using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Text.RegularExpressions;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.Collections;

namespace LexicalAnalyzerV1

{

public partial class Form1 : Form

{

List<List<String>> Symboltable = new List<List<String>>();

ArrayList LineNumber;

ArrayList Variables;

ArrayList KeyWords;

ArrayList Constants;

ArrayList finalArray;

ArrayList tempArray;

Regex variable\_Reg;

Regex constants\_Reg;

Regex operators\_Reg;

int lexemes\_per\_line;

int ST\_index;

public Form1()

{

InitializeComponent();

String[] k\_ = { "int", "float", "begin", "end", "print", "if",

"else" };

ArrayList key = new ArrayList(k\_);

LineNumber = new ArrayList();

Variables = new ArrayList();

KeyWords = new ArrayList();

Constants = new ArrayList();

finalArray = new ArrayList();

tempArray = new ArrayList();

variable\_Reg = new Regex(@"^[A-Za-z|\_][A-Za-z|0-9]\*$");

constants\_Reg = new Regex(@"^[0-9]+([.][0

9]+)?([e]([+|-])?[0-9]+)?$");

operators\_Reg = new Regex(@"[+-/\*=;>(){}]");

int L = 1;

Output.Text = "";

ST.Text = "";

Symboltable.Clear();

if\_deleted = false;

string strinput = Input.Text;

char[] charinput = strinput.ToCharArray();

}

private void btn\_Input\_Click(object sender, EventArgs e)

{

//taking user input from rich textbox

String userInput = tfInput.Text;

//List of keywords which will be used to seperate

keywords from variables

List<String> keywordList = new List<String>();

keywordList.Add("int");

keywordList.Add("float");

keywordList.Add("while");

keywordList.Add("main");

keywordList.Add("if");

keywordList.Add("else");

keywordList.Add("new");

//row is an index counter for symbol table

int row = 1;

//count is a variable to incremenet variable id in

tokens

int count = 1;

//line\_num is a counter for lines in user input

int line\_num = 0;

//SymbolTable is a 2D array that has the following

structure

//[Index][Variable Name][type][value][line#]

//rows are incremented with each variable information

entry

String[,] SymbolTable = new String[20, 6];

List<String> varListinSymbolTable = new List<String>();

//Input Buffering

ArrayList finalArray = new ArrayList();

ArrayList finalArrayc = new ArrayList();

ArrayList tempArray = new ArrayList();

char[] charinput = userInput.ToCharArray();

//Regular Expression for Variables

Regex variable\_Reg = new Regex(@"^[A-Za-z|\_][A-Za-z|0

9]\*$");

//Regular Expression for Constants

Regex constants\_Reg = new Regex(@"^[0-9]+([.][0

9]+)?([e]([+|-])?[0-9]+)?$");

//Regular Expression for Operators

Regex operators\_Reg = new Regex(@"^[-\*+/><&&||=]$");

//Regular Expression for Special\_Characters

Regex Special\_Reg = new Regex(@"^[.,'\[\]{}();:?]$");

for (int itr = 0; itr < charinput.Length; itr++)

{

Match Match\_Variable =

variable\_Reg.Match(charinput[itr] + "");

Match Match\_Constant =

constants\_Reg.Match(charinput[itr] + "");

Match Match\_Operator =

operators\_Reg.Match(charinput[itr] + "");

Match Match\_Special =

Special\_Reg.Match(charinput[itr] + "");

if (Match\_Variable.Success ||

Match\_Constant.Success || Match\_Operator.Success ||

Match\_Special.Success || charinput[itr].Equals(' '))

{

tempArray.Add(charinput[itr]);

}

if (charinput[itr].Equals('\n'))

{

if (tempArray.Count != 0)

{

int j = 0;

String fin = "";

for (; j < tempArray.Count; j++)

{

fin += tempArray[j];

}

finalArray.Add(fin);

tempArray.Clear();

}

}

}

if (tempArray.Count != 0)

{

int j = 0;

String fin = "";

for (; j < tempArray.Count; j++)

{

fin += tempArray[j];

}

finalArray.Add(fin);

tempArray.Clear();

}

// Final Array SO far correct

tfTokens.Clear();

symbolTable.Clear();

//looping on all lines in user input

for (int i = 0; i < finalArray.Count; i++)

{

String line = finalArray[i].ToString();

//tfTokens.AppendText(line + "\n");

char[] lineChar = line.ToCharArray();

line\_num++;

//taking current line and splitting it into lexemes

by space

for (int itr = 0; itr < lineChar.Length; itr++)

{

Match Match\_Variable =

variable\_Reg.Match(lineChar[itr] + "");

Match Match\_Constant =

constants\_Reg.Match(lineChar[itr] + "");

Match Match\_Operator =

operators\_Reg.Match(lineChar[itr] + "");

Match Match\_Special =

Special\_Reg.Match(lineChar[itr] + "");

if (Match\_Variable.Success ||

Match\_Constant.Success)

{

tempArray.Add(lineChar[itr]);

}

if (lineChar[itr].Equals(' '))

{

if (tempArray.Count != 0)

{

int j = 0;

String fin = "";

for (; j < tempArray.Count; j++)

{

fin += tempArray[j];

}

finalArrayc.Add(fin);

tempArray.Clear();

}

}

if (Match\_Operator.Success ||

Match\_Special.Success)

{

if (tempArray.Count != 0)

{

int j = 0;

String fin = "";

for (; j < tempArray.Count; j++)

{

fin += tempArray[j];

}

finalArrayc.Add(fin);

tempArray.Clear();

}

finalArrayc.Add(lineChar[itr]);

}

}

if (tempArray.Count != 0)

{

String fina = "";

for (int k = 0; k < tempArray.Count; k++)

{

fina += tempArray[k];

}

finalArrayc.Add(fina);

tempArray.Clear();

}

// we have asplitted line here

for (int x = 0; x < finalArrayc.Count; x++)

{

Match operators =

operators\_Reg.Match(finalArrayc[x].ToString());

Match variables =

variable\_Reg.Match(finalArrayc[x].ToString());

Match digits =

constants\_Reg.Match(finalArrayc[x].ToString());

Match punctuations =

Special\_Reg.Match(finalArrayc[x].ToString());

if (operators.Success)

{

// if a current lexeme is an operator

then make a token e.g. < op, = >

tfTokens.AppendText("< op, " +

finalArrayc[x].ToString() + "> ");

}

else if (digits.Success)

{

// if a current lexeme is a digit then

make a token e.g. < digit, 12.33 >

tfTokens.AppendText("< digit, " +

finalArrayc[x].ToString() + "> ");

}

else if (punctuations.Success)

{

// if a current lexeme is a punctuation

then make a token e.g. < punc, ; >

tfTokens.AppendText("< punc, " +

finalArrayc[x].ToString() + "> ");

}

else if (variables.Success)

{

// if a current lexeme is a variable

and not a keyword

if

(!keywordList.Contains(finalArrayc[x].ToString())) // if it is not

a keyword

{

// check what is the category of

varaible, handling only two cases here

//Category1- Variable

initialization of type digit e.g. int count = 10 ;

//Category2- Variable

initialization of type String e.g. String var = ' Hello ' ;

Regex reg1 = new

Regex(@"^(int|float|double)\s([A-Za-z|\_][A-Za-z|0

9]{0,10})\s(=)\s([0-9]+([.][0-9]+)?([e][+|-]?[0-9]+)?)\s(;)$"); //

line of type int alpha = 2 ;

Match category1 = reg1.Match(line);

Regex reg2 = new

Regex(@"^(String|char)\s([A-Za-z|\_][A-Za-z|0

9]{0,10})\s(=)\s[']\s([A-Za-z|\_][A-Za-z|0-9]{0,30})\s[']\s(;)$");

// line of type String alpha = ' Hello ' ;

Match category2 = reg2.Match(line);

//if it is a category 1 then add a

row in symbol table containing the information related to that

variable

if (category1.Success)

{

SymbolTable[row, 1] =

row.ToString(); //index

SymbolTable[row, 2] =

finalArrayc[x].ToString(); //variable name

SymbolTable[row, 3] =

finalArrayc[x - 1].ToString(); //type

SymbolTable[row, 4] =

finalArrayc[x+2].ToString(); //value

SymbolTable[row, 5] =

line\_num.ToString(); // line number

tfTokens.AppendText("<var" +

count + ", " + row + "> ");

symbolTable.AppendText(SymbolTable[row, 1].ToString() + " \t ");

symbolTable.AppendText(SymbolTable[row, 2].ToString() + " \t ");

symbolTable.AppendText(SymbolTable[row, 3].ToString() + " \t ");

symbolTable.AppendText(SymbolTable[row, 4].ToString() + " \t ");

symbolTable.AppendText(SymbolTable[row, 5].ToString() + " \n ");

row++;

count++;

}

//if it is a category 2 then add a

row in symbol table containing the information related to that

variable

else if (category2.Success)

{

// if a line such as String

var = ' Hello ' ; comes and the loop moves to index of array

containing Hello ,

//then this if condition

prevents addition of Hello in symbol Table because it is not a

variable it is just a string

if (!(finalArrayc[x

1].ToString().Equals("'") &&

finalArrayc[x+1].ToString().Equals("'")))

{

SymbolTable[row, 1] =

row.ToString(); // index

SymbolTable[row, 2] =

finalArrayc[x].ToString(); //varname

SymbolTable[row, 3] =

finalArrayc[x-1].ToString(); //type

SymbolTable[row, 4] =

finalArrayc[x+3].ToString(); //value

SymbolTable[row, 5] =

line\_num.ToString(); // line number

tfTokens.AppendText("<var"

+ count + ", " + row + "> ");

symbolTable.AppendText(SymbolTable[row, 1].ToString() + " \t ");

symbolTable.AppendText(SymbolTable[row, 2].ToString() + " \t ");

symbolTable.AppendText(SymbolTable[row, 3].ToString() + " \t ");

symbolTable.AppendText(SymbolTable[row, 4].ToString() + " \t ");

symbolTable.AppendText(SymbolTable[row, 5].ToString() + " \n ");

row++;

count++;

}

else

{

tfTokens.AppendText("<String" + count +

", " + finalArrayc[x].ToString() + "> ");

}

}

else

{

// if any other category line comes in we check if we have

initializes that varaible before,

// if we have initiazed it before then we put the index of that

variable in symbol table, in its token

String ind = "Default";

String ty = "Default";

String val = "Default";

String lin = "Default";

for (int r = 1; r <= SymbolTable.GetLength(0); r++)

{

//search in the symbol table if variable entry already exists

if (SymbolTable[r,

2].Equals(finalArrayc[x].ToString()))

{

ind = SymbolTable[r,

1];

ty = SymbolTable[r, 3];

val = SymbolTable[r,

4];

lin = SymbolTable[r,

5];

tfTokens.AppendText("<var" + ind + ", " + ind + "> ");

break;

}

}

}

}

// if a current lexeme is not a variable but a keyword then make a

token such as: <keyword, int>

else

{

tfTokens.AppendText("<keyword, " +

finalArrayc[x].ToString() + "> ");

}

}

}

tfTokens.AppendText("\n");

finalArrayc.Clear();

}

}

}

}

#region Display Symbol Table

for (int j = 0; j < Symboltable.Count; j++)

{

for (int z = 0; z < Symboltable[j].Count; z++)

{ ST.AppendText(Symboltable[j][z] + "\t"); }

ST.AppendText("\n");

}

#endregion

}

#region Make Entry Symbol Table

void Check\_And\_Make\_Entries()

{

KeyWords.Remove("begin"); KeyWords.Remove("end");

KeyWords.Remove("print");

KeyWords.Remove("if"); KeyWords.Remove("else");

if (lexemes\_per\_line - 4 == 0 || lexemes\_per\_line - 7

== 0)

{

if (lexemes\_per\_line == 7)

{

Variables.RemoveAt(Variables.Count - 1);

Variables.RemoveAt(Variables.Count - 1);

}

for (; ST\_index < KeyWords.Count; ST\_index++)

{

Symboltable.Add(new List<string>());

Symboltable[ST\_index].Add(ST\_index + 1 + "");

Symboltable[ST\_index].Add(Variables[ST\_index] +

"");

Symboltable[ST\_index].Add(KeyWords[ST\_index] +

"");

Symboltable[ST\_index].Add(Constants[ST\_index] +

"");

Symboltable[ST\_index].Add(LineNumber[ST\_index]

+ "");

}

}

if (lexemes\_per\_line - 6 == 0)

{

Variables.RemoveAt(Variables.Count - 1);

Variables.RemoveAt(Variables.Count - 1);

Variables.RemoveAt(Variables.Count - 1);

}

}

#endregion

**out put**

