Лабораторная работа № 3.1
«Самоприменимый генератор компиляторов на основе предсказывающего анализа»

16 мая 2023 г.

Самохвалова Полина, ИУ9-62Б

Цель работы

Целью данной работы является изучение алгоритма построения таблиц предсказывающего анализатора.

Индивидуальный вариант

```
$AXIOM E
$NTERM E' T T' F
$TERM  "+"  "*"  "("  ")"  "n"

* правила грамматики
  $RULE  E  = T E'
  $RULE  E' = "+" T E'
  $EPS
  $RULE  T  = F T'
  $RULE  T' = "*" F T'
  $EPS
  $RULE  F  = "n"
  "(" E ")"
```

Реализация

SelfMain.java

```
import grammar_parser.CompilerGenerator;
import grammar_parser.GrammarInterpreter;
import grammar_parser.GrammarScanner;
import grammar_parser.GrammarStructure;
```

```java
import lex_analyze.Scanner;
import syntax_analyze.Parser;

import java.io.File;

public class SelfMain {
    public static void main(String[] args) {
        String grammar_src = args[0];
        Scanner scanner = new GrammarScanner(grammar_src);
        Parser parser = GrammarStructure.getParser();
        parser.topDownParse(scanner);
        parser.addFile("output1" + File.separator +
                "grammar_graph.dot");

        GrammarInterpreter gr = new GrammarInterpreter
                (parser.getParseTree());
        CompilerGenerator cg = gr.getCompilerGenerator();
        cg.calculateJava("output1/GrammarStructure.java");
    }
}
```

Coords.java

```java
package lex_analyze;


public class Coords {
    private final int row;
    private int col;
    private int pos;

    public int getPos() {
        return pos;
    }

    public void setPos() {
        ++pos;
        ++col;
    }
    private Coords(int row, int col, int pos) {
        this.row = row;
        this.col = col;
        this.pos = pos;
    }

    public static Coords undefined() {
        return new Coords(-1, -1, -1);
```

```
    }

    public static Coords start() {
        return new Coords(1, 1, 0);
    }

    public Coords shift(int positions) {
        return new Coords(row, col + positions, pos + positions);
    }

    public Coords newline() {
        return new Coords(row + 1, 1, pos + 1);
    }

    @Override
    public String toString() {
        return pos > -1 ? String.format("(%d, %d)", row, col) : "?";
    }
}
```

Fragment.java

```
package lex_analyze;

public class Fragment {
    private final String image;
    private final Coords start;
    private final Coords follow;

    public Fragment(String image, Coords start, Coords follow) {
        this.image = image;
        this.start = start;
        this.follow = follow;
    }

    @Override
    public String toString(){
        return String.format("COMMENT %s-%s %s",
                start.toString(), follow.toString(), image);
    }
}
```

Scanner.java

```
package lex_analyze;

import syntax_analyze.symbols.Term;
```

```
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import java.util.Objects;
import java.util.regex.Matcher;
import java.util.regex.Pattern;


public class Scanner {
    public String NEWLINE = "newline";
    public String BLANK   = "blank";
    public String newline_epxr = "\\R";
    public String blank_expr = "[ \\t]+";

    public ArrayList<Token> tokens_list  = new ArrayList<>();

    public int index;


    protected HashMap<String, String> regexp;

    protected String text = "";
    private final Pattern p ;
    protected Matcher m;
    public Coords coord;
    protected String image = "";
    protected StringBuilder log = new StringBuilder();

    private final ArrayList<Fragment> comments  = new
            ArrayList<>();

    public static String makeGroup(String name, String expr) {
        return "(?<" + name + ">(" + expr + "))";
    }

    public String setPattern() {
        StringBuilder res = new StringBuilder(makeGroup(BLANK,
                blank_expr)+ "|" + makeGroup("comments",
                "\\*[^\\*\\n]*\\n"));
        for (Map.Entry<String, String> e: regexp.entrySet()) {
            res.append("|").append(makeGroup(e.getKey(),
                    e.getValue()));
        }
```

```
    return res.toString();

}

public Scanner(String filepath, HashMap<String, String>
        termsexpr) {
    File file = new File(filepath);
    try {
        text = new String(Files.readAllBytes(file.toPath()));
    } catch (IOException e) {
        System.err.printf("file %s cannot be read\n",
                file.toPath());
    }
    regexp = termsexpr;
    String pattern = setPattern();
    p = Pattern.compile(pattern, Pattern.DOTALL);
    m = p.matcher(text);
    coord = Coords.start();
    Token t = getNextToken();
    tokens_list.add(t);
    while (!Objects.equals(t.getType(), "$")) {
        t = getNextToken();
        tokens_list.add(t);
    }
    index = 0;
}

public Scanner(HashMap<String, String> termsexpr, String
        filepath){
    File file = new File(filepath);
    try {
        text = new String(Files.readAllBytes(file.toPath()));
    } catch (IOException e) {
        System.err.printf("file %s cannot be read\n",
                file.toPath());
    }
    regexp = termsexpr;
    String pattern = setPattern();
    p = Pattern.compile(pattern, Pattern.DOTALL);
    m = p.matcher(text);
    coord = Coords.start();
    Token t = getNextToken();
    tokens_list.add(t);
    while (!Objects.equals(t.getType(), "$")) {
        t = getNextToken();
        tokens_list.add(t);
```

```
    }
    index = 0;
}

protected boolean isType(String type) {
    return (image = m.group(type)) != null;
}

public String getText() {
    return text;
}

protected Token returnToken (String type) {
    Coords last = coord;
    if (Objects.equals(type, "NewLine")) {
        for (int i = 0; i < image.length(); i++) {
            coord = coord.newline();
        }
    } else {
        coord = coord.shift(image.length());
    }
    log.append(type).append(' ').append(last.toString())
            .append('-').append(coord.toString())
            .append(": <").append(image).append(">\n");
    return new Token(type, image, last, coord);
}

public Token getNextToken() {
    if (coord.getPos() >= text.length()) {
        return new Token(Term.EOF, coord);
    }
    String image;
    if (m.find()) {
        if (m.start() != coord.getPos()) {
            log.append(String.format("SYNTAX ERROR: %d",
                   coord.getPos())).append(coord.toString())
                    .append('\n');
            System.out.println(String.format
                    ("SYNTAX ERROR: %d",
                    coord.getPos()) + coord.toString());
            System.exit(-1);
        }
        if ((image = m.group(BLANK)) != null) {
            coord = coord.shift(image.length());
            return getNextToken();
        }
```

```
            if ((image = m.group("comments")) != null) {
                Coords last = coord;
                coord = coord.shift(image.length() - 1);
                comments.add(new Fragment(image, last, coord));
                coord = coord.newline();
                return getNextToken();
            }
            for (String s: regexp.keySet()) {
                if (isType(s)) {
                    return returnToken(s);
                }
            }
            System.out.println("ERROR " + coord.toString() + " "
                    + text.substring(coord.getPos()));
            return getNextToken();

        } else {
            log.append("SYNTAX ERROR: ").append(coord.toString())
                    .append('\n');
            log.append("SYNTAX ERROR: ").append(coord.toString())
                    .append('\n');
            System.out.println("SYNTAX ERROR: " + coord.toString());
            return new Token(Term.EOF, coord);
        }
    }

    public Token nextToken() {
        if (index < tokens_list.size() - 1) {
            if ((Objects.equals(tokens_list.get(index).getType(),
                    "NewLine")) && (Objects.equals(tokens_list
                    .get(index + 1).getType(), "AxiomKeyword") ||
                    Objects.equals(tokens_list.get(index + 1)
                            .getType(), "NTermKeyword") ||
                    Objects.equals(tokens_list.get(index + 1)
                            .getType(), "TermKeyword") ||
                    Objects.equals(tokens_list.get(index + 1)
                            .getType(), "RuleKeyword"))) {
                index++;
            }
            index++;
            return tokens_list.get(index - 1);
        } else if ((index == (tokens_list.size() - 1)) &&
                (!(Objects.equals(tokens_list.get(index)
                        .getType(), "NewLine")))) {
            index++;
            return tokens_list.get(index - 1);
```

```
        } else {
            return new Token(Term.EOF, coord);
        }
    }
}
```

Token.java

```
package lex_analyze;

import syntax_analyze.symbols.Term;


public class Token extends Term {
    private String image;

    public Token(String type, String image, Coords start,
                 Coords follow) {
        super(type, start, follow);
        this.image = image;
    }

    public Token(String type, Coords start) {
        super(type);
        this.image = "";
        this.start = start;
        this.follow = start;
    }

    public Token(String type) {
        super(type);
        this.image  = "";
    }

    @Override
    public String toString(){
        return String.format("lex_analyze.Token %s %s-%s <%s>",
                super.toString(), start.toString(), follow
                        .toString(), image);
    }

    @Override
    public String toDot() {
        return String.format("[label=\"%s\"][color=red]\n",
                toString().replaceAll("\"", "\\\\\""));
    }
```

```
    public String getImage() {
        return image;
    }

    public void setImage(String image){
        this.image = image;
    }
}
```

Epsilon.java

```
package syntax_analyze.rules;

public class Epsilon extends RHS {

}
```

Error.java

```
package syntax_analyze.rules;

public class Error extends RHS {
    @Override
    public String toString() {
        return "Error";
    }
}
```

RHS.java

```
package syntax_analyze.rules;

import lex_analyze.Coords;
import syntax_analyze.symbols.Symbol;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;

public class RHS extends ArrayList<Symbol> {

    private Coords coords  = Coords.undefined();

    public RHS(Symbol... symbols) {
        super(Arrays.asList(symbols));
    }

    public static final RHS EPSILON = new RHS();
    public static final RHS ERROR = null;
```

```
public static boolean isError(RHS rhs) {
    return rhs == null;
}

public RHS() {
    super();
}

public RHS(Collection<? extends Symbol> c) {
    super(c);
}

public RHS reverse() {
    RHS rule = new RHS();
    rule.ensureCapacity(this.size());
    for (int i = size() - 1; i >= 0; --i) {
        rule.add(get(i));
    }
    return rule;
}

public void setCoords(Coords c) {
    coords = c;
}

public Coords getCoords() {
    return coords;
}

public String printConstructor() {
    if (this.equals(EPSILON)) {
        return "RHS.EPSILON";
    }
    StringBuilder res = new StringBuilder("new RHS(");
    if (!isEmpty()) {
        res.append("\n                ").append(get(0)
                .printConstructor());
    }
    for (int i = 1; i < size(); ++i) {
        res.append(",\n                 ").append(get(i)
                .printConstructor());
    }
    res.append("\n               )");
    return  res.toString();
}
```

```
}
```

Rules.java

```
package syntax_analyze.rules;

import java.util.ArrayList;
import java.util.Arrays;

public class Rules extends ArrayList<RHS> {

    public Rules(RHS... rules) {
        super(Arrays.asList(rules));
    }

}
```

ParseNode.java

```
package syntax_analyze;

import lex_analyze.Token;
import syntax_analyze.rules.RHS;
import syntax_analyze.symbols.Symbol;

import javax.swing.tree.TreeNode;
import java.util.ArrayList;
import java.util.Enumeration;

public
class ParseNode implements TreeNode {
    private Symbol symbol;
    private int number = 0;
    private final ArrayList<ParseNode> children =
            new ArrayList<>();
    private ParseNode parent = null;

    public void setNumber(int number) {
        this.number = number;
    }

    public ParseNode(Symbol symbol) {
        this.symbol = symbol;
    }

    public ParseNode(Symbol symbol, ParseNode parent) {
        this.symbol = symbol;
        this.parent = parent;
```

```
}

public void setToken(Token token) {
    symbol = token;
}

public Symbol getSymbol() {
    return symbol;
}

public Symbol getSymbolAt(int n) {
    return children.get(n).symbol;
}

public void addChildren(RHS nodes) {
    for (Symbol s: nodes) {
        children.add(new ParseNode(s, this));
    }
}

private boolean isTheMostRightChild() {
    return (parent.getIndex(this) == (parent
            .children.size() - 1));
}

private boolean isRoot () {
    return parent == null;
}

public ParseNode succ() {
    if (isRoot()) return this;
    ParseNode res = this.parent;
    ParseNode prev = this;
    while(!res.isRoot() && prev.isTheMostRightChild()) {
        prev = res;
        res = res.parent;
    }

    if (res.isRoot() && prev.isTheMostRightChild()) {
        return this;
    }
    res = res.children.get(res.getIndex(prev) + 1);

    while (!res.isLeaf()) {
        res = res.children.get(0);
    }
```

```
    return res;
}

@Override
public TreeNode getChildAt(int childIndex) {
    return children.get(childIndex);
}

@Override
public int getChildCount() {
    return children.size();
}

@Override
public TreeNode getParent() {
    return parent;
}

@Override
public int getIndex(TreeNode node) {
    if (node instanceof ParseNode) {
        return children.indexOf(node);
    }
    else return -1;
}

@Override
public boolean getAllowsChildren() {
    return !(symbol instanceof Token);
}

@Override
public boolean isLeaf() {
    return children.isEmpty();
}

@Override
public Enumeration children() {
    return (Enumeration)children;
}

public String toDot() {
    StringBuilder res = new StringBuilder(number + " " +
            symbol.toDot());
    for (ParseNode child: children) {
```

```
            res.append(child.toDot());
            res.append(number).append("->").append(child.number)
                    .append("\n");
        }
        return res.toString();
    }
}
```

Parser.java

```
package syntax_analyze;

import lex_analyze.Scanner;
import lex_analyze.Token;
import syntax_analyze.rules.RHS;
import syntax_analyze.symbols.Nonterm;
import syntax_analyze.symbols.Symbol;
import syntax_analyze.symbols.Term;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.util.ArrayList;
import java.util.Stack;

public class Parser {

    protected ArrayList<String> terms;
    protected ArrayList<String> nonterms;
    protected Nonterm axiom;
    protected RHS[][] q;
    private ParseTree parse_tree = null;
    protected StringBuilder log = new StringBuilder();

    public Parser() {
    }

    public Parser(ArrayList<String> terms, ArrayList<String>
            nonterms, Nonterm axiom, RHS[][] q) {
        this.terms    = terms;
        this.nonterms = nonterms;
        this.axiom    = axiom;
        this.q = q;
    }

    private RHS delta(Nonterm N, Token T) {
        int i = nonterms.indexOf(N.getType());
```

```
    int j = terms.indexOf(T.getType());
    if (i == -1) {
        System.out.println("[Parser.java/delta]: no nontem "
                + N.toString() + " is found in " + nonterms);
    }
    if (j == -1) {
        System.out.println("[Parser.java/delta]: no term "
                + T.getType() + " is found in " + terms);
    }
    return q[i][j];
}

private void printError(Token tok, Symbol expected) {
    System.out.println("***ERROR: " + expected.toString()
            + " expected, got: " + tok.toString());
}

public ParseTree topDownParse(Scanner scanner) {
    log.setLength(0);
    Stack<Symbol> stack = new Stack<>();
    stack.push(new Term(Term.EOF));
    stack.push(axiom);
    parse_tree = new ParseTree(axiom);
    Token tok = scanner.nextToken();
    do {
        log.append(stack).append("-----------")
                .append(tok.toString()).append('\n');
        Symbol X = stack.pop();
        if (X instanceof Term) {
            if (X.equals(tok)) {
                parse_tree.setToken(tok);
                tok = scanner.nextToken();
            } else {
                printError(tok, X);
                return parse_tree;
            }
        } else {
            RHS nextRule = delta((Nonterm)X, tok);
            if (RHS.isError(nextRule)) {
                printError(tok, X);
                return parse_tree;
            } else {
                stack.addAll(nextRule.reverse());
                parse_tree.add(nextRule);
            }
        }
```

```
        } while (!stack.empty());
        return parse_tree;
    }

    public ParseTree getParseTree() {
        return parse_tree;
    }

    public void addFile(String path) {
        File dotfile = new File(path);
        try {
            Files.write(dotfile.toPath(), parse_tree.toDot()
                    .getBytes());
        } catch (IOException e) {
            System.err.printf("file %s cannot be read\n",
                    dotfile.toPath());
        }
    }

}
```

ParseTree.java

```
package syntax_analyze;

import lex_analyze.Token;
import syntax_analyze.rules.RHS;
import syntax_analyze.symbols.Nonterm;

import javax.swing.event.TreeModelListener;
import javax.swing.tree.TreeModel;
import javax.swing.tree.TreeNode;
import javax.swing.tree.TreePath;

public class ParseTree implements TreeModel {
    private final ParseNode root;
    private ParseNode current;
    private int current_number;

    public ParseTree(Nonterm axiom) {
        root = new ParseNode(axiom);
        current = root;
        current_number = 0;
        update();
    }

    private void update() {
```

```
    ++current_number;
    current.setNumber(current_number);
}

public void add(RHS rule) {
    if (!rule.isEmpty()) {
        current.addChildren(rule);
        current = (ParseNode)current.getChildAt(0);
    } else {
        current = current.succ();
    }
    update();
}

public void setToken(Token token) {
    current.setToken(token);
    current = current.succ();
    update();
}

public String toDot() {
    return "digraph {\n" + root.toDot() + "}\n";
}

@Override
public Object getRoot()
{
    return root;
}

public Object getChild(Object parent, int index)
{
    return ((ParseNode)parent).getChildAt(index);
}

@Override
public int getChildCount(Object parent) {
    return ((ParseNode)parent).getChildCount();
}

@Override
public boolean isLeaf(Object node) {
    return ((ParseNode)node).isLeaf();
}

@Override
```

```
    public void valueForPathChanged(TreePath path, Object newValue)
    {

    }

    @Override
    public int getIndexOfChild(Object parent, Object child) {
        return ((ParseNode)parent).getIndex((TreeNode)child);
    }

    @Override
    public void addTreeModelListener(TreeModelListener l) {

    }

    @Override
    public void removeTreeModelListener(TreeModelListener l) {

    }
}
```

Nonterm.java

```
package syntax_analyze.symbols;

import lex_analyze.Coords;

public class Nonterm extends Symbol {

    public Nonterm (String type) {
        super(type);
    }

    public Nonterm (String type, Coords start, Coords follow) {
        super(type, start, follow);
    }

    @Override
    public String toString() {
        return "<" + super.toString() + ">";
    }

    public String toDot() {
        return String.format("[label=\"%s\"][color=green]\n",
                getType());
    }
```

```
    public String printConstructor() {
        return "new Nonterm(\"" + getType() + "\")";
    }

}
```

Symbol.java

```
package syntax_analyze.symbols;

import lex_analyze.Coords;

public class Symbol {

    protected String type;

    protected Coords start, follow;

    public String getType() {
        return type;
    }

    protected Symbol(String type)
    {
        this.type = type;
        this.start  = Coords.undefined();
        this.follow = Coords.undefined();
    }

    protected Symbol (String type, Coords start, Coords follow)
    {
        this.type = type;
        this.start  = start;
        this.follow = follow;
    }

    @Override
    public String toString() {
        return type;
    }

    @Override
    public boolean equals (Object o) {
        return ((o instanceof Symbol) && type.equals(((Symbol)o)
                .type)) || ((o instanceof String) && type.equals(o));
    }
```

```
    public String toDot() {
        return "";
    }

    public String coordsToString() {
        return start.toString() + "-" + follow.toString();
    }

    public Coords getStart() {
        return start;
    }

    public Coords getFollow() {
        return follow;
    }

    public String printConstructor() {
        return "*** Error in Symbol.printConstructor(): " +
                "no public constructor";
    }

}
```

Term.java

```
package syntax_analyze.symbols;

import lex_analyze.Coords;

public class Term extends Symbol{
    public final static String EOF = "$";
    public final static String EPSILON = "";

    public Term (String type) {
        super(type);
    }

    public Term (String type, Coords start, Coords follow) {
        super(type, start, follow);
    }

    @Override
    public String toString() {
        return super.toString();
    }

    public String toDot() {
```

```
        return String.format("[label=\"%s\"][color=black]\n",
                getType());
    }

    @Override
    public String printConstructor() {
        return "new Term(" + getType() + ")";
    }
}
```

CompilerGenerator.java

```
package grammar_parser;

import syntax_analyze.Parser;
import syntax_analyze.rules.RHS;
import syntax_analyze.rules.Rules;
import syntax_analyze.symbols.Nonterm;
import syntax_analyze.symbols.Symbol;
import syntax_analyze.symbols.Term;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.util.*;

public class CompilerGenerator extends Parser {
    protected HashMap<String, Rules> gLst;

    private final HashMap<String, HashSet<String>> FIRST =
            new HashMap<>();
    private final 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).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 part) {
    HashSet<String> res = new HashSet<>();
    if (part.equals(RHS.EPSILON)) {
        res.add(Term.EPSILON);
        return res;
    }
    for (Symbol symbol: part) {
        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 part: pair.getValue()) {
            if (part.isEmpty()) continue;
            Symbol symbol = part.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 part: pair.getValue()) {
                changed |= (FIRST.get(pair.getKey()))
                        .addAll(calculateFIRST(part));
            }
        }
    } while (changed);
}



private void buildFOLLOW() {
    FOLLOW.get(axiom.getType()).add(Term.EOF);
    for (Rules rule: gLst.values()) {
        for (RHS part: rule) {
            if (part.isEmpty()) continue;
            for (int i = 0; i < part.size() - 1; ++i) {
                Symbol symbol = part.get(i);
                if (symbol instanceof Nonterm) {
                    HashSet<String> sublist_first =
                            calculateFIRST(
                            new RHS(
                                    part.subList(i+1,
                                            part.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 part: pair.getValue()) {
                    if (part.isEmpty()) continue;
                    int last_elem = part.size() - 1;
                    Symbol Y = part.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 = part.get(i);
                        if (Y instanceof Term) break;
                        HashSet<String> sublist_first =
                                calculateFIRST(
                                new RHS(
                                        part.subList(i+1,
                                                part.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> part_first = calculateFIRST(rule);
            for (String a: part_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;
}

private StringBuilder makeFileT(ArrayList<String> list)
{
    StringBuilder res = new StringBuilder(
            "        return new ArrayList<>" +
                    "(Arrays.asList(\n                "
    );
    if (!list.isEmpty()) {
        res.append(list.get(0));
    }
    for (int i = 1; i < list.size(); ++i) {
        res.append(", ").append(list.get(i));
    }
    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(makeFileT(terms))
        .append("    }\n");;


res.append("""
            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);
                }
        """);
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());
        }
    }

}
```

GrammarInterpreter.java

```
package grammar_parser;

import lex_analyze.Token;
import syntax_analyze.ParseNode;
import syntax_analyze.ParseTree;
import syntax_analyze.rules.RHS;
import syntax_analyze.rules.Rules;
import syntax_analyze.symbols.Nonterm;
import syntax_analyze.symbols.Symbol;
import syntax_analyze.symbols.Term;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import java.util.Objects;

public class GrammarInterpreter {
    private final ArrayList<String> terms = new ArrayList<>();
    private final ArrayList<String> nonterms = new ArrayList<>();
    private Nonterm axiom = new Nonterm("S");
    private final RHS[][] q = null;
    private final ParseTree tree;
    private final HashMap<String, Rules> grammar_list =
            new HashMap<>();
```

```
public GrammarInterpreter (ParseTree parse_tree) {
    super();
    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) {
    if (terms.contains(token.getImage())) {
        System.out.println("*** Terminal <" +
                token.getImage() + "> " + "defined twice at "
                + token.coordsToString() + " ***");
        System.exit(1);
    }
    terms.add(token.getImage());
}

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);
    }
}

// $RULE S = "AxiomKeyword" "Nterm" "NTermKeyword" "Nterm"
// NTERMS TERMS_DEF RULES_DEF
private void interpretS(ParseNode root) {
    ParseNode axiom_name = (ParseNode)root.getChildAt(1);
    Token symbol = (Token)axiom_name.getSymbol();
    addNonterm(symbol);
    this.axiom = new Nonterm(symbol.getImage());
    Symbol symbol1 = ((ParseNode)root.getChildAt(3))
            .getSymbol();
    addNonterm((Token)symbol1);
    scanNTERMS((ParseNode)root.getChildAt(4));
    scanTERMS_DEF((ParseNode)root.getChildAt(5));
    scanRULES_DEF((ParseNode)root.getChildAt(6));
}

// $RULE NTERMS = "Nterm" NTERMS
//               $EPS
private void scanNTERMS(ParseNode NTERMS) {
    if (!NTERMS.isLeaf()) {
        Symbol symbol = ((ParseNode)NTERMS.getChildAt(0))
                .getSymbol();
        addNonterm((Token)symbol);
```

```
        scanNTERMS((ParseNode)NTERMS.getChildAt(1));
    }
}

// $RULE TERMS_DEF = "TermKeyword" "Term" TERMS
private void scanTERMS_DEF(ParseNode TERMS_DEF) {
    Symbol symbol = ((ParseNode)TERMS_DEF.getChildAt(1))
            .getSymbol();
    addTerm((Token)symbol);
    scanTERMS((ParseNode)TERMS_DEF.getChildAt(2));
}

// $RULE TERMS = "Term" TERMS
//               $EPS

private void scanTERMS(ParseNode TERMS) {
    if (!TERMS.isLeaf()) {
        Symbol symbol = ((ParseNode)TERMS.getChildAt(0))
                .getSymbol();
        addTerm((Token)symbol);
        scanTERMS((ParseNode)TERMS.getChildAt(1));
    }
}

// $RULE RULES_DEF = RULE RULES
private void scanRULES_DEF(ParseNode RULES_DEF) {
    scanRULE((ParseNode)RULES_DEF.getChildAt(0));
    scanRULES((ParseNode)RULES_DEF.getChildAt(1));
}

// $RULE RULES = RULE RULES
//               $EPS

private void scanRULES(ParseNode RULES) {
    if (!RULES.isLeaf()) {
        scanRULE((ParseNode)RULES.getChildAt(0));
        scanRULES((ParseNode)RULES.getChildAt(1));
    }
}

// $RULE RULE = "RuleKeyword" "Nterm" "Equal" R

private void scanRULE(ParseNode RULE) {
    String Nterm = ((Token)(RULE.getSymbolAt(1)))
            .getImage();
    if (grammar_list.containsKey(Nterm)) {
```

```
        Rules rules = scanR((ParseNode)RULE.getChildAt(3));
        Rules union_rules_list = grammar_list.get(Nterm);
        union_rules_list.addAll(rules);
        grammar_list.put(Nterm, union_rules_list);
    } else {
        Token tok = (Token)(RULE.getSymbolAt(1));
        System.out.println("*** A rule for undefined " +
                "nonterminal <" + tok.getImage() + "> "+
                "at " + tok.coordsToString() + " ***");
    }
}

// $RULE R = R1 R2
private Rules scanR(ParseNode R) {
    Rules rules = new Rules();
    rules.add(scanR1((ParseNode)R.getChildAt(0)));
    rules.addAll(scanR2((ParseNode)R.getChildAt(1)));
    return rules;
}

// $RULE R1 = "Term" R3
//            "Nterm" R3
//            "EpsKeyword"
private RHS scanR1(ParseNode R1) {
    if (R1.getChildCount() == 1) {
        RHS res = new RHS(RHS.EPSILON);
        res.setCoords(R1.getSymbolAt(0).getStart());
        return res;
    } else {
        RHS res = new RHS();
        Token sym = (Token)R1.getSymbolAt(0);
        if (Objects.equals(sym.getType(), "Term")) {
            res.add(new Term(sym.getImage(),
                    sym.getStart(), sym.getFollow()));
        } else {
            res.add(new Nonterm(sym.getImage(),
                    sym.getStart(), sym.getFollow()));
        }
        res.addAll(scanR3((ParseNode)R1.getChildAt(1)));
        return res;
    }
}

// $RULE R3 = "Term" R3
//            "Nterm" R3
//            $EPS
```

```
    private RHS scanR3(ParseNode R3) {
        RHS res = new RHS();
        while (!R3.isLeaf()) {
            Token sym = (Token)R3.getSymbolAt(0);
            if (Objects.equals(sym.getType(), "Term")) {
                res.add(new Term(sym.getImage(), sym
                        .getStart(), sym.getFollow()));
            } else {
                res.add(new Nonterm(sym.getImage(),
                        sym.getStart(), sym.getFollow()));
            }
            R3 = (ParseNode)R3.getChildAt(1);
        }
        return res;
    }

    // $RULE R2 = "NewLine" R
    //            $EPS

    private Rules scanR2(ParseNode R2) {
        Rules rules = new Rules();
        if (!R2.isLeaf()) {
            rules.addAll(scanR((ParseNode)R2.getChildAt(1)));
        }
        return rules;
    }

    private void interpretTree() {
        interpretS((ParseNode)tree.getRoot());
    }
}
```

GrammarScanner.java

```
package grammar_parser;

import lex_analyze.Coords;
import lex_analyze.Scanner;
import lex_analyze.Token;

import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Objects;


public class GrammarScanner extends Scanner {
```

```
    public final static HashMap<String, String> regexp =
            staticRegExpressions();

    private static HashMap<String, String>
    staticRegExpressions() {
        LinkedHashMap<String, String> exprs =
                new LinkedHashMap<>();
        exprs.put("AxiomKeyword", "\\$AXIOM");
        exprs.put("NTermKeyword", "\\$NTERM");
        exprs.put("TermKeyword", "\\$TERM");
        exprs.put("RuleKeyword", "\\$RULE");
        exprs.put("EpsKeyword", "\\$EPS");
        exprs.put("Nterm", "[A-Z][A-Z_]*'|" +
                "[A-Z][A-Z_]*[0-9]|[A-Z][A-Z_]*");
        exprs.put("Term", "\"[a-zA-Z\\+\\*\\(\\)]+\"");
        exprs.put("Equal", "=");
        exprs.put("NewLine", "\\n+");
        return exprs;
    }
    public GrammarScanner(String filepath) {
        super(filepath, regexp);
    }
    @Override
    protected Token returnToken (String type) {
        Coords last = coord;
        if (Objects.equals(type, "NewLine")) {
            for (int i = 0; i < image.length(); i++) {
                coord = coord.newline();
            }
        } else {
            coord = coord.shift(image.length());
        }
        log.append(type).append(' ').append(last
                        .toString()).append('-')
                .append(coord.toString())
                .append(": <").append(image)
                .append(">\n");
        return new Token(type, image, last, coord);
    }
}
```

GrammarStructure.java

```
package grammar_parser;

import syntax_analyze.Parser;
```

```
import syntax_analyze.rules.Epsilon;
import syntax_analyze.rules.Error;
import syntax_analyze.rules.RHS;
import syntax_analyze.rules.Rules;
import syntax_analyze.symbols.Nonterm;
import syntax_analyze.symbols.Term;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;

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");

    final static HashMap<String, Rules> grammarList =
            getGrammar();
    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", "NTERMS",
                "TERMS_DEF", "TERMS", "RULES_DEF", "RULES",
                "RULE", "R", "R1", "R2", "R3"));
    }

    private static ArrayList<String> staticTermList() {
        return new ArrayList<>(Arrays.asList("AxiomKeyword",
                "NTermKeyword", "TermKeyword", "RuleKeyword",
                "EpsKeyword", "Term", "Nterm", "Equal",
                "NewLine", Term.EOF
        ));
    }

    private static HashMap<String, Rules> getGrammar() {
        HashMap<String, Rules> rules = new HashMap<>();
        rules.put("S", new Rules(new RHS(
                new Term("AxiomKeyword"),
                new Term("Nterm"),
                new Term("NTermKeyword"),
```

```
        new Term("Nterm"),
        new Nonterm("NTERMS"),
        new Nonterm("TERMS_DEF"),
        new Nonterm("RULES_DEF")
)));
rules.put("NTERMS",
        new Rules(new RHS(
                new Term("Nterm"),
                new Nonterm("NTERMS")
        ),
                new Epsilon())));

rules.put("TERMS_DEF", new Rules(new RHS(
        new Term("TermKeyword"),
        new Term("Term"),
        new Nonterm("TERMS")
)));
rules.put("TERMS", new Rules(
        new RHS(
                new Term("Term"),
                new Nonterm("TERMS")
        ),
        new Epsilon()
));
rules.put("RULES_DEF", new Rules(new RHS(
        new Nonterm("RULE"),
        new Nonterm("RULES")
)));
rules.put("RULE", new Rules(new RHS(
        new Term("RuleKeyword"),
        new Term("Nterm"),
        new Term("Equal"),
        new Nonterm("R")
)));
rules.put("RULES", new Rules(new RHS(
        new Nonterm("RULE"),
        new Nonterm("RULES")
),
        new Epsilon()
));
rules.put("R", new Rules(
        new RHS(
                new Nonterm("R1"),
                new Nonterm("R2")
        )
));
```

```
        rules.put("R1", new Rules(
                new RHS(
                        new Term("Term"),
                        new Nonterm("R3")
                ),
                new RHS(
                        new Term("Nterm"),
                        new Nonterm("R3")
                ),
                new RHS(
                        new Term("EpsKeyword")
                )
        ));
        rules.put("R3", new Rules(
                new RHS(
                        new Term("Term"),
                        new Nonterm("R3")
                ),
                new RHS(
                        new Term("Nterm"),
                        new Nonterm("R3")
                ),
                new Epsilon()
        ));
        rules.put("R2", new Rules(
                new RHS(
                        new Term("NewLine"),
                        new Nonterm("R")
                ),
                new Epsilon()
        ));

        return rules;
    }

    private static RHS[][] staticDelta() {
        ArrayList<String> T = terms;
        ArrayList<String> N = nonterms;
        HashMap<String, Rules> rules = grammarList;
        int m = N.size();
        int n = T.size();
        RHS[][] q = new RHS[m][n];
        for (RHS[] line: q) {
            Arrays.fill(line, new Error());
        }
```

```
q[N.indexOf("S")][T.indexOf("AxiomKeyword")] =
        rules.get("S").get(0);

q[N.indexOf("NTERMS")][T.indexOf("TermKeyword")] =
        rules.get("NTERMS").get(1);
q[N.indexOf("NTERMS")][T.indexOf("Nterm")] =
        rules.get("NTERMS").get(0);

q[N.indexOf("TERMS_DEF")][T.indexOf("TermKeyword")] =
        rules.get("TERMS_DEF").get(0);

q[N.indexOf("TERMS")][T.indexOf("Term")] =
        rules.get("TERMS").get(0);
q[N.indexOf("TERMS")][T.indexOf("RuleKeyword")] =
        rules.get("TERMS").get(1);

q[N.indexOf("RULES_DEF")][T.indexOf("RuleKeyword")] =
        rules.get("RULES_DEF").get(0);

q[N.indexOf("RULE")][T.indexOf("RuleKeyword")] =
        rules.get("RULE").get(0);

q[N.indexOf("RULES")][T.indexOf("RuleKeyword")] =
        rules.get("RULES").get(0);
q[N.indexOf("RULES")][T.indexOf(Term.EOF)] =
        rules.get("RULES").get(1);

q[N.indexOf("R")][T.indexOf("Nterm")] =
        rules.get("R").get(0);
q[N.indexOf("R")][T.indexOf("EpsKeyword")] =
        rules.get("R").get(0);
q[N.indexOf("R")][T.indexOf("Term")] =
        rules.get("R").get(0);

q[N.indexOf("R1")][T.indexOf("EpsKeyword")] =
        rules.get("R1").get(2);
q[N.indexOf("R1")][T.indexOf("Nterm")] =
        rules.get("R1").get(1);
q[N.indexOf("R1")][T.indexOf("Term")] =
        rules.get("R1").get(0);

q[N.indexOf("R2")][T.indexOf("RuleKeyword")] =
        rules.get("R2").get(1);
q[N.indexOf("R2")][T.indexOf("NewLine")] =
        rules.get("R2").get(0);
q[N.indexOf("R2")][T.indexOf(Term.EOF)] =
```

```
                rules.get("R2").get(1);

        q[N.indexOf("R3")][T.indexOf("NewLine")] =
                rules.get("R3").get(2);
        q[N.indexOf("R3")][T.indexOf(Term.EOF)] =
                rules.get("R3").get(2);
        q[N.indexOf("R3")][T.indexOf("RuleKeyword")] =
                rules.get("R3").get(2);
        q[N.indexOf("R3")][T.indexOf("Term")] =
                rules.get("R3").get(0);
        q[N.indexOf("R3")][T.indexOf("Nterm")] =
                rules.get("R3").get(1);

        return q;
    }
}
```

CalcMain.java

```
import calculator.*;
import lex_analyze.Scanner;
import syntax_analyze.Parser;

import java.io.File;

public class CalcMain {
    public static void main(String[] args) {
        String expr_src = args[1];

        Parser parser = ArithmeticStructure
                .getParser();
        Scanner scanner = new ArithmeticScanner
                (expr_src);
        parser.topDownParse(scanner);

        parser.addFile("output2" + File.separator +
                "expr_graph.dot");
        ArithmeticInterpreter evaluator =
                new ArithmeticInterpreter(parser.getParseTree());
        String expression = scanner.getText();
        System.out.println(expression.replaceAll("\n", "") +
                " = " + evaluator.getResult());

    }
}
```

ArithmeticInterpreter.java

```java
package calculator;

import lex_analyze.Token;
import syntax_analyze.ParseNode;
import syntax_analyze.ParseTree;

public class ArithmeticInterpreter {
    private int result;
    private final ParseTree tree;

    public ArithmeticInterpreter(ParseTree parseTree)
    {
        super();
        this.tree = parseTree;
        interpretTree();
    }

    public int getResult() {
        return result;
    }

    private void interpretTree() {
        result = scanE((ParseNode)tree.getRoot());
    }

    // E ::= T E1
    private int scanE(ParseNode root) {
        return
        scanT ((ParseNode)root.getChildAt(0)) +
                scanE1((ParseNode)root.getChildAt(1));
    }

    //E1 ::= '+' T E1 | eps
    private int scanE1(ParseNode node) {
        int res = 0;
        while (node.getChildCount() == 3) {
            res += scanT((ParseNode)node.getChildAt(1));
            node = (ParseNode)node.getChildAt(2);
        }
        return res;
    }

    //T ::= F T1
    private int scanT(ParseNode node) {
        return scanF ((ParseNode)node.getChildAt(0)) *
                scanT1((ParseNode)node.getChildAt(1));
```

```
    }

    //T1 ::= '*' F T1 | eps
    private int scanT1(ParseNode node) {
        int res = 1;
        while (node.getChildCount() == 3) {
            res *= scanF((ParseNode)node.getChildAt(1));
            node = (ParseNode)node.getChildAt(2);
        }
        return res;
    }

    //F ::= n | '(' E ')'
    private int scanF(ParseNode node) {
        if (node.getChildCount() == 3) {
            return scanE((ParseNode)node.getChildAt(1));
        } else {
            Token tok = (Token)node.getSymbolAt(0);
            return Integer.parseInt(tok.getImage());
        }
    }

}
```

ArithmeticScanner.java

```
package calculator;

import lex_analyze.Coords;
import lex_analyze.Scanner;
import lex_analyze.Token;
import syntax_analyze.symbols.Term;

import java.util.Map;

public class ArithmeticScanner extends Scanner {
    public ArithmeticScanner(String filepath) {
        super(ArithmeticStructure.staticRegExpressions(),
                filepath);
    }

    @Override
    public String setPattern() {
        StringBuilder res =
                new StringBuilder(makeGroup(BLANK,
                        blank_expr) + "|" + makeGroup
                        ("newline", "\\R") + "|" +
```

```
                    makeGroup("comments", "\\*[^\\*\\n]*\\n"));
        for (Map.Entry<String, String> e: regexp.entrySet()) {
            res.append("|").append(makeGroup(e.getKey(),
                    e.getValue()));
        }
        return res.toString();

    }
    @Override
    protected Token returnToken (String type) {
        Coords last = coord;
        coord = coord.shift(image.length());
        log.append(type).append(' ').append(last.toString())
                .append('-').append(coord.toString())
                .append(": <").append(image).append(">\n");
        if (image.matches("[0-9]+")){
            return new Token(type, image, last, coord);
        }
        return new Token(image, image, last, coord);
    }

    @Override
    public Token getNextToken() {
        if (coord.getPos() >= text.length()) {
            return new Token(Term.EOF, coord);
        }
        String image;
        if (m.find()) {
            if (m.start() != coord.getPos()) {
                log.append(String.format("SYNTAX ERROR: %d",
                        coord.getPos())).append(coord.toString())
                        .append('\n');
                System.out.println(String.format("SYNTAX ERROR: %d",
                        coord.getPos()) + coord.toString());
                System.exit(-1);
            }
            if ((image = m.group(BLANK)) != null) {
                coord = coord.shift(image.length());
                return getNextToken();
            }
            if ((image = m.group(NEWLINE)) != null) {
                coord = coord.newline();
                coord = coord.shift(image.length() - 1);
                return getNextToken();
            }
            for (String s: regexp.keySet()) {
```

```
                if (isType(s)) {
                    return returnToken(s);
                }
            }

            System.out.println("ERROR " + coord.toString() + " "
                    + text.substring(coord.getPos()));
            return getNextToken();

        } else {
            log.append("SYNTAX ERROR: ").append(coord.toString())
                    .append('\n');
            log.append("SYNTAX ERROR: ").append(coord.toString())
                    .append('\n');
            System.out.println("SYNTAX ERROR: " + coord.toString());
            return new Token(Term.EOF, coord);
        }
    }

    @Override
    public Token nextToken() {
        if (index <= tokens_list.size() - 1) {
            index++;
            return tokens_list.get(index - 1);
        } else {
            return new Token(Term.EOF, coord);
        }
    }
}
```

ArithmeticStructure.java

```
package calculator;

import syntax_analyze.Parser;
import syntax_analyze.rules.RHS;
import syntax_analyze.symbols.Nonterm;
import syntax_analyze.symbols.Term;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.LinkedHashMap;

public class ArithmeticStructure {
    public final static HashMap<String, String> regexp =
            staticRegExpressions();
```

```
public final static ArrayList<String> terms =
        staticTermList();
public final static ArrayList<String> nonterms =
        staticNontermList();
public final static Nonterm axiom = new Nonterm("E");
public final static RHS[][] q = staticDelta();

public static HashMap<String, String>
staticRegExpressions() {
    LinkedHashMap<String, String> exprs =
            new LinkedHashMap<>();
    exprs.put("n", "[0-9]+");
    exprs.put("PLUS", "\\+");
    exprs.put("STAR", "\\*");
    exprs.put("OPENBRACE", "\\(");
    exprs.put("CLOSEBRACE", "\\)");
    return exprs;
}

public static Parser getParser() {
    return new Parser(terms, nonterms, axiom, q);
}

private static ArrayList<String> staticNontermList() {
    return new ArrayList<>(Arrays.asList(
            "E", "E1", "T", "T1", "F"
    ));
}
private static ArrayList<String> staticTermList() {
    return new ArrayList<>(Arrays.asList(
            "+", "*", "(", ")", "n", "$"
    ));
}
private static RHS[][] staticDelta() {
    int m = nonterms.size();
    int n = terms.size();
    RHS[][] q = new RHS[m][n];
    for (RHS[] line: q) {
        Arrays.fill(line, RHS.ERROR);
    }
    q[0][2] = new RHS(
            new Nonterm("T"),
            new Nonterm("E1")
    );
    q[0][4] = new RHS(
```

```
                new Nonterm("T"),
                new Nonterm("E1")
        );
        q[1][0] = new RHS(
                new Term("+"),
                new Nonterm("T"),
                new Nonterm("E1")
        );
        q[1][3] = RHS.EPSILON;
        q[1][5] = RHS.EPSILON;
        q[2][2] = new RHS(
                new Nonterm("F"),
                new Nonterm("T1")
        );
        q[2][4] = new RHS(
                new Nonterm("F"),
                new Nonterm("T1")
        );
        q[3][0] = RHS.EPSILON;
        q[3][1] = new RHS(
                new Term("*"),
                new Nonterm("F"),
                new Nonterm("T1")
        );
        q[3][3] = RHS.EPSILON;
        q[3][5] = RHS.EPSILON;
        q[4][2] = new RHS(
                new Term("("),
                new Nonterm("E"),
                new Term(")")
        );
        q[4][4] = new RHS(
                new Term("n")
        );
        return q;
    }
}
```

Тестирование

SelfMain

Входные данные

grammar.txt

```
$AXIOM S
$NTERM NTERMS TERMS_DEF TERMS RULES_DEF RULE RULES R R1 R2 R3
$TERM "AxiomKeyword" "NTermKeyword" "TermKeyword" "RuleKeyword"
"EpsKeyword" "Nterm" "Term" "Equal" "NewLine"

* правила грамматики
$RULE S = "AxiomKeyword" "Nterm" "NTermKeyword" "Nterm" NTERMS
TERMS_DEF RULES_DEF
$RULE NTERMS = "Nterm" NTERMS
               $EPS
$RULE TERMS_DEF = "TermKeyword" "Term" TERMS
$RULE TERMS = "Term" TERMS
              $EPS
$RULE RULES_DEF = RULE RULES
$RULE RULES = RULE RULES
              $EPS
$RULE RULE = "RuleKeyword" "Nterm" "Equal" R
$RULE R = R1 R2
$RULE R1 = "Term" R3
           "Nterm" R3
           "EpsKeyword"
$RULE R3 = "Term" R3
           "Nterm" R3
            $EPS
$RULE R2 = "NewLine" R
            $EPS
```

Вывод на `stdout`

GrammarStructure.java

```
import syntax_analyze.rules.RHS;
import syntax_analyze.Parser;
import syntax_analyze.symbols.Nonterm;
import syntax_analyze.symbols.Term;

import java.util.ArrayList;
import java.util.Arrays;

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", "NTERMS", "TERMS_DEF", "TERMS", "RULES_DEF",
            "RULES", "RULE", "R", "R1", "R2", "R3"
    ));
}
private static ArrayList<String> staticTermList() {
    return new ArrayList<>(Arrays.asList(
            "AxiomKeyword", "Nterm", "Term", "NTermKeyword",
            "TermKeyword", "RuleKeyword", "EpsKeyword",
            "NewLine", "Equal", Term.EOF
    ));
}
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][0] = new RHS(
            new Term("AxiomKeyword"),
            new Term("Nterm"),
            new Term("NTermKeyword"),
            new Term("Nterm"),
            new Nonterm("NTERMS"),
            new Nonterm("TERMS_DEF"),
            new Nonterm("RULES_DEF")
            );
    q[1][1] = new RHS(
            new Term("Nterm"),
            new Nonterm("NTERMS")
            );
    q[1][4] = RHS.EPSILON;
    q[2][4] = new RHS(
            new Term("TermKeyword"),
            new Term("Term"),
            new Nonterm("TERMS")
            );
    q[3][5] = new RHS(
```

```
        new Nonterm("RULE"),
        new Nonterm("RULES")
        );
q[4][2] = new RHS(
        new Term("Term"),
        new Nonterm("TERMS")
        );
q[4][5] = RHS.EPSILON;
q[5][5] = new RHS(
        new Term("RuleKeyword"),
        new Term("Nterm"),
        new Term("Equal"),
        new Nonterm("R")
        );
q[6][5] = new RHS(
        new Nonterm("RULE"),
        new Nonterm("RULES")
        );
q[6][9] = RHS.EPSILON;
q[7][1] = new RHS(
        new Nonterm("R1"),
        new Nonterm("R2")
        );
q[7][2] = new RHS(
        new Nonterm("R1"),
        new Nonterm("R2")
        );
q[7][6] = new RHS(
        new Nonterm("R1"),
        new Nonterm("R2")
        );
q[8][1] = new RHS(
        new Term("Nterm"),
        new Nonterm("R3")
        );
q[8][2] = new RHS(
        new Term("Term"),
        new Nonterm("R3")
        );
q[8][6] = new RHS(
        new Term("EpsKeyword")
        );
q[9][5] = RHS.EPSILON;
q[9][7] = new RHS(
        new Term("NewLine"),
        new Nonterm("R")
```

```
                );
        q[9][9] = RHS.EPSILON;
        q[10][1] = new RHS(
                new Term("Nterm"),
                new Nonterm("V3")
                );
        q[10][2] = new RHS(
                new Term("Term"),
                new Nonterm("R3")
                );
        q[10][5] = RHS.EPSILON;
        q[10][7] = RHS.EPSILON;
        q[10][9] = RHS.EPSILON;
        return q;
    }
}
```

Входные данные

arithmetic.txt

```
$AXIOM E
$NTERM E' T T' F
$TERM "+" "*" "(" ")" "n"

* правила грамматики
$RULE E = T E'
$RULE E' = "+" T E'
            $EPS
$RULE T = F T'
$RULE T' = "*" F T'
            $EPS
$RULE F = "n"
          "(" E ")"
```

Вывод на `stdout`

```
import syntax_analyze.rules.RHS;
import syntax_analyze.Parser;
import syntax_analyze.symbols.Nonterm;
import syntax_analyze.symbols.Term;

import java.util.ArrayList;
import java.util.Arrays;

public class GrammarStructure {
    public final static ArrayList<String> terms =
    staticTermList();
```

```
public final static ArrayList<String> nonterms =
staticNontermList();
public final static Nonterm axiom = new Nonterm("E");
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(
            "E", "E'", "T", "T'", "F"
    ));
}
private static ArrayList<String> staticTermList() {
    return new ArrayList<>(Arrays.asList(
            "+", "*", "(", ")", "n", Term.EOF
    ));
}
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][2] = new RHS(
            new Nonterm("T"),
            new Nonterm("E'")
            );
    q[0][4] = new RHS(
            new Nonterm("T"),
            new Nonterm("E'")
            );
    q[1][0] = new RHS(
            new Term("+"),
            new Nonterm("T"),
            new Nonterm("E'")
            );
    q[1][3] = RHS.EPSILON;
    q[1][5] = RHS.EPSILON;
    q[2][2] = new RHS(
            new Nonterm("F"),
            new Nonterm("T'")
```

```
                );
        q[2][4] = new RHS(
                new Nonterm("F"),
                new Nonterm("T'")
                );
        q[3][0] = RHS.EPSILON;
        q[3][1] = new RHS(
                new Term("*"),
                new Nonterm("F"),
                new Nonterm("T'")
                );
        q[3][3] = RHS.EPSILON;
        q[3][5] = RHS.EPSILON;
        q[4][2] = new RHS(
                new Term("("),
                new Nonterm("E"),
                new Term(")")
                );
        q[4][4] = new RHS(
                new Term("n")
                );
        return q;
    }
}
```

CalcMain

Входные данные

expr.txt

```
5 * (3 + 7) + 6 * 7
```

Вывод на `stdout`

```
92
```

Вывод

В результате выполнения лабораторной работы был изучен алгоритм построения таблиц предсказывающего анализатора, разработан самоприменимый генератор компиляторов на основе предсказывающего анализа.