**COMSATS UNIVERSITY ISLAMABAD**

**COMPILER CONSTRUCTION**

**Lab Terminal**

**Aliza Tanweer (SP20-BCS-013)**

**Kulsoom Khurshid (SP20-BCS-044)**

**Main Form1.cs**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.Text.RegularExpressions;

namespace SLR\_parser {

public partial class Form1 : Form {

public List<List<String>> Table = new List<List<string>>();

public List<List<string>> tokens = new List<List<string>>();

public SLR\_DFA dfa;

public string test;

public Form1() {

InitializeComponent();

}

private void button1\_Click(object sender, EventArgs e) {

String grammar = InputBox.Text;

Preprocessor preprocessor1 = new Preprocessor();

Tuple<int, String> output = preprocessor1.InitGrammar(grammar);

preprocessor1.Find\_FirstSet();

preprocessor1.Find\_FollowSet();

//Console.WriteLine(preprocessor1.FOLLOW\_SET);

// Display first and follow in GUI

foreach (var item in preprocessor1.FIRST\_SET) {

String val = item.Key + " : { " + String.Join(", ", item.Value) + " } \n\n";

FirstSetBox.Text = FirstSetBox.Text + val;

}

foreach (var item in preprocessor1.FOLLOW\_SET) {

String val = item.Key + " : { " + String.Join(", ", item.Value) + " } \n\n";

FollowSetBox.Text = FollowSetBox.Text + val;

}

dfa = new SLR\_DFA(preprocessor1.Rules, preprocessor1.nonterminals, preprocessor1.Start\_symbol);

dfa.augmentGrammar(preprocessor1.Rules, preprocessor1.nonterminals, preprocessor1.Start\_symbol);

dfa.statesDict[0] = new List<List<List<String>>>(dfa.findClosure(dfa.AugRules, dfa.AugRules[0][0][0]));

dfa.generateStates(dfa.statesDict);

Table = dfa.createParseTable(dfa.statesDict, dfa.stateMap, preprocessor1.terminals, preprocessor1.nonterminals, preprocessor1.FOLLOW\_SET);

//// Display GOTO in GUI

foreach (var item in dfa.stateMap) {

String val = "GOTO: ( I" + item.Key.Item1 + " , " + item.Key.Item2 + " ) = I" + item.Value + "\n\n";

GotoBox.Text = GotoBox.Text + val;

}

// Display DFA

foreach (var st in dfa.statesDict) {

//Console.WriteLine("STATE : "+st.Key);

DFABox.Text = DFABox.Text + "STATE I" + st.Key + ":\n";

foreach (var item in dfa.statesDict[st.Key]) {

//Console.WriteLine(" "+ String.Join(" ", item[0]) + " -> "+String.Join(" ",item[1]));

if (!String.Join(" ", item[1]).Contains(".")) {

DFABox.Text = DFABox.Text + " " + String.Join(" ", item[0]) + " -> " + String.Join(" ", item[1]) + " .\n";

} else {

DFABox.Text = DFABox.Text + " " + String.Join(" ", item[0]) + " -> " + String.Join(" ", item[1]) + "\n";

}

}

DFABox.Text = DFABox.Text + "\n-----------------------------\n";

}

Ptable.ColumnCount = dfa.colss.Count + 1;

Ptable.AutoResizeColumns();

Ptable.AutoSizeColumnsMode = DataGridViewAutoSizeColumnsMode.AllCells;

int i = 1;

Ptable.Columns[0].Name = "";

foreach (var x in dfa.colss) {

Ptable.Columns[i].Name = x;

i++;

}

int count = 0;

foreach (var x in Table) {

//Console.WriteLine(count + ": " + String.Join(" ", x));

x.Insert(0, "I" + count);

Ptable.Rows.Add(x.ToArray());

count++;

}

//Console.WriteLine(dfa.numbered\_rules);

foreach (var x in dfa.numbered\_rules) {

NumberedBox.Text = NumberedBox.Text + x.Key + ": " + String.Join(" ", x.Value[0]) + " -> " + String.Join(" ", x.Value[1]) + "\n";

}

}

private void ClearAll\_Click(object sender, EventArgs e) {

InputBox.Clear();

GotoBox.Clear();

DFABox.Clear();

FirstSetBox.Clear();

FollowSetBox.Clear();

Ptable.Columns.Clear();

Ptable.Rows.Clear();

Ptable.Refresh();

NumberedBox.Clear();

ParsingBox.Columns.Clear();

ParsingBox.Rows.Clear();

InputString.Clear();

tokenGridView.Rows.Clear();

}

private void Ptable\_CellContentClick(object sender, DataGridViewCellEventArgs e) {

}

private void ParseButton\_Click(object sender, EventArgs e) {

String grammar = InputBox.Text;

String input = InputString.Text;

Lexer lexer = new Lexer(input, grammar);

this.tokens = lexer.LexAnalysis();

this.renderTokens();

ParsingBox.Columns.Clear();

ParsingBox.Rows.Clear();

InputParsing parser = new InputParsing(Table, dfa.numbered\_rules);

List<List<String>> outputTrace = parser.parse(InputString.Text, dfa.colss);

ParsingBox.ColumnCount = 3;

ParsingBox.AutoResizeColumns();

ParsingBox.AutoSizeColumnsMode = DataGridViewAutoSizeColumnsMode.AllCells;

ParsingBox.Columns[0].Name = "STACK";

ParsingBox.Columns[1].Name = "INPUT";

ParsingBox.Columns[2].Name = "ACTION";

foreach (var x in outputTrace) {

ParsingBox.Rows.Add(x.ToArray());

}

// Generate Semantic rules

// Generate Annotated Parse Tree

SemanticAnalyzer SDT = new SemanticAnalyzer(InputString.Text, "");

SDD.Text = SDT.performSDD();

semanticText2.Text = SDT.generateAnnotatedPTree();

// Generate Syntax Tree

MainSyntaxTree MST = new MainSyntaxTree();

String expr = MST.InfixToPostfix(InputString.Text);

Console.WriteLine(expr.ToString());

String[] arrPostfix = expr.Split(' ');

//MST.PrintSyntaxTree(MST.GenerateSyntaxTree(arrPostfix));

String tree = MST.GenerateSyntaxTree(arrPostfix);

syntaxTreeBox.Text = tree;

//Generate 3 addr code

ThreeAddrCode IR = new ThreeAddrCode();

String code = IR.GenerateCode(MST.getRoot(arrPostfix));

Console.WriteLine(code);

IRbox.Text = code;

}

private void renderTokens() {

tokenGridView.Columns.Clear();

tokenGridView.Rows.Clear();

tokenGridView.ColumnCount = 2;

tokenGridView.AutoResizeColumns();

tokenGridView.AutoSizeColumnsMode = DataGridViewAutoSizeColumnsMode.AllCells;

tokenGridView.Columns[0].Name = "TYPE";

tokenGridView.Columns[1].Name = "VALUE";

foreach(var x in tokens)

{

tokenGridView.Rows.Add(x.ToArray());

}

}

private void tabPage3\_Click(object sender, EventArgs e) {

}

private void groupBox7\_Enter(object sender, EventArgs e) {

}

private void semanticText2\_TextChanged(object sender, EventArgs e)

{

}

private void NumberedBox\_TextChanged(object sender, EventArgs e)

{

}

private void Form1\_Load(object sender, EventArgs e)

{

}

}

}

**Preprocess.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Text.RegularExpressions;

namespace SLR\_parser {

class Preprocessor {

public IDictionary<String, List<List<String>>> Rules = new Dictionary<String, List<List<String>>>();

private Regex regex = new Regex(@"^[a-z.(),\[\]0-9$#\*;+-/]$");

private Regex regexNonTerm = new Regex(@"^[A-Z]+$");

public List<string> nonterminals = new List<string>();

public List<string> terminals = new List<string>();

public String Start\_symbol;

public IDictionary<String, List<String>> FIRST\_SET = new Dictionary<String, List<String>>();

public IDictionary<String, List<String>> FOLLOW\_SET = new Dictionary<String, List<String>>();

public Tuple<int, String> InitGrammar(string grammar) {

List<string> g0 = grammar.Split('\n').ToList();

List<string> g1 = new List<string>();

Start\_symbol = Regex.Split(g0[0], "->")[0];

Start\_symbol = Start\_symbol.Split(' ')[0];

nonterminals.Add(Start\_symbol+"'");

nonterminals.Add(Start\_symbol);

// Adding Augment Rule

List<String> a1 = new List<string>();

List<List<String>> b1 = new List<List<string>>();

String newStart = Start\_symbol.Replace("'","");

a1.Add(newStart);

b1.Add(a1);

Rules[Start\_symbol+"'"] = b1;

// Making | an new rule

foreach (String a in g0) {

if (a.Contains(" | ")) {

g1.Add(Regex.Split(a, "\\|")[0].ToString());

Console.WriteLine("g1 here: " + Regex.Split(a, "\\|")[0].ToString());

List<String> rules = Regex.Split(a, "\\|").ToList();

for (int i = 1; i < rules.Count; i++) {

g1.Add(Regex.Split(a, "->")[0].ToString() + "->" + rules[i]);

Console.WriteLine("g1 here2: " + Regex.Split(a, " | ")[0][0].ToString() + " ->" + rules[i]);

}

} else {

g1.Add(a);

}

}

foreach(var rul in g1) {

String prod = Regex.Split(rul, "->")[1];

Console.WriteLine("PRODD::: ");

String nonterm = Regex.Replace(Regex.Split(rul, "->")[0], @"\s+", "");

if (Rules.ContainsKey(nonterm)) {

//Rules[nonterm].Add(Regex.Replace(prod, @"\s+", "").ToCharArray().Select(c => c.ToString()).ToList());7

List<String> ppro = prod.Split(' ').ToList();

ppro.RemoveAll(s => s == "");

Rules[nonterm].Add(ppro);

//Console.WriteLine("Test HERE: ", Regex.Replace(prod, @"\s+", "").ToCharArray().Select(c => c.ToString()).ToList());

} else {

var val = new List<List<String>>();

//List<String> pro = Regex.Replace(prod, @"\s+", "").ToCharArray().Select(c => c.ToString()).ToList();

List<String> pro = prod.Split(' ').ToList();

pro.RemoveAll(s => s == "");

val.Add(pro);

Rules.Add(nonterm, val);

}

}

foreach (String a in g1) {

String nonterm = Regex.Split(Regex.Replace(a, @"\s+", ""), "->")[0];

if (!regexNonTerm.Match(nonterm).Success) {

return new Tuple<int, String>(-1, "[-] Non Terminal must be Capitalized::AT:: " + nonterm);

} else {

nonterminals.Add(nonterm);

}

}

nonterminals = nonterminals.Distinct().ToList();

foreach (String a in g1) {

String term = Regex.Split(a, "->")[1];

foreach (String b in term.Split(' ')) {

if (regexNonTerm.Match(b).Success) {

if (!nonterminals.Contains(b)) {

return new Tuple<int, String>(-1, "[-] Remove Useless Symbol::AT:: " + b);

}

} else {

terminals.Add(b);

}

}

}

terminals = terminals.Distinct().ToList();

terminals.RemoveAll(s => s == "");

return new Tuple<int, String>(0, "[+] Success");

}

public List<String> Calc\_FirstSet(String s, IDictionary<String, List<List<String>>> prod) {

// Prod => { A:[ ['a','b'], ['e','d'] ], B: ... }

List<String> first\_set = new List<String>();

for(int i = 0; i < prod[s].Count; i++) {

for(int j = 0; j < prod[s][i].Count; j++) {

String item = prod[s][i][j];

if(item == s) {

break;

}

else if (regexNonTerm.Match(item).Success) {

var f = Calc\_FirstSet(item, prod);

if (!f.Contains("#")) {

foreach(var k in f) {

first\_set.Add(k);

}

break;

} else {

if(j == (prod[s][i].Count - 1)) {

foreach (var k in f) {

first\_set.Add(k);

}

} else {

f.Remove("#");

foreach (var k in f) {

first\_set.Add(k);

}

}

}

} else {

first\_set.Add(item);

break;

}

}

}

return first\_set.Distinct().ToList();

}

public List<String> Calc\_FollowSet(String s, IDictionary<String, List<List<String>>> prod) {

List<String> follow\_set = new List<String>();

if (s.Length == 0) {

return new List<String>();

}

if (s == prod.Keys.ElementAt(0)) {

follow\_set.Add("$");

}

foreach (String i in prod.Keys.ToArray()) {

for (int j = 0; j < prod[i].Count; j++) {

if (prod[i][j].Contains(s)) {

int idx = prod[i][j].IndexOf(s);

if (idx == (prod[i][j].Count - 1)){

if (prod[i][j][idx] == i) {

break;

} else {

var f = Calc\_FollowSet(i, prod);

foreach (var x in f) {

follow\_set.Add(x);

}

}

} else {

while (idx != (prod[i][j].Count - 1)) {

idx = idx + 1;

if (!regexNonTerm.Match(prod[i][j][idx]).Success) {

follow\_set.Add(prod[i][j][idx]);

break;

} else {

var f = Calc\_FirstSet(prod[i][j][idx], prod);

if (!f.Contains("#")) {

foreach (var x in f) {

follow\_set.Add(x);

}

break;

} else if (f.Contains("#") && idx != (prod[i][j].Count - 1)) {

f.Remove("#");

foreach (var k in f) {

follow\_set.Add(k);

}

} else if (f.Contains("#") && idx == (prod[i][j].Count - 1)) {

f.Remove("#");

foreach (var k in f) {

follow\_set.Add(k);

}

f = Calc\_FollowSet(i, prod);

foreach (var k in f) {

follow\_set.Add(k);

}

}

}

}

}

}

}

}

return follow\_set.Distinct().ToList();

}

public void Find\_FirstSet() {

Console.WriteLine("--------------FIRST SET----------------");

foreach (var item in nonterminals) {

FIRST\_SET.Add(item, Calc\_FirstSet(item, Rules));

Console.Write(item + " : ");

foreach (var val in Calc\_FirstSet(item, Rules)) {

Console.Write(" " + val);

}

Console.WriteLine("\n------------------------------");

}

}

public void Find\_FollowSet() {

Console.WriteLine("---------------FOLLOW SET---------------");

foreach (var item in nonterminals) {

FOLLOW\_SET.Add(item, Calc\_FollowSet(item, Rules));

Console.WriteLine("FOLLOW "+String.Join("",FOLLOW\_SET[item]));

Console.Write(item + " : ");

foreach (var val in Calc\_FollowSet(item, Rules)) {

Console.Write(" " + val);

}

Console.WriteLine("\n------------------------------");

}

}

}

}

**InputParsing.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Text.RegularExpressions;

using System.Threading.Tasks;

namespace SLR\_parser {

public class InputParsing {

public List<String> ParsingStack = new List<string>();

public List<String> InputTape = new List<string>();

public List<List<String>> Table = new List<List<string>>();

public List<List<String>> StackTable = new List<List<string>>();

public IDictionary<int, List<List<String>>> numbered\_rules = new Dictionary<int, List<List<String>>>();

public InputParsing(List<List<String>> Table, IDictionary<int, List<List<String>>> numbered\_rules) {

int count = 0;

foreach (var x in Table) {

//Console.WriteLine(count + ": " + String.Join(" ", x));

x.RemoveAt(0);

count++;

}

this.Table = Table;

this.numbered\_rules = numbered\_rules;

}

public List<List<String>> parse(String Input, List<String> cols) {

// Convert Input to list and add $ at the end

//InputTape.Add("((a),a,(a,a))".ToCharArray().Select(c => c.ToString()));

foreach (var a in Input.ToCharArray()) {

if(a != ' ')

InputTape.Add(a.ToString());

}

InputTape.Add("$");

// Add $0 to stack

ParsingStack.Add("$\_0");

//for(int i = 0; i<Table.Count; i++) {

// for(int j = 0; j<Table.Max(l => l.Count); j++) {

// Console.WriteLine("Row: "+i+" Col: "+j+" : "+Table[i][j]);

// }

//}

//for(int i = 0; i < cols.Count; i++) {

// Console.WriteLine("I: "+i+" : "+cols[i]);

//}

String Action = "";

List<String> l1 = new List<string>();

while(InputTape.Count >= 0) {

l1.Clear();

l1.Add(String.Join(" ", ParsingStack));

l1.Add(String.Join(" ", InputTape));

l1.Add(Action);

String stackTop = ParsingStack[ParsingStack.Count - 1];

String inputTop = InputTape[0];

Console.WriteLine(cols.IndexOf(inputTop));

String TableLookup = Table[int.Parse(stackTop.Split('\_')[1].ToString())][cols.IndexOf(inputTop)];

if (TableLookup.Contains("S")) { // If Shift Rule Exists

InputTape.RemoveAt(0);

ParsingStack.Add(inputTop+"\_"+ int.Parse(Regex.Match(TableLookup.ToString(), @"\d+").Value));

//Console.WriteLine("ACTION: SHIFT-"+ TableLookup[1]);

Action = "SHIFT-"+ int.Parse(Regex.Match(TableLookup.ToString(), @"\d+").Value);

}

else if (TableLookup.Contains("R")) { // If Reduce Rule Exists

//Console.WriteLine("R: "+ TableLookup[1]);

//Console.WriteLine(numbered\_rules[TableLookup[1]][0] + " -> "+String.Join("", numbered\_rules[TableLookup[1]][1]));

Console.WriteLine(numbered\_rules[int.Parse(TableLookup[1].ToString())][0]);

Console.WriteLine(numbered\_rules[int.Parse(TableLookup[1].ToString())][1]);

String test = String.Join(",", numbered\_rules[int.Parse(TableLookup[1].ToString())][1]);

int cou = int.Parse(Regex.Match(TableLookup.ToString(), @"\d+").Value);

Console.WriteLine(numbered\_rules[cou][1].Count);

int pops = numbered\_rules[cou][1].Count;

for(int pop = 0; pop < pops; pop++) {

ParsingStack.RemoveAt(ParsingStack.Count - 1);

}

//Console.WriteLine(int.Parse(ParsingStack[ParsingStack.Count - 1][1].ToString()));

//Console.WriteLine(cols.IndexOf(numbered\_rules[int.Parse(TableLookup[1].ToString())][0][0]));

String num = Table[int.Parse(ParsingStack[ParsingStack.Count - 1].Split('\_')[1].ToString())][cols.IndexOf(numbered\_rules[cou][0][0])];

ParsingStack.Add((numbered\_rules[cou][0][0]).ToString()+"\_"+num);

Action = "REDUCE " + numbered\_rules[cou][0][0] + " -> "+String.Join(" ", numbered\_rules[cou][1]);

}

else if (TableLookup.Contains("Accept")) { // Accepting at $ in input

//Console.WriteLine("ACCEPTED!");

Action = "Accept";

break;

}

else if(TableLookup.Contains("")) { // Error is no table entry exists

//Console.WriteLine("Error in Input");

Action = "Error";

break;

}

StackTable.Add(new List<String> (l1));

}

l1.Clear();

l1.Add(String.Join(" ", ParsingStack));

l1.Add(String.Join(" ", InputTape));

l1.Add(Action);

StackTable.Add(new List<String>(l1));

return StackTable;

}

}

}

**Lexer.cs**

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

using System.Text;

using System.Text.RegularExpressions;

using System.Threading.Tasks;

namespace SLR\_parser

{

class Lexer

{

public string test;

public Dictionary<string, Regex> dict = new Dictionary<string, Regex>();

public List<List<string>> tokens = new List<List<string>>();

public List<string> parsedInput = new List<string>();

public string input;

public string grammar;

public Lexer(string input, string grammar)

{

this.input = input;

this.grammar = grammar;

}

public List<List<string>> LexAnalysis()

{

Debug.WriteLine("Input In lexer", input);

Debug.WriteLine("Grammar In lexer", grammar);

//adding dictionary

dict.Add("integer", new Regex(@"^[0-9]+$"));

dict.Add("operator", new Regex(@"^[\+\-\\*]$"));

dict.Add("parenthesis", new Regex(@"^[\(\)]$"));

//splitting the input

parsedInput = input.Split(' ').ToList<string>();

//looping the parsedInput to match the tokens with dictionary

foreach(string parsedChar in parsedInput)

{

Debug.WriteLine(parsedChar + "\n");

//if(dict["integer"].Match(parsedChar))

Debug.WriteLine(dict["integer"].IsMatch(parsedChar));

if (dict["integer"].IsMatch(parsedChar))

{

List<string> list = new List<string>

{

"Integer",

parsedChar

};

tokens.Add(list);

}else if(dict["operator"].IsMatch(parsedChar))

{

List<string> list = new List<string>

{

"Operator",

parsedChar

};

tokens.Add(list);

}

else if (dict["parenthesis"].IsMatch(parsedChar))

{

List<string> list = new List<string>

{

"Parenthesis",

parsedChar

};

tokens.Add(list);

}

}

return tokens;

}

}

}

**Rules.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SLR\_parser

{

class Rules

{

private static string rule;

private static string semantics;

public static string getRule()

{

rule = "" + "1) exp -> exp + term\n" +

"2) exp -> exp - term\n" +

"3) exp -> term\n" +

"4) term -> term \* factor\n" +

"5) term -> factor\n" +

"6) factor -> ( exp )\n" +

"7) factor -> number";

return rule;

}

public static string getSemantics()

{

semantics = "" + "1) exp1.val = exp2.val + term.val\n" +

"2) exp1.val = exp2.val - term.val\n" +

"3) exp.val = term.val\n" +

"4) term1.val = term2.val\*factor.val\n" +

"5) term.val = factor.val\n" +

"6) factor.val = exp.val\n" +

"7) factor.val = num.val\n" +

"8) num.val = 0\n" +

"9) num.val = 1\n" +

"10) num.val = 2\n" +

"11) num.val = 3\n" +

"12) num.val = 4\n" +

"13) num.val = 5\n" +

"14) num.val = 6\n" +

"15) num.val = 7\n" +

"16) num.val = 8\n" +

"17) num.val = 9";

return semantics;

}

}

}

**SemanticAnalyzer.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Text.RegularExpressions;

using System.Threading.Tasks;

namespace SLR\_parser

{

class Rules {

private static string rule;

private static string semantics;

public static string getRule() {

rule = "" + "1) exp -> exp + term\n" +

"2) exp -> exp - term\n" +

"3) exp -> term\n" +

"4) term -> term \* factor\n" +

"5) term -> factor\n" +

"6) factor -> ( exp )\n" +

"7) factor -> number";

return rule;

}

public static string getSemantics() {

semantics = "" + "1) exp1.val = exp2.val + term.val\n" +

"2) exp1.val = exp2.val - term.val\n" +

"3) exp.val = term.val\n" +

"4) term1.val = term2.val\*factor.val\n" +

"5) term.val = factor.val\n" +

"6) factor.val = exp.val\n" +

"7) factor.val = num.val\n" +

"8) num.val = 0\n" +

"9) num.val = 1\n" +

"10) num.val = 2\n" +

"11) num.val = 3\n" +

"12) num.val = 4\n" +

"13) num.val = 5\n" +

"14) num.val = 6\n" +

"15) num.val = 7\n" +

"16) num.val = 8\n" +

"17) num.val = 9";

return semantics;

}

}

class SemanticAnalyzer

{

string parseTree;

string rules;

string output;

public SemanticAnalyzer(string parseTree, string rules)

{

this.parseTree = parseTree;

this.rules = rules;

}

public string performSDD()

{

var tokenList = new List<int>();

var stringBuilder = new StringBuilder();

var random = new Random();

for (int i = 0; i < 1000; i++)

{

tokenList.Add(random.Next(0, 100));

}

var complexLambdaExpression = tokenList.Where(x => x > 200)

.OrderByDescending(x => x)

.Select(x => x \* 2)

.ToList();

foreach (var item in complexLambdaExpression)

{

stringBuilder.AppendLine(item.ToString());

}

Console.WriteLine(stringBuilder.ToString());

Console.WriteLine("test");

Console.ReadLine();

return Rules.getSemantics();

}

public string generateAnnotatedPTree()

{

string trimmedParseTree = parseTree.Trim();

trimmedParseTree = trimmedParseTree.Replace(" ", "");

char[] chars = trimmedParseTree.ToCharArray();

List<NonTerminal> list = new List<NonTerminal>();

string state = "exp";

NonTerminal ro = performSDT(state, chars, 0, chars.Length - 1);

if (ro.val != int.MinValue)

{

output += "exp.val = " + ro.val + "\n";

}

else

{

output += "ERROR" + "\n";

}

return output;

}

public NonTerminal performSDT(string state, char[] chars, int start, int end)

{

NonTerminal root = new NonTerminal(0);

if (state == "exp")

{

bool found = false;

int started = 0;

for (int i = end; i >= start; i--)

{

if (chars[i] == '(')

{

started--;

}

else if (chars[i] == ')')

{

started++;

}

if ((chars[i] == '+' || chars[i] == '-') && (started == 0))

{

found = true;

NonTerminal left = performSDT("exp", chars, start, i - 1);

output += "exp.val = " + left.val + "\n";

NonTerminal right = performSDT("term", chars, i + 1, end);

output += "term.val = " + right.val + "\n";

if (chars[i] == '+')

{

root.val = (left.val + right.val);

}

else

{

root.val = (left.val - right.val);

}

break;

}

}

if (!found)

{

NonTerminal term = performSDT("term", chars, start, end);

output += "term.val = " + term.val + "\n";

root.val = term.val;

}

}

else if (state == "term")

{

bool found = false;

int started = 0;

for (int i = end; i >= start; i--)

{

if (chars[i] == '(')

{

started--;

}

else if (chars[i] == ')')

{

started++;

}

if ((chars[i] == '\*') && (started == 0))

{

found = true;

NonTerminal left = performSDT("term", chars, start, i - 1);

output += "term.val = " + left.val + "\n";

NonTerminal right = performSDT("factor", chars, i + 1, end);

output += "factor.val = " + right.val + "\n";

root.val = (left.val \* right.val);

break;

}

}

if (!found)

{

NonTerminal factor = performSDT("factor", chars, start, end);

output += "factor.val = " + factor.val + "\n";

root.val = factor.val;

}

}

else if (state == "factor")

{

Regex num\_reg = new Regex("^[0-9]+$");

int stop = (end + 1) - start;

try

{

String str = new string(chars).Substring(start, stop);

if (num\_reg.IsMatch(str))

{

NonTerminal number = performSDT("number", chars, start, end);

output += "number.inval = " + number.val + "\n";

root.val = number.val;

}

else

{

NonTerminal exp = performSDT("exp", chars, start + 1, end - 1);

output += "exp.val = " + exp.val + "\n";

root.val = exp.val;

}

}

catch (Exception e)

{

Console.WriteLine(e);

//MessageBox.Show("Incorrect Input", "ERROR",

//MessageBoxButtons.OK, MessageBoxIcon.Error);

return root;

}

}

else if (state == "number")

{

string str = new string(chars);

int stop = (end + 1) - start;

int val = int.Parse(str.Substring(start, stop));

root.val = val;

}

return root;

}

}

}

public struct NonTerminal

{

public long val;

public int basee;

public string dtype;

public NonTerminal(int v)

{

val = int.MinValue;

basee = int.MinValue;

dtype = null;

}

}

**SLR\_DFA.cs**

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Text.RegularExpressions;

namespace SLR\_parser {

public class SLR\_DFA {

public IDictionary<String, List<List<String>>> Rules = new Dictionary<String, List<List<String>>>();

public IDictionary<Tuple<int, String>, int> stateMap = new Dictionary<Tuple<int, String>, int>();

public IDictionary<int, List<List<List<String>>>> statesDict = new Dictionary<int, List<List<List<String>>>>();

public IDictionary<int, List<List<String>>> numbered\_rules = new Dictionary<int, List<List<String>>>();

public IDictionary<String, List<List<String>>> diction = new Dictionary<String, List<List<String>>>();

public IDictionary<String, List<List<String>>> AugRulesNoDot = new Dictionary<String, List<List<String>>>();

public List<String> colss = new List<string>();

public List<List<List<String>>> AugRules = new List<List<List<String>>>();

public List<string> nonterminals = new List<string>();

public int stateCount = 0;

public String start\_symbol;

public SLR\_DFA(IDictionary<String, List<List<String>>> rules, List<String> nont, String start) {

this.Rules = rules;

this.nonterminals = nont;

this.start\_symbol = start;

//Augment Rule

List<String> a = new List<string>();

List<List<String>> b = new List<List<string>>();

a.Add(start);

b.Add(a);

AugRulesNoDot[start + "'"] = new List<List<string>>(b);

// Old rules

AugRulesNoDot[start] = new List<List<string>>(rules[start]);

}

public List<List<List<String>>> augmentGrammar(IDictionary<String, List<List<String>>> rules, List<String> nont, String start) {

//IDictionary<String, List<String>> New\_Rules = new Dictionary<String, List<String>>();

if (start.Contains("'")) {

start = start.Replace("'","");

}

String newChar = start + "'";

//while (nont.Contains(newChar)) {

// newChar += "'";

//}

List<List<List<String>>> newRules = new List<List<List<String>>>(); // [ [[nonterm] , ['.','terms'...]], [[nonterm] , ['.','terms'...]], ... ]

List<List<String>> inner\_main = new List<List<String>>(); // [[nonterm] , ['.','terms'...]]

List<String> inner\_rule = new List<String>(); // ['.','terms'...]

foreach (var rul in rules) {

foreach (List<String> ruls in rul.Value) { // A : [ [a,b], [c,d,#] ]

inner\_main.Clear();

inner\_rule.Clear();

inner\_rule.Add(".");

foreach (String item in ruls) {

inner\_rule.Add(item);

} // ['.', 'a', 'b']

inner\_main.Add(new List<String> { rul.Key });

inner\_main.Add(new List<String>(inner\_rule));

newRules.Add(new List<List<String>>(inner\_main));

}

}

AugRules = newRules;

// [ [ [nonterm] , ['.','terms'...] ], [[nonterm] , ['.','terms'...]], ... ]

// Numbered \_ Rules : {0: ["E'", ['E']], 1: ['E', ['E', '+', 't']], 2: ['E', ['t']]}

List<String> a = new List<string>();

List<String> b = new List<string>();

List<List<String>> c = new List<List<string>>();

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

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

a.Clear();

b.Clear();

c.Clear();

//Console.WriteLine("NTL :: "+newRules[i][0][0]);

a.Add(newRules[i][0][0]);

for(int j = 0; j < newRules[i][1].Count; j++) {

if(newRules[i][1][j] == ".") {

continue;

}

//Console.WriteLine(" TL :: " + newRules[i][1][j]);

b.Add(newRules[i][1][j]);

}

c.Add(new List<string>(a));

c.Add(new List<string> (b));

numbered\_rules[i] = new List<List<string>> (c);

//AugRulesNoDot[c[i][0]] = new List<List<string>>(c);

//numbered\_rules[i].Add( new List<List<string>>());

}

//Console.WriteLine(numbered\_rules);

return newRules;

}

public List<List<List<String>>> findClosure(List<List<List<String>>> input\_state , String symbol) {

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

// AugRule => [ [[nonterm] , ['.','terms'...]], [[nonterm] , ['.','terms'...]], ... ]

if (symbol == start\_symbol+"'") {

foreach (var rule in AugRules) {

if (rule[0][0] == symbol) {

closureset.Add(new List<List<String>> (rule));

}

}

} else {

closureset = new List<List<List<String>>> (input\_state);

}

int prevlen = -1;

while (prevlen != closureset.Count) {

prevlen = closureset.Count;

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

foreach(var rule in closureset) {

int indexOfDot = rule[1].IndexOf(".");

if (!rule[1][rule[1].Count - 1][0].ToString().Equals(".")) {

String dotPointsHere = rule[1][indexOfDot + 1];

foreach (var in\_rule in AugRules) {

// This Check is not working

//Console.WriteLine("TEST 1--- "+tempclosureset.Contains(in\_rule));

//Console.WriteLine("dotPointsHere: "+ dotPointsHere);

//Console.WriteLine("in\_rule[0][0]: "+ in\_rule[0][0]);

//Console.WriteLine("TEST 2--- " + dotPointsHere.Equals(in\_rule[0][0]));

if (dotPointsHere.Equals(in\_rule[0][0]) && !tempclosureset.Contains(in\_rule)) {

tempclosureset.Add(in\_rule);

}

}

}

}

List<String> val = new List<string>();

foreach (var r in closureset) {

val.Add(String.Join(" ",r[0])+String.Join(" ",r[1])); // E'. E

}

foreach (var rule in tempclosureset) {

// This check is now not working

// Create Flat ClouseSet String for comparison

String rule\_string = String.Join(" ", rule[0]) + String.Join(" ", rule[1]);

if (!val.Contains(rule\_string)) {

closureset.Add(new List<List<string>> (rule));

}

}

}

//Console.WriteLine(closureset);

return closureset;

}

public void generateStates(IDictionary<int, List<List<List<String>>>> statesDict) {

int prev\_len = -1;

List<int> called\_GOTO\_on = new List<int>();

List<int> keys = new List<int>();

while (statesDict.Count != prev\_len) {

prev\_len = statesDict.Count;

keys = statesDict.Keys.ToList();

foreach(var key in keys) {

if (!called\_GOTO\_on.Contains(key)) { // 1st Point

called\_GOTO\_on.Add(key);

compute\_GOTO(key);

}

}

}

}

public void compute\_GOTO(int state) {

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

foreach (var rule in statesDict[state]) {

if (!rule[1][rule[1].Count - 1][0].ToString().Equals(".")) { // 2nd point

int indexOfDot = rule[1].IndexOf(".");

String dotPointsHere = rule[1][indexOfDot + 1];

if (!generateStatesFor.Contains(dotPointsHere)) {

generateStatesFor.Add(dotPointsHere);

}

}

}

if (generateStatesFor.Count != 0) {

foreach(var symbol in generateStatesFor) {

GOTO(state, symbol);

}

}

}

public void GOTO(int state, String charNextToDot) {

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

foreach (var rule in statesDict[state]) {

int indexOfDot = rule[1].IndexOf("."); // 3rd point

if (!rule[1][rule[1].Count - 1][0].ToString().Equals(".")) {

if (rule[1][indexOfDot+1] == charNextToDot) {

List<List<String>> shiftedRule = new List<List<string>>();

foreach(var r in rule) {

shiftedRule.Add(new List<String>(r));

}

shiftedRule[1][indexOfDot] = shiftedRule[1][indexOfDot + 1]; // TODO CAN HAVE ERRORS HERE

shiftedRule[1][indexOfDot + 1] = ".";

newState.Add(shiftedRule);

}

}

}

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

foreach(var rule in newState) {

int indexOfDot = rule[1].IndexOf(".");

if (!rule[1][rule[1].Count - 1][0].ToString().Equals(".")) {

List<List<List<String>>> closureRes = new List<List<List<string>>>();

closureRes = findClosure(newState, rule[1][indexOfDot + 1]);

foreach(var rule2 in closureRes) {

if(!addClosureRules.Contains(rule2) && !newState.Contains(rule2)) {

addClosureRules.Add(rule2); // CHECK FOR ERRORS HERE

}

}

}

}

foreach(var rule in addClosureRules) {

newState.Add(new List<List<string>> (rule));

}

int stateExists = -1;

// newState to String

String newState\_string = "";

foreach (var x in newState) {

newState\_string = newState\_string + x[0][0] + String.Join(" ", x[1]);

}

foreach(var state\_num in statesDict.Keys) {

String newStateDict\_string = "";

foreach (var x in statesDict[state\_num]) {

newStateDict\_string = newStateDict\_string + x[0][0] + String.Join(" ", x[1]);

}

if(newStateDict\_string.Equals(newState\_string)) { // <----------- [ Problem ]

stateExists = state\_num;

break;

}

}

if(stateExists == -1) {

stateCount += 1;

statesDict[stateCount] = new List<List<List<String>>>(newState);

stateMap[(state, charNextToDot).ToTuple()] = stateCount;

} else {

stateMap[(state, charNextToDot).ToTuple()] = stateExists;

}

}

public List<List<String>> createParseTable(IDictionary<int, List<List<List<String>>>> statesDict, IDictionary<Tuple<int, String>, int> stateMap, List<string> T, List<string> NT, IDictionary<String, List<String>> FOLLOW\_SET) {

List<int> rows = new List<int>();

List<String> cols = new List<string>();

rows = statesDict.Keys.ToList();

IDictionary<int, List<List<List<String>>>> statesDict\_copy = new Dictionary<int, List<List<List<String>>>>();

// DEEP COPY DICT !!!

foreach (var key in statesDict.Keys) {

statesDict\_copy[key] = new List<List<List<string>>>(statesDict[key]);

}

foreach (var a in T) {

cols.Add(a);

}

cols.Add("$");

foreach (var a in NT) {

cols.Add(a);

}

List<List<String>> Table = new List<List<string>>();

List<String> tempRow = new List<string>();

for(int y = 0; y < cols.Count; y++) {

tempRow.Add("");

}

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

Table.Add(new List<string> (tempRow));

}

foreach (var entry in stateMap) {

int state = entry.Key.Item1;

String symbol = entry.Key.Item2;

int a = rows.IndexOf(state);

int b = cols.IndexOf(symbol);

if (NT.Contains(symbol)) {

Table[a][b] = Table[a][b] + stateMap[entry.Key];

} else if (T.Contains(symbol)) {

Table[a][b] = Table[a][b] + "S" + stateMap[entry.Key];

}

}

//IDictionary<int, List<List<String>>> numbered = new Dictionary<int, List<List<String>>>();

//IDictionary<int, String> rules = new Dictionary<int, String>();

//int key\_count = 0;

//Console.Write(statesDict);

//foreach (var rulee in AugRules) {

// Console.Write(statesDict);

// List<List<String>> temprule = new List<List<string>>();

// temprule = new List<List<string>>(rulee);

// //temprule[1].Remove(".");

// numbered[key\_count] = new List<List<String>>(temprule);

// key\_count += 1;

//}

List<List<String>> val = new List<List<string>>();

val.Add(new List<String> (AugRules[0][1]));

Rules[AugRules[0][0][0]] = val;

diction = Rules;

// AugRule => [ [[nonterm] , ['.','terms'...]], [[nonterm] , ['.','terms'...]], ... ]

//String addedR = AugRules[0][0][0] + "->" + String.Join(" ", AugRules[0][1]);

//rules[0] = addedR;

//foreach (var rul in rules ) {

// String[] k = Regex.Split(rul.Value,"->");

// k[0] = k[0].Trim();

// k[1] = k[1].Trim();

//}

//Console.Write(statesDict\_copy);

//Preprocessor p1 = new Preprocessor();

//p1.Rules = AugRulesNoDot;

//p1.Find\_FollowSet();

//Console.WriteLine(p1.FOLLOW\_SET);

foreach (var stateno in statesDict.Keys) {

foreach (var rule in statesDict[stateno]) {

//Console.WriteLine("\_\_\_\_---\_\_\_\_: " + rule[1][rule[1].Count - 1][0]);

//Console.Write("EQUAL TEST: ");

//Console.WriteLine(rule[1][rule[1].Count - 1][0].ToString().Equals("."));

if (rule[1][rule[1].Count - 1][0].ToString().Equals(".")) {

List<List<String>> temprule2 = new List<List<string>>();

temprule2 = new List<List<string>>(rule);

temprule2[1].Remove(".");

foreach(var key in numbered\_rules.Keys) {

//Console.WriteLine("Keys: "+key);

//Console.WriteLine(numbered\_rules[key]);

//Console.WriteLine(String.Join("", temprule2[0]) + String.Join("",temprule2[1]));

//Console.WriteLine(String.Join(" ", numbered\_rules[key][0]) + String.Join(" ", numbered\_rules[key][1]));

String a1 = String.Join("", temprule2[0]) + String.Join("", temprule2[1]);

String a2 = String.Join("", numbered\_rules[key][0]) + String.Join("", numbered\_rules[key][1]);

if (a2.Equals(a1)) {

//List<String> follow\_result = p1.Calc\_FollowSet(rule[0][0], AugRulesNoDot); // WATCH FOR ERRORS HERE (USE P1.FOLLOW\_SET)

//Console.WriteLine("\nFOLOFLOLOFLOFLFO: ");

//Console.WriteLine(String.Join(" ", FOLLOW\_SET["E"]));

//Console.WriteLine(rule[0][0]);

List<String> follow\_result = FOLLOW\_SET[rule[0][0]];

//Console.WriteLine("KEY: " + key + " StateNo I" + stateno + " Follow: " + String.Join(" ", follow\_result));

foreach (var col in follow\_result) {

int index = cols.IndexOf(col);

if(key == 0) {

Table[stateno][index] = "Accept";

} else {

Table[stateno][index] = Table[stateno][index] + "R"+key;

}

}

}

}

}

}

}

colss = cols;

//Console.WriteLine("Parsing Table");

//Console.WriteLine("I "+String.Join(" ", cols));

int count = 0;

foreach (var x in Table) {

//Console.WriteLine(count+": "+String.Join(" ",x));

count++;

}

return Table;

}

}

}

**ThreeAddressCode.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SLR\_parser {

class ThreeAddrCode {

private int tempCount = 0;

public string GenerateCode(Node root) {

StringBuilder sb = new StringBuilder();

GenerateCode(root, sb);

return sb.ToString();

}

private void GenerateCode(Node node, StringBuilder sb) {

if (node.Left != null) {

GenerateCode(node.Left, sb);

}

if (node.Right != null) {

GenerateCode(node.Right, sb);

}

if (node.Value == "+") {

string temp = GetTemp();

sb.AppendLine($"{temp} = {node.Left.Value} + {node.Right.Value}");

node.Value = temp;

} else if (node.Value == "-") {

string temp = GetTemp();

sb.AppendLine($"{temp} = {node.Left.Value} - {node.Right.Value}");

node.Value = temp;

} else if (node.Value == "\*") {

string temp = GetTemp();

sb.AppendLine($"{temp} = {node.Left.Value} \* {node.Right.Value}");

node.Value = temp;

} else if (node.Value == "/") {

string temp = GetTemp();

sb.AppendLine($"{temp} = {node.Left.Value} / {node.Right.Value}");

node.Value = temp;

}

}

private string GetTemp() {

return $"t{tempCount++}";

}

}

}

**SyntaxTree.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SLR\_parser {

class Node {

public string Value { get; set; }

public Node Left { get; set; }

public Node Right { get; set; }

public Node(string value) {

Value = value;

}

}

class MainSyntaxTree {

public Node getRoot(string[] postfixExpression) {

Stack<Node> stack = new Stack<Node>();

foreach (string token in postfixExpression) {

if (IsOperator(token)) {

Node node = new Node(token);

node.Right = stack.Pop();

node.Left = stack.Pop();

stack.Push(node);

} else if (token == "(") {

// Do nothing

} else if (token == ")") {

stack.Push(stack.Pop());

} else {

stack.Push(new Node(token));

}

}

//return stack.Pop();

stack.Pop();

return stack.Pop();

}

public string GenerateSyntaxTree(string[] postfixExpression) {

Stack<Node> stack = new Stack<Node>();

foreach (string token in postfixExpression) {

if (IsOperator(token)) {

Node node = new Node(token);

node.Right = stack.Pop();

node.Left = stack.Pop();

stack.Push(node);

} else if (token == "(") {

// Do nothing

} else if (token == ")") {

stack.Push(stack.Pop());

} else {

stack.Push(new Node(token));

}

}

//return stack.Pop();

stack.Pop();

Node outNode = stack.Pop();

String output = PrintTree(outNode);

Console.WriteLine("Expression Tree:");

//Stack<Node> stack = new Stack<Node>();

//stack.Push(val);

//int level = 0;

//while (stack.Count > 0) {

// Node node = stack.Pop();

// if (node != null) {

// Console.WriteLine(new String(' ', level \* 2) + "| " + node.Value + " |");

// output += new String(' ', level \* 2) + "| " + node.Value + " |" + "\n";

// if (node.Right != null) {

// stack.Push(node.Right);

// }

// if (node.Left != null) {

// stack.Push(node.Left);

// }

// if (node.Left != null || node.Right != null) {

// level++;

// }

// } else {

// level--;

// }

//}

Console.WriteLine(output);

return output;

}

public string PrintTree(Node root) {

string outp = "";

Stack<Tuple<Node, string, bool>> stack = new Stack<Tuple<Node, string, bool>>();

stack.Push(Tuple.Create(root, "", true));

while (stack.Count > 0) {

Tuple<Node, string, bool> tuple = stack.Pop();

Node node = tuple.Item1;

string indent = tuple.Item2;

bool last = tuple.Item3;

Console.Write(indent);

outp += indent;

if (last) {

Console.Write("\\-");

outp += "\\-";

indent += " ";

} else {

Console.Write("|-");

outp += "|-";

indent += "| ";

}

Console.WriteLine(node.Value);

outp += node.Value + "\n";

if (node.Right != null) {

stack.Push(Tuple.Create(node.Right, indent, true));

}

if (node.Left != null) {

stack.Push(Tuple.Create(node.Left, indent, node.Right == null));

}

}

Console.WriteLine(outp);

return outp;

}

private static bool IsOperator(string token) {

return token == "+" || token == "-" || token == "\*" || token == "/";

}

public string PrintSyntaxTree(Node root) {

Stack<Node> stack = new Stack<Node>();

stack.Push(root);

int level = 0;

String output = "";

while (stack.Count > 0) {

Node node = stack.Pop();

if (node != null) {

Console.WriteLine(new String(' ', level \* 2) + " " + node.Value);

output += new String(' ', level \* 2) + " " + node.Value + "\n";

if (node.Right != null) {

stack.Push(node.Right);

}

if (node.Left != null) {

stack.Push(node.Left);

}

if (node.Left != null || node.Right != null) {

Console.WriteLine(new String(' ', level \* 2) + "|");

output += new String(' ', level \* 2) + "|" + "\n";

level++;

}

} else {

level--;

}

}

Console.WriteLine(output);

return output;

}

//static void Main(string[] args) {

// string[] postfixExpression = { "2", "3", "(", "4", "\*", ")", "+" };

// Node root = GenerateSyntaxTree(postfixExpression);

// PrintSyntaxTree(root, 0);

//}

public string InfixToPostfix(string infix) {

Console.WriteLine("INFIX: "+infix);

// Create a stack to store operators

Stack<char> stack = new Stack<char>();

// Create a string to store the postfix expression

string postfix = "";

// Loop through the infix expression

for (int i = 0; i < infix.Length; i++) {

// Get the current character

char c = infix[i];

// If the current character is a digit, add it to the postfix expression

if (char.IsDigit(c)) {

// Check if the next character is also a digit

while (i + 1 < infix.Length && char.IsDigit(infix[i + 1])) {

// If the next character is a digit, add it to the postfix expression

postfix += c;

i++;

c = infix[i];

}

// Add the last digit and a space to the postfix expression

postfix += c + " ";

}

// If the current character is an opening parenthesis, push it to the stack

else if (c == '(') {

stack.Push(c);

}

// If the current character is a closing parenthesis, pop all the operators from the stack and add them to the postfix expression until you find an opening parenthesis

else if (c == ')') {

while (stack.Peek() != '(') {

postfix += stack.Pop() + " ";

}

stack.Pop();

}

// If the current character is an operator, pop all the operators from the stack and add them to the postfix expression until you find an operator with a lower precedence

else if (c == '+' || c == '-' || c == '\*' || c == '/' || c == '^') {

while (stack.Count > 0 && GetPrecedence(c) <= GetPrecedence(stack.Peek())) {

postfix += stack.Pop() + " ";

}

stack.Push(c);

}

}

// Pop all the remaining operators from the stack and add them to the postfix expression

while (stack.Count > 0) {

postfix += stack.Pop() + " ";

}

// Return the postfix expression

Console.WriteLine("POSTFIX: ");

Console.WriteLine(postfix.ToString());

return postfix;

}

// Method to get the precedence of an operator

public int GetPrecedence(char c) {

if (c == '+' || c == '-') {

return 1;

} else if (c == '\*' || c == '/') {

return 2;

} else if (c == '^') {

return 3;

} else {

return 0;

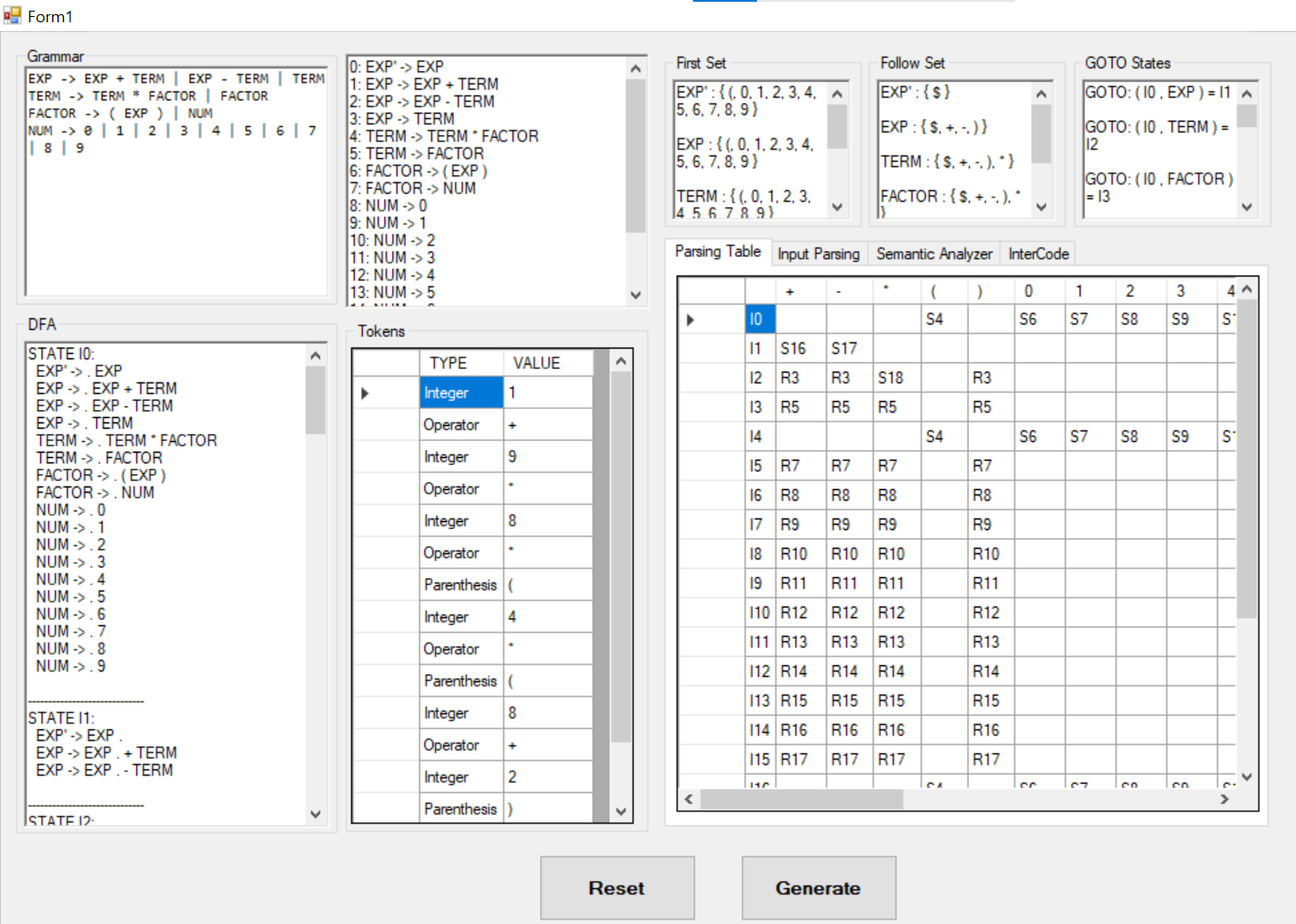
}

}

}

}

**Output:**

****

**A screenshot of a computer

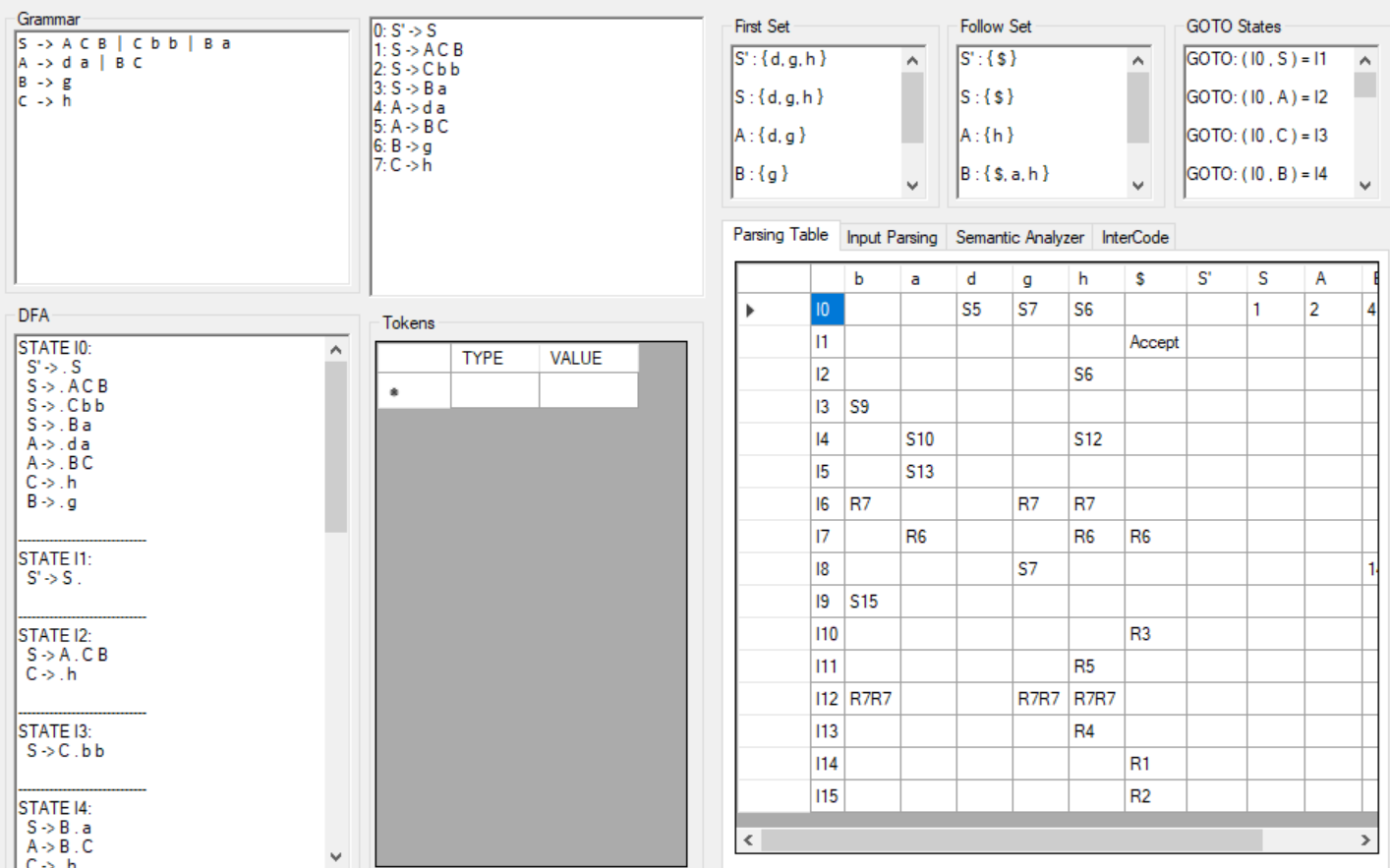
Description automatically generated with medium confidence**

**A screenshot of a computer program

Description automatically generated with medium confidence**

**A screenshot of a computer

Description automatically generated with medium confidence**

****