERATORS MACHINE



RELEVENT C++ CODE

```
int ArithmaticOp(int State)
                {
                        this->Loop = true;
                        this->State = State;
                        int Temp;
                        char Char;

                        while(this->Loop)
                        {
                                switch(this->State)
                                {
                                        case -1:
                                                if(this->Buffer.Stop())
                                                {
                                                        return this->State = 0;
                                                }
                                                else
                                                {
                                                        this->Loop = false;
                                                }
                                        break;

                                        case 21:
                                                this->String = "";
                                                Char = this->Buffer.getChar();
                                                if(Char == '+' || Char == '-' || Char == '*' || Char ==
'/' || Char == '%')
                                                {
                                                        this->String = Char;
                                                        this->State = 22;
```

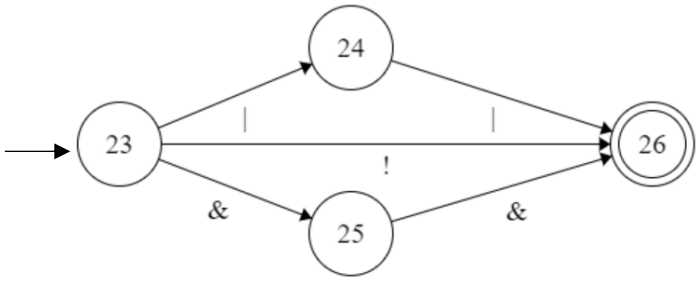
```
                    }
                    else return this->State = this->Buffer.TokenFail();
                break;

                case 22:
                    RetToken.setLexemeName("ArithOp");
                    RetToken.setTokenName(this->String);
                    SymbolTable.Add(RetToken);
                    this->Buffer.TokenSuccess();
                    this->State = -1;
                break;
            }
        }
        return -1;
}
```

Explanation of Arithmetic Operator Machine:

Starting state of this machine is state 21.

➔ If it reads any arithmetic operator (+ , - , * , / , %) it will move on state 22 and return the name and value of token (ArthOp,val). If it reads any other character it will move on next automata.

LOGICAL OPERATOR MACHINE



RELEVENT C++ CODE

```
int LogicalOp(int State)
            {
                this->Loop = true;
                this->State = State;
                int Temp;
                char Char;
```

```
while(this->Loop)
{
	switch(this->State)
	{
		case -1:
			if(this->Buffer.Stop())
			{
				return this->State = 0;
			}
			else
			{
				this->Loop = false;
			}
		break;

		case 23:
			this->String = "";
			Char = this->Buffer.getChar();
			if(Char == '&' || Char == '|'){this->String = Char;this->State =
24;}
			else if(Char == '!'){this->String = Char;this->State = 25;}
			else return this->State = this->Buffer.TokenFail();
		break;

		case 24:
			Char = this->Buffer.getChar();
			if(Char == '&' && !(this->String.compare("&"))){this->String +=
Char;this->State = 25;}
			else if(Char == '|' && !(this->String.compare("|"))){this-
>String += Char;this->State = 25;}
```

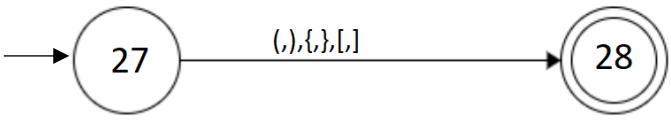
```
                                        else {this->Buffer.Retract();this->Buffer.Retract();return this->State = 33;}
                                    break;

                                    case 25:
                                        RetToken.setLexemeName("LogicOp");
                                        RetToken.setTokenName(this->String);
                                        SymbolTable.Add(RetToken);
                                        this->Buffer.TokenSuccess();
                                        this->State = -1;
                                    break;
                                }
                            }
                            return -1;
                        }
```

Explanation of Logical Operator Machine:

Starting state of Logical Operator is state 23.

- ➔ If it reads OR operator at state 23 it will move on state 24, if it reads AND operator it will move on state 25, and if it reads NOT operator it will move on final state 26 and return the name and value of Token.
- ➔ At state 24 when it reads OR operator it will move on final state 26 and return the name and value of Token. If any other, then it will show that previous OR is not part of language.
- ➔ At state 25 when it reads AND operator it will move on final state 26 and return the name and value of Token. If any other, then it will show that previous AND is not part of language.

BRACKETS MACHINE



RELEVENT C++ CODE

```
int Brackets(int State)
		{
			this->Loop = true;
			this->State = State;
			int Temp;
			char Char;

			while(this->Loop)
			{
				switch(this->State)
				{
					case -1:
						if(this->Buffer.Stop())
						{
							return this->State = 0;
						}
						else
						{
							this->Loop = false;
						}
					break;

					case 26:
						this->String = "";
						Char = this->Buffer.getChar();
						if(Char == '(' || Char == ')' || Char == '{' || Char == '}' || Char
== '[' || Char == ']')
						{
							this->String = Char;
							this->State = 27;
						}
```

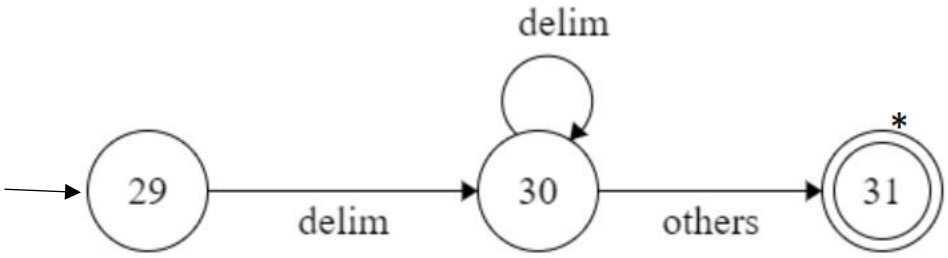
```
                else return this->State = this->Buffer.TokenFail();
            break;

            case 27:
                RetToken.setLexemeName("BracKet");
                RetToken.setTokenName(this->String);
                SymbolTable.Add(RetToken);
                this->Buffer.TokenSuccess();
                this->State = -1;
            break;
        }
    }
    return -1;
}
```

Explanation of Brackets Machine:

Starting state of this machine is state 27.

➔ If it reads any bracket ((,),{,},[,]) it will move on state 28 and return the name and value of token (bracket,val). If it reads any other character it will move on next automata.

DELIMITERS MACHINE



RELEVENT C++ CODE

```
int Delimiters(int State)
```

```
{
        this->Loop = true;
        this->State = State;
        int Temp;
        char Char;

        while(this->Loop)
        {
            switch(this->State)
            {
                case -1:
                    if(this->Buffer.Stop())
                    {
                        return this->State = 0;
                    }
                    else
                    {
                        this->Loop = false;
                    }
                break;

                case 28:
                    this->String = "";
                    Char = this->Buffer.getChar();
                    Temp = int(Char);
                    if(Temp == 9){this->String = "VT,";this->State = 29;}
                    else if(Temp == 10){this->String += "LF,";this->State = 29;}
                    else if(Temp == 11){this->String += "HT,";this->State = 29;}
                    else if(Temp == 32){this->String += "SPC,";this->State = 29;}
                    else return this->State = this->Buffer.TokenFail();
                break;
```

```
                case 29:
                        Char = this->Buffer.getChar();
                        Temp = int(Char);
                        if(Temp == 9){this->String += "VT,";this->State = 29;}
                        else if(Temp == 10){this->String += "LF,";this->State = 29;}
                        else if(Temp == 11){this->String += "HT,";this->State = 29;}
                        else if(Temp == 32){this->String += "SPC,";this->State = 29;}
                        else this->State = 30;
                break;

                case 30:
                        this->Buffer.Retract();
                        RetToken.setLexemeName("Delimiter");
                        RetToken.setTokenName(this->String);
                        SymbolTable.Add(RetToken);
                        this->Buffer.TokenSuccess();
                        this->State = -1;
                break;
            }
        }
        return -1;
}
```

Explanation of Delimiters Machine:

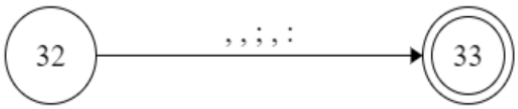
Starting state of Delimiters Machine is state 29.

➔ If it reads any delimiter (space, tab, line feed …) it will move on state 30.

➔ At state 30 there is loop of delimiter ((delimiter)*). If it reads any delimiter it will remain on state 30 until other character is read.

➔ If it reads any other character at state 30 then it will move on state 31 and return the name and value of token (Delim, val). If at starting state there is no delimiter then it will move on next automata.

PUNTUATORS MACHINE



RELEVENT C++ CODE

```
int Puntuaters(int State)
		{
			this->Loop = true;
			this->State = State;
			int Temp;
			char Char;

			while(this->Loop)
			{
				switch(this->State)
				{
					case -1:
						if(this->Buffer.Stop())
						{
							return this->State = 0;
						}
						else
						{
							this->Loop = false;
						}
					break;

					case 31:
						this->String = "";
						Char = this->Buffer.getChar();
						if(Char == ';' || Char == ',' || Char == ':'){this->String =
Char;this->State = 32;}
						else return this->State = this->Buffer.TokenFail();
					break;

					case 32:
```

```
                                        RetToken.setLexemeName("Puntuator");
                                        RetToken.setTokenName(this->String);
                                        SymbolTable.Add(RetToken);
                                        this->Buffer.TokenSuccess();
                                        this->State = -1;
                                break;
                        }
                }
                return -1;
        }
```

Explanation of Punctuators Machine:

Starting state of this machine is state 32.

➔ If it reads any punctuator (, , ; , :) it will move on state 33 and return the name and value of token (punctuator, val). If it reads any other character it will move on next automata.

NOT IN LANGUAGE

```
int NotInLanguage(int State)
        {
                this->Loop = true;
                this->State = State;
                int Temp;
                char Char;

                while(this->Loop)
                {
                        switch(this->State)
                        {
                                case -1:
                                        if(this->Buffer.Stop())
                                        {
                                                return this->State = 0;
                                        }
```

```
                    else
                    {
                        this->Loop = false;
                    }
                break;

                case 33:
                    this->String = "";
                    Char = this->Buffer.getChar();
                    this->String = Char;
                    RetToken.setLexemeName("NotInLang");
                    RetToken.setTokenName(this->String);
                    SymbolTable.Add(RetToken);
                    this->Buffer.TokenSuccess();
                    this->State = -1;
                break;
            }
        }
        return -1;
}
```

Case 33: This is the case in which all those characters that are not the part of language are identified and returned.