**Министерство науки и высшего образования Российской Федерации**

**Федеральное государственное бюджетное образовательное учреждение высшего образования  
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ  
УНИВЕРСИТЕТ имени Н.Э.БАУМАНА  
(национальный исследовательский университет)»**

Факультет: Информатика и системы управления

Кафедра: Теоретическая информатика и компьютерные технологии

**Лабораторная работа № 7.2**

“Самоприменимый генератор компиляторов на основе предсказывающего анализа”

по дисциплине «Конструирование компиляторов»

Вариант 1 (JAVA)

Работу выполнил

студент группы ИУ9-62Б

Сербин Денис

Москва, 2022

# **1. Цель работы**

Целью данной работы является изучение алгоритма построения таблиц предсказывающего анализатора.

**2. Исходные данные**

В данной лабораторной работе требуется разработать простейший генератор компиляторов, который по описанию грамматики рабочего языка, записанному на входном языке, строит таблицы предсказывающего анализа в виде составного литерала на целевом языке.

Здесь рабочий язык - это некоторый формальный язык с LL(1)-грамматикой, синтаксический анализатор которого должен быть построен генератором компиляторов.

В качестве входного языка генератора компиляторов должен выступать

язык представления правил грамматики, лексику и синтаксис которого можно восстановить по листингу 1.

non-terminal E, E1, T, T1, F;

terminal '+', '\*', '(',')', n;

E ::= T E1;

E1 ::= '+' T E1 epsilon;

T1 ::= '\*' F TI epsilon;

T ::= F T1;

F ::= n | ('E')';

axiom E;

Листинг 1 - пример языка алгебраических выражений описанный на

данном языке описания грамматик

И, наконец, целевым языком мы будем называть язык реализации компилятора рабочего языка. Так как разрабатываемый генератор компиляторов должен быть самоприменимым, целевой язык должен совпадать с языком реализации генератора компиляторов.

# **3. Задание**

Лабораторная работа делается на основе выполненной лабораторной работы 7.1. Выполнение данной лабораторной работы состоит из следующих этапов:

1. Переписывание грамматики входного языка на самом входном языке.

2. Добавление в программу, написанную на лабораторной работе 7.1, генератора таблицы разбора на основе дерева разбора. Таблица разбора должна представлять собой инициализированный двумерный массив на рабочем языке (он совпадает компиляторов с языком реализации генератора).

3. Тестирование генератора компиляторов путём написания простейшего калькулятора арифметических выражений на основе грамматики, описанной в листинге 1.

4. Раскрутка генератора компиляторов путём подачи на вход грамматики входного языка, написанной на самом входном языке (пункт 1) и замены таблицы разбора, написанной вручную, на сгенерированную таблицу разбора.

**4. Индивидуальный вариант**

# объявления

non-terminal E, E1, T, T1, F;

terminal '+', '\*', '(',')', n;

# правила грамматики

E ::= TE1;

E1 ::= ‘+' T E1 | epsilon;

T ::= F T1;

T1 ::= '\*' F T1 | epsilon;

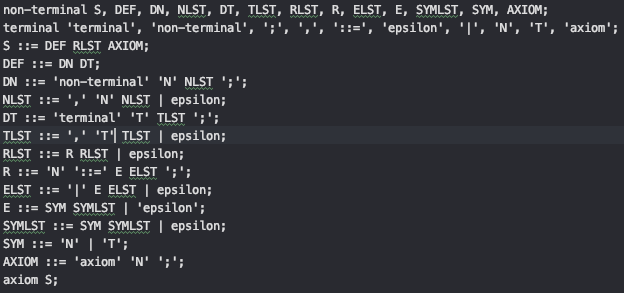
F ::= n | '(' E ')';

axiom E;

# **5. Реализация**

Грамматика входного языка на самом входном языке представлена на листинге 2.

На листинге 3 представлена реализация самоприменимого генератора компиляторов на основе предсказывающего анализа, реализованного на языке JAVA.



Листинг 2 - грамматика входного языка на самом входном языке

**public class** GrammarInterpreter **extends** Interpreter {  
 **private** ArrayList<String> terms = **new** ArrayList<>();  
 **private** ArrayList<String> nonterms = **new** ArrayList<>();  
 **private** Nonterm axiom = **new** Nonterm("S");  
 **private** RHS[][] q = **null**;  
 **private** ParseTree tree = **null**;  
 **private** HashMap<String, Rules> grammar\_list = **new** HashMap<>();  
  
 **public** GrammarInterpreter (ParseTree parse\_tree) {  
 **super**(parse\_tree);  
 tree = parse\_tree;  
 interpretTree();  
 checkForUndefinedNonterms();  
  
 System.*out*.println("TERMS: " + terms);  
 System.*out*.println("NONTERMS: " + nonterms);  
 System.*out*.println("AXIOM: " + axiom);  
 System.*out*.println("GRAMMAR: " + grammar\_list + "\n");  
 }  
  
 **public** CompilerGenerator getCompilerGenerator() {  
 **return new** CompilerGenerator(terms, nonterms, axiom, grammar\_list);  
 }  
  
 **private void** addNonterm(Token token) {  
 **if** (nonterms.contains(token.getImage())) {  
 System.*out*.println("\*\*\* Nonterminal <" + token.getImage() + "> " +  
 "defined twice at " + token.coordsToString() + " \*\*\*");  
 System.*exit*(1);  
 }  
 nonterms.add(token.getImage());  
 grammar\_list.put(token.getImage(), **new** Rules());  
 }  
  
 **private void** addTerm(Token token) {  
 token.setImage(token.getImage().replaceAll("[\\s']+", ""));  
 **if** (terms.contains(token.getImage())) {  
 System.*out*.println("\*\*\* Terminal <" + token.getImage() + "> " +  
 "defined twice at " + token.coordsToString() + " \*\*\*");  
 System.*exit*(1);  
 }  
 terms.add(token.getImage());  
// grammar\_list.put(token.getImage(), new Rules());  
 }  
  
 **private void** checkForUndefinedNonterms() {  
 **boolean** error = **false**;  
 **for** (Map.Entry<String, Rules> entry: grammar\_list.entrySet()) {  
 Rules rule = entry.getValue();  
 **if** (rule.isEmpty()) {  
 System.*out*.println("\*\*\* No rules found for nonterminal <" + entry.getKey() + "> \*\*\*");  
 error = **true**;  
 }  
 **for** (RHS chunk: rule) {  
 **for** (Symbol symbol: chunk) {  
 **if** (symbol **instanceof** Nonterm && !nonterms.contains(symbol.getType())) {  
 System.*out*.println("\*\*\* Undefined nonterminal <" + symbol.getType() + "> " +  
 "at " + symbol.coordsToString() + " \*\*\*");  
 error = **true**;  
 }  
 }  
 }  
 }  
 **if** (error) {  
 System.*exit*(2);  
 }  
 }  
  
// DEF ::= D\_N D\_T  
 **private void** scanDef(ParseNode def){  
 scanDN((ParseNode)def.getChildAt(0));  
 scanDT((ParseNode)def.getChildAt(1));  
 }  
  
// D\_N ::= "non-terminal" N N\_LST ';'  
 **private void** scanDN(ParseNode dn){  
 Symbol symbol = ((ParseNode)dn.getChildAt(1)).getSymbol();  
 addNonterm((Token)symbol);  
 scanNList((ParseNode)dn.getChildAt(2));  
 }  
  
// N\_LST ::= ',' N N\_LST | eps  
 **private void** scanNList(ParseNode nonterm){  
 **while**(!nonterm.isLeaf()){  
 Symbol symbol = ((ParseNode)nonterm.getChildAt(1)).getSymbol();  
 addNonterm((Token)symbol);  
 nonterm = (ParseNode)nonterm.getChildAt(2);  
 }  
 }  
  
// D\_T ::= "terminal" T T\_LST ';'  
 **private void** scanDT(ParseNode dt){  
 Symbol symbol = ((ParseNode)dt.getChildAt(1)).getSymbol();  
 addTerm((Token)symbol);  
 scanTList((ParseNode)dt.getChildAt(2));  
 }  
  
// T\_LST ::= ',' T T\_LST | eps  
 **private void** scanTList(ParseNode term){  
 **while** (!term.isLeaf()){  
 Symbol symbol = ((ParseNode)term.getChildAt(1)).getSymbol();  
 addTerm((Token)symbol);  
 term = (ParseNode)term.getChildAt(2);  
 }  
 }  
  
// R\_LST ::= R R\_LST | eps  
 **private void** scanRList(ParseNode rule){  
 **while**(!rule.isLeaf()){  
 scanR((ParseNode)rule.getChildAt(0));  
 rule = (ParseNode)rule.getChildAt(1);  
 }  
 }  
  
// R ::= N "::=" E E\_LST ';'  
 **private void** scanR(ParseNode rule){  
 String N = ((Token)(rule.getSymbolAt(0))).getImage();  
 **if** (grammar\_list.containsKey(N)) {  
 RHS E = scanE((ParseNode)rule.getChildAt(2));  
 Rules Elst = scanEList((ParseNode)rule.getChildAt(3));  
 Rules rules = **new** Rules(E);  
 rules.addAll(Elst);  
 Rules union\_rules\_list = grammar\_list.get(N);  
 union\_rules\_list.addAll(rules);  
 grammar\_list.put(N, union\_rules\_list);  
 }**else** {  
 Token tok = (Token)(rule.getSymbolAt(0));  
 System.*out*.println("\*\*\* A rule for undefined nonterminal <" + tok.getImage() + "> "+  
 "at " + tok.coordsToString() + " \*\*\*");  
 }  
 }  
  
// E\_LST ::= '|' E E\_LST | eps  
 **private** Rules scanEList(ParseNode expr){  
 Rules rules = **new** Rules();  
 **while** (!expr.isLeaf()) {  
 rules.add(scanE((ParseNode) expr.getChildAt(1)));  
 expr = (ParseNode) expr.getChildAt(2);  
 }  
 **return** rules;  
 }  
  
// E ::= SYM SYM\_LST | "epsilon"  
 **private** RHS scanE(ParseNode rule){  
 **if** (rule.getChildCount() == 1) {  
 RHS res = **new** RHS(RHS.*EPSILON*);  
 res.setCoords(rule.getSymbolAt(0).getStart());  
 **return** res;  
 } **else** {  
 Symbol symbol = scanSym((ParseNode)rule.getChildAt(0));  
 RHS res = **new** RHS(symbol);  
 res.setCoords(symbol.getStart());  
 res.addAll(scanSymList((ParseNode)rule.getChildAt(1)));  
 **return** res;  
 }  
 }  
  
// SYM\_LST ::= SYM SYM\_LST | eps  
 **private** RHS scanSymList(ParseNode node){  
 RHS res = **new** RHS();  
 **while** (!node.isLeaf()) {  
 Symbol sym = scanSym((ParseNode)node.getChildAt(0));  
 res.add(sym);  
 node = (ParseNode)node.getChildAt(1);  
 }  
 **return** res;  
 }  
  
// SYM ::= N | T  
 **private** Symbol scanSym(ParseNode node){  
 Token sym = (Token)node.getSymbolAt(0);  
 String image = sym.getImage();  
// System.out.println(sym.toString());  
 **if** (sym.getType().equals(GrammarScanner.*TERMINAL*)) {  
 image = image.replaceAll("[\\s']+", "");  
 **if** (image.isEmpty()) image = " "; //Видимо, терминалом был пробельный символ  
 **return new** Term(image, sym.getStart(), sym.getFollow());  
 } **else** {  
 **return new** Nonterm(image, sym.getStart(), sym.getFollow());  
 }  
 }  
  
// AXIOM ::= "axiom" N ';'  
 **private void** scanAxiom(ParseNode axiom){  
 ParseNode axiom\_name = (ParseNode)axiom.getChildAt(1);  
 Token symbol = (Token)axiom\_name.getSymbol();  
 **this**.axiom = **new** Nonterm(symbol.getImage());  
 }  
  
 // S ::= DEF R\_LST AXIOM  
 **private void** interpretS(ParseNode root) {  
 scanDef((ParseNode)root.getChildAt(0));  
 scanRList((ParseNode)root.getChildAt(1));  
 scanAxiom((ParseNode)root.getChildAt(2));  
 }  
  
 **private void** interpretTree() {  
 interpretS((ParseNode)tree.getRoot());  
 }  
}

**public class** CompilerGenerator **extends** Parser {  
 **protected** HashMap<String, Rules> gLst;  
  
 **private** HashMap<String, HashSet<String>> FIRST = **new** HashMap<>();  
 **private** HashMap<String, HashSet<String>> FOLLOW = **new** HashMap<>();  
 **public** CompilerGenerator(ArrayList<String> terms, ArrayList<String> nonterms, Nonterm axiom,  
 HashMap<String, Rules> gLst) {  
 **this**.terms = terms;  
 **this**.nonterms = nonterms;  
 **this**.gLst = gLst;  
 **this**.axiom = axiom;  
 **for** (String t: nonterms) {  
 FIRST.put (t, **new** HashSet<>());  
 FOLLOW.put(t, **new** HashSet<>());  
 }  
 buildFIRST();  
 buildFOLLOW();  
 isLL1();  
 calculateDelta();  
 log.append("Terms: ").append(terms.toString()).append('\n')  
 .append("Nonterms: ").append(nonterms.toString()).append('\n')  
 .append("Axiom: ").append(axiom.toString()).append('\n')  
 .append("FIRST: ").append(FIRST.toString()).append('\n')  
 .append("FOLLOW: ").append(FOLLOW.toString()).append('\n');  
 **for** (**int** i = 0; i < nonterms.size(); ++i) {  
 **for** (**int** j = 0; j < terms.size(); ++j) {  
 log.append(String.*format*("q[%s][%s] = %s\n", nonterms.get(i), terms.get(j),  
 q[i][j] != **null** ? q[i][j].toString() : "ERROR"));  
 }  
 }  
 }  
  
 **private** HashSet<String> calculateFIRST(RHS chunk) {  
 HashSet<String> res = **new** HashSet<>();  
 **if** (chunk.equals(RHS.*EPSILON*)) {   
 res.add(Term.*EPSILON*);  
 **return** res;  
 }  
 **for** (Symbol symbol: chunk) {  
 **if** (symbol **instanceof** Term) {  
 res.add(symbol.getType());  
 **return** res;  
 }  
 HashSet<String> symbol\_first = FIRST.get(symbol.getType());  
 **if** (!symbol\_first.contains(Term.*EPSILON*)) {  
 res.addAll(symbol\_first);  
 **return** res;  
 } **else** {  
 HashSet<String> copy = **new** HashSet<>(symbol\_first);  
 copy.remove(Term.*EPSILON*);  
 res.addAll(copy);  
 }  
 }  
 res.add(Term.*EPSILON*);  
 **return** res;  
 }  
  
 **private void** buildFIRST() {  
 **for** (Map.Entry<String, Rules> pair: gLst.entrySet()) {  
 **for** (RHS chunk: pair.getValue()) {  
 **if** (chunk.isEmpty()) **continue**;  
 Symbol symbol = chunk.get(0);  
 **if** (symbol **instanceof** Term) {  
 FIRST.get(pair.getKey()).add(symbol.getType());  
 }  
 }  
 }  
 **boolean** changed;  
 **do** {  
 changed = **false**;  
 **for** (Map.Entry<String, Rules> pair: gLst.entrySet()) {  
 **for** (RHS chunk: pair.getValue()) {  
 changed |= (FIRST.get(pair.getKey())).addAll(calculateFIRST(chunk));  
 }  
 }  
 } **while** (changed);  
 }  
  
  
  
 **private void** buildFOLLOW() {  
 FOLLOW.get(axiom.getType()).add(Term.*EOF*);  
 **for** (Rules rule: gLst.values()) {  
 **for** (RHS chunk: rule) {  
 **if** (chunk.isEmpty()) **continue**;  
 **for** (**int** i = 0; i < chunk.size() - 1; ++i) {  
 Symbol symbol = chunk.get(i);  
 **if** (symbol **instanceof** Nonterm) {  
 HashSet<String> sublist\_first = calculateFIRST(  
 **new** RHS(  
 chunk.subList(i+1,   
 chunk.size()  
 )  
 )  
 );  
 sublist\_first.remove(Term.*EPSILON*);  
 FOLLOW.get(symbol.getType()).addAll(sublist\_first);  
 }  
 }  
 }  
 }  
  
 **boolean** changed;  
 **do** {  
 changed = **false**;  
 **for** (Map.Entry<String, Rules> pair: gLst.entrySet()) {  
 String X = pair.getKey();  
 **for** (RHS chunk: pair.getValue()) {  
 **if** (chunk.isEmpty()) **continue**;  
 **int** last\_elem = chunk.size() - 1;  
 Symbol Y = chunk.get(last\_elem);  
 **if** (Y **instanceof** Nonterm) {  
 changed |= FOLLOW.get(Y.getType()).addAll(FOLLOW.get(X));  
 } **else** {  
 **continue**;  
 }  
 **for** (**int** i = last\_elem-1; i >= 0; --i) {  
 Y = chunk.get(i);  
 **if** (Y **instanceof** Term) **break**;  
 HashSet<String> sublist\_first = calculateFIRST(  
 **new** RHS(  
 chunk.subList(i+1,  
 chunk.size()  
 )  
 )  
 );  
 **if** (sublist\_first.contains(Term.*EPSILON*)) {  
 changed |= FOLLOW.get(Y.getType()).addAll(FOLLOW.get(X));  
 } **else** {  
 **break**;  
 }  
 }  
 }  
 }  
  
 } **while** (changed);  
 }  
  
**private boolean** isFIRSTAndFOLLOW(RHS u, HashSet<String> firstU,  
 RHS v, HashSet<String> firstV,  
 String A, HashSet<String> followA) {  
  
 HashSet<String> intersection\_uA = **new** HashSet<>(firstU);  
 intersection\_uA.retainAll(followA);  
 **if** (firstV.contains(Term.*EPSILON*) && !intersection\_uA.isEmpty()) {  
 StringBuilder log = **new** StringBuilder();  
 log.append("\*\* Grammar not LL(1): at " ).append(v.getCoords().toString()).append(' ')  
 .append(v.toString()).append(" =>\* epsilon")  
 .append("and FIRST (").append(u.toString()).append(" at ").append(u.getCoords().toString()).append(' ')  
 .append(") and FOLLOW (").append(A).append(") != empty").append('\n');  
 log.append("FIRST ").append(u.toString()).append(" = ").append(firstU.toString());  
 log.append("FOLLOW ").append(A).append(" = ").append(followA.toString()).append('\n');  
 System.*out*.print(log);  
 **this**.log.append(log);  
 **return true**;  
 }  
 **return false**;  
}  
  
 **private void** isLL1() {  
 **boolean** error = **false**;  
 **for** (Map.Entry<String, Rules> entry: gLst.entrySet()) {  
 String A = entry.getKey();  
 HashSet<String> follow\_A = FOLLOW.get(A);  
 Rules rules = entry.getValue();  
 **for** (**int** i = 0; i < rules.size() - 2; ++i) {  
 RHS u = rules.get(i);  
 HashSet<String> first\_u = calculateFIRST(u);  
 **for** (**int** j = i + 1; j < rules.size() - 1; ++j) {  
 RHS v = rules.get(j);  
 HashSet<String> first\_v = calculateFIRST(v);  
 HashSet<String> intersection = **new** HashSet<>(first\_u);  
 intersection.retainAll(first\_v);  
 **if** (!intersection.isEmpty()) {  
 error = **true**;  
 System.*out*.println("Grammar not LL(1): FIRST (u) and FIRST (v) != empty for " +  
 u + " at " + u.getCoords().toString() + " and " + v + " at " + v.getCoords().toString());  
 System.*out*.println("FIRST "+ u +" = " + calculateFIRST(u));  
 System.*out*.println("FIRST "+ v +" = " + calculateFIRST(v));  
 }  
 error |= isFIRSTAndFOLLOW(u, first\_u, v, first\_v, A, follow\_A);  
 error |= isFIRSTAndFOLLOW(v, first\_v, u, first\_u, A, follow\_A);  
 }  
 }  
 }  
 **if** (error) System.*exit*(3);  
 }  
  
 **private void** calculateDelta() {  
 **if** (!terms.contains(Term.*EOF*)) {  
 terms.add(Term.*EOF*);  
 }  
 **int** m = nonterms.size();  
 **int** n = terms.size();  
 q = **new** RHS[m][n];  
 **for** (RHS[] line : q) {  
 Arrays.*fill*(line, RHS.*ERROR*);  
 }  
 **for** (Map.Entry<String, Rules> pair: gLst.entrySet()) {  
 String X = pair.getKey();  
 **for** (RHS rule: pair.getValue()) {  
 HashSet<String> chunk\_first = calculateFIRST(rule);  
 **for** (String a: chunk\_first) {  
 **if** (!a.equals(Term.*EPSILON*)) {  
 q[nonterms.indexOf(X)][terms.indexOf(a)] = rule;  
 } **else** {  
 **for** (String b: FOLLOW.get(X)) {  
 q[nonterms.indexOf(X)][terms.indexOf(b)] = RHS.*EPSILON*;  
 }  
 }  
 }  
 }  
 }  
 }  
  
 **private** StringBuilder makeFile(ArrayList<String> list)  
 {  
 StringBuilder res = **new** StringBuilder(  
 " return new ArrayList<>(Arrays.asList(\n "  
 );  
 **if** (!list.isEmpty()) {  
 res.append('"').append(list.get(0)).append('"');  
 }  
 **for** (**int** i = 1; i < list.size(); ++i) {  
 res.append(", ").append('"').append(list.get(i)).append('"');  
 }  
 res.append("\n ));\n");  
 **return** res;  
 }  
  
 **public** String printCompiler(String classname) {  
 StringBuilder res = **new** StringBuilder(  
 "import syntax\_analyze.rules.RHS;\n" +  
 "import syntax\_analyze.Parser;\n" +  
 "import syntax\_analyze.symbols.Nonterm;\n" +  
 "import syntax\_analyze.symbols.Term;\n\n" +  
 "import java.util.ArrayList;\n" +  
 "import java.util.Arrays;\n\n" +  
 "public class " + classname + " {\n" +  
 " public final static ArrayList<String> terms = staticTermList();\n" +  
 " public final static ArrayList<String> nonterms = staticNontermList();\n" +  
 " public final static Nonterm axiom = " + axiom.printConstructor() + ";\n" +  
 " public final static RHS[][] q = staticDelta();\n\n" +  
 " public static Parser getParser() {\n" +  
 " return new Parser(terms, nonterms, axiom, q);\n" +  
 " }\n\n"  
 );  
 res.append(" private static ArrayList<String> staticNontermList() {\n")  
 .append(makeFile(nonterms))  
 .append(" }\n");  
  
 res.append(" private static ArrayList<String> staticTermList() {\n")  
 .append(makeFile(terms))  
 .append(" }\n");;  
  
  
 res.append(" private static RHS[][] staticDelta() {\n" +  
 " ArrayList<String> T = terms;\n" +  
 " ArrayList<String> N = nonterms;\n" +  
 " int m = N.size();\n" +  
 " int n = T.size();\n" +  
 " RHS[][] q = new RHS[m][n];\n" +  
 " for (RHS[] line: q) {\n" +  
 " Arrays.fill(line, RHS.ERROR);\n" +  
 " }\n");  
 **for** (**int** i = 0; i < q.length; ++ i) {  
 **for** (**int** j = 0; j < q[0].length; ++j) {  
 **if** (!RHS.*isError*(q[i][j])) {  
 res.append(String.*format*(" q[%d][%d] = ", i, j))  
 .append(q[i][j].printConstructor()).append(";\n");  
 }  
 }  
 }  
 res.append(" return q;\n }\n");  
 res.append("}\n");  
 **return** res.toString();  
 }  
  
 **public void** calculateJava(String path) {  
 File javafile = **new** File(path);  
 **try** {  
 Files.*write*(javafile.toPath(), printCompiler(javafile.getName().replace(".java", "")).getBytes());  
 } **catch** (IOException e) {  
 System.*err*.printf("file %s cannot be read\n", javafile.toPath());  
 }  
 }  
  
}

Листинг 3-генератор компиляторов на основе предсказывающего анализа

**6. Результаты**

После раскрутки компилятора путём подачи на вход грамматики входного языка, написанной на самом входном языке, на выходе получилась таблица разбора идентичная таблице разбора, составленной самостоятельно. Сгенерированная таблица разбора представлена на листинге 4.

**public class** GrammarStructure {  
 **public final static** ArrayList<String> terms = staticTermList();  
 **public final static** ArrayList<String> nonterms = staticNontermList();  
 **public final static** Nonterm axiom = **new** Nonterm("S");  
 **public final static** RHS[][] q = staticDelta();  
  
 **public static** Parser getParser() {  
 **return new** Parser(terms, nonterms, axiom, q);  
 }  
  
 **private static** ArrayList<String> staticNontermList() {  
 **return new** ArrayList<>(Arrays.asList(  
 "S", "DEF", "DN", "NLST", "DT", "TLST", "RLST", "R", "ELST", "E", "SYMLST", "SYM", "AXIOM"  
 ));  
 }  
 **private static** ArrayList<String> staticTermList() {  
 **return new** ArrayList<>(Arrays.asList(  
 "terminal", "non-terminal", ";", ",", "::=", "epsilon", "|", "N", "T", "axiom", "$"  
 ));  
 }  
 **private static** RHS[][] staticDelta() {  
 ArrayList<String> T = terms;  
 ArrayList<String> N = nonterms;  
 **int** m = N.size();  
 **int** n = T.size();  
 RHS[][] q = **new** RHS[m][n];  
 **for** (RHS[] line: q) {  
 Arrays.fill(line, RHS.ERROR);  
 }  
 q[0][1] = **new** RHS(  
 **new** Nonterm("DEF"),  
 **new** Nonterm("RLST"),  
 **new** Nonterm("AXIOM")  
 );  
 q[1][1] = **new** RHS(  
 **new** Nonterm("DN"),  
 **new** Nonterm("DT")  
 );  
 q[2][1] = **new** RHS(  
 **new** Term("non-terminal"),  
 **new** Term("N"),  
 **new** Nonterm("NLST"),  
 **new** Term(";")  
 );  
 q[3][2] = RHS.EPSILON;  
 q[3][3] = **new** RHS(  
 **new** Term(","),  
 **new** Term("N"),  
 **new** Nonterm("NLST")  
 );  
 q[4][0] = **new** RHS(  
 **new** Term("terminal"),  
 **new** Term("T"),  
 **new** Nonterm("TLST"),  
 **new** Term(";")  
 );  
 q[5][2] = RHS.EPSILON;  
 q[5][3] = **new** RHS(  
 **new** Term(","),  
 **new** Term("T"),  
 **new** Nonterm("TLST")  
 );  
 q[6][7] = **new** RHS(  
 **new** Nonterm("R"),  
 **new** Nonterm("RLST")  
 );  
 q[6][9] = RHS.EPSILON;  
 q[7][7] = **new** RHS(  
 **new** Term("N"),  
 **new** Term("::="),  
 **new** Nonterm("E"),  
 **new** Nonterm("ELST"),  
 **new** Term(";")  
 );  
 q[8][2] = RHS.EPSILON;  
 q[8][6] = **new** RHS(  
 **new** Term("|"),  
 **new** Nonterm("E"),  
 **new** Nonterm("ELST")  
 );  
 q[9][5] = **new** RHS(  
 **new** Term("epsilon")  
 );  
 q[9][7] = **new** RHS(  
 **new** Nonterm("SYM"),  
 **new** Nonterm("SYMLST")  
 );  
 q[9][8] = **new** RHS(  
 **new** Nonterm("SYM"),  
 **new** Nonterm("SYMLST")  
 );  
 q[10][2] = RHS.EPSILON;  
 q[10][6] = RHS.EPSILON;  
 q[10][7] = **new** RHS(  
 **new** Nonterm("SYM"),  
 **new** Nonterm("SYMLST")  
 );  
 q[10][8] = **new** RHS(  
 **new** Nonterm("SYM"),  
 **new** Nonterm("SYMLST")  
 );  
 q[11][7] = **new** RHS(  
 **new** Term("N")  
 );  
 q[11][8] = **new** RHS(  
 **new** Term("T")  
 );  
 q[12][9] = **new** RHS(  
 **new** Term("axiom"),  
 **new** Term("N"),  
 **new** Term(";")  
 );  
 **return** q;  
 }  
}

Листинг 4 - сгенерированная таблица разбора

**7. Вывод**

В ходе лабораторной работы были получены навыки написания самоприменимых генераторов компиляторов на основе предсказывающего анализа. Помимо этого, был разработан генератор компиляторов, способный по собственной грамматике вернуть таблицу разбора идентичную изначальной таблице разбора, другими словами самоприменимый генератор компиляторов.