****

**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)»**

**(МГТУ им. Н. Э. Баумана)**

Отчёт по лабораторной работе № 3

по курсу

“Конструирование компиляторов”

Тема: “Синтаксический разбор с использованием метода рекурсивного спуска.”

Вариант 3.

Выполнил студент группы ИУ7-21М:

Осипов Олег Константинович.

Преподаватель:

Ступников А.А.

Москва 2020 г.

**Синтаксический разбор с использованием метода рекурсивного спуска.**

**1. Основные понятия и соглашения.**

Здесь и далее используются все соглашения, присутствующие в предыдущих лабораторных работах [3], а также те, что приняты в нижележащем тексте.

Одним из наиболее простых и потому одним из наиболее популярных методов нисходящего синтаксического анализа является *метод рекурсивного спуска (recursive descent method).* Метод основан на “зашивании”

правил грамматики непосредственно в управляющие конструкции распознавателя. Синтаксические анализаторы, работающие по методу рекурсивного спуска без возврата, могут быть

построены для класса грамматик, называющегося LL(1). Первая буква L в названии связана с тем, что входная цепочка читается слева направо, вторая буква L означает, что строится левый вывод входной цепочки, 1

означает, что на каждом шаге для принятия решения используется один символ непрочитанной части входной

цепочки. Для строгого определения LL(1) грамматики потребуются две функции - FIRST и FOLLOW.

**Определение.** Пусть Для

контекстно-свободной грамматики , определена функция:

Иначе говоря, множество состоит из всех терминальных префиксов длины *k* (или меньше, если из выводится терминальная цепочка длины, меньшей *k*) терминальных цепочек, выводимых из . По

определению полагают, что

**Определение.** Пусть Для

контекстно-свободной грамматики , определена функция:

Иначе говоря, множество состоит из всех цепочек длины *k* (или меньше) терминальных цепочек, которые могут встречаться непосредственно справа от в каких-нибудь цепочках, выводимых из аксиомы,

причем если выводимая цепочка, то тоже принадлежит.

Для грамматики LL(1) k=1, и имеет смысл говорить только о функциях а вместо фразы «терминальных цепочек» говорить «терминальных символов». Далее, везде, где k не указано, полагаем k = 1 для функций . Кроме того, для стартового символа грамматики S, добавим в множество маркер конца входного потока, выраженный символом $.

Ниже даны алгоритмы вычисления множеств FIRST и FOLLOW, при k = 1.

**Алгоритм 1.1.** Вычисление FIRST.

Вход: символы грамматики

Выход: Семейство множеств FIRST(X).

Метод:

(1)

(2)

(3)

(4)

(5)

(6)

(7)

(8)

(9)

(10)

(11)

(12)

(13)

(14)

(15)

(16)

(17) );

(18)

(19)

(20)

**Алгоритм 1.2.** Вычисление FOLLOW.

Вход: Символы грамматики

Выход: Семейство множеств FOLLOW(A).

Метод:

(1)

(2)

(3)

(4)

(5)

(6)

(7)

(8)

(9)

Пример 1.3.

Для КС-грамматики

Заметим, что нетерминал E начинается с T, а T с F. Так как нетерминал F не порождает пустых строк (нет правил вида ), и он начинается с терминалов ( и id, то всё, что попало в F кроме , попадёт в T, затем, так как там в множестве нет в множестве, то Т уже вычислен, и всё, что попало в T кроме , попадёт в Е. Так как в нет , то множество окончательно построено.

И нетерминал *E’* и нетерминал *T’* имеют правило вида Поэтому будет лежать в обоих множествах. Заметим, что остальные продукции для этих нетерминалов начинаются с терминального символа. Поскольку k = 1, то кроме этих символов (+, \*) ничего больше не добавится.

. Так, как E – стартовое правило грамматики, то добавим символ конца входного потока $. Поскольку E встречается только в одном правиле F -> ( E ), и справа от него стоит скобка, то добавим скобку в множество. Так как FIRST( **)** ) не содержит , то ничего больше мы не добавляем.

Так, как Е’ является крайним справа нетерминалом в продукции E -> T E’, то всё, что содержит множество , добавится к В продукции E’ -> T E’ E’ как бы добавляет сам себя, но поскольку значением функции является множество, то никаких новых элементов добавлено не будет. Так как больше E’ нигде не встречается то вычисления заканчиваются.

Рассмотрим продукцию Так как нетерминал T стоит справа от E’, то добавим все элементы, кроме Так, как будут добавлены. В продукции , ничего нового добавлено не будет, так как за символом следует всё тот же E’, а FOLLOW(E’) = FOLLOW(E). Что касается нетерминала T’, то он является самым крайним, т.е. правым нетерминалом для правил T и T’. Поэтому все элементы из FOLLOW(T) добавятся в FOLLOW(T’).

Справа от нетерминала F стоит нетерминал T’. Поэтому, все элементы из FIRST(T’) добавляются в FOLLOW(F). Поскольку T’ порождает пустые строки, то все элементы из множеств FOLLOW(T) и FOLLOW(T’) добавляются в FOLLOW(F). Нигде больше F не встречается, поэтому ничего больше не будет добавлено.

**Теорема 1.4.** КС-грамматика *G* = (*T*, *N*, *P*, *S*) является LL(1)-грамматикой тогда и только тогда, когда для каждых двух различных правил *A* → β и *A* → γ выполняется условие:

*,* для любого нетерминала

Иначе говоря, для каждых двух различных правил *A* → β и *A* → γ их множества представляют из себя непересекающиеся множества (т.е. две или более альтернативы правила *А* не имеют общий префикс). И если не пересекаются. Аналогично, если

не пересекаются.

Пример 1.5.

Грамматика *G1,* c правилами:

не является LL(1)-грамматикой.

Чтобы понять почему так, отметим, что FOLLOW(S) = FOLLOW(S’) = { e, $ }. Поскольку одна из альтернатив S’ порождает пустую строку, то вторая альтернатива не должна начинаться с символа из множества FOLLOW(S’). Однако для продукции Т.е. пересечение множеств , что не удовлетворяет теореме 1.4.

Отметим также, что грамматика G2 с правилами:

не является также LL(1)-грамматикой, по причине наличии пересекающихся множеств FIRST для двух или более альтернатив правила S. (FIRST(i E t S) = FIRST(i E t S e S) = { I }, т.е. имеется общий префикс для двух альтернатив).

Опишем алгоритм построение таблицы синтаксического анализа для предиктивного нисходящего анализатора.

**Алгоритм 1.6.** Построение таблицы синтаксического анализа для грамматики LL(1).

Вход: Множества FIRST(X), FOLLOW(X) для

Выход: таблица синтаксического анализа M.

Метод:

(1)

(2)

(3)

(4)

(5)

Таблица синтаксического анализа для LL(1)-грамматики представляет из себя таблицу, где в качестве строк используются нетерминальные символы грамматики, а в качестве столбцов – терминальные символы в купе с символом $, кроме символа пустой строки .

Ниже дана схема предиктивного синтаксического анализатора, который использует стэк явно, а не стэк вызовов процедур, который образуется при рекурсивном спуске.

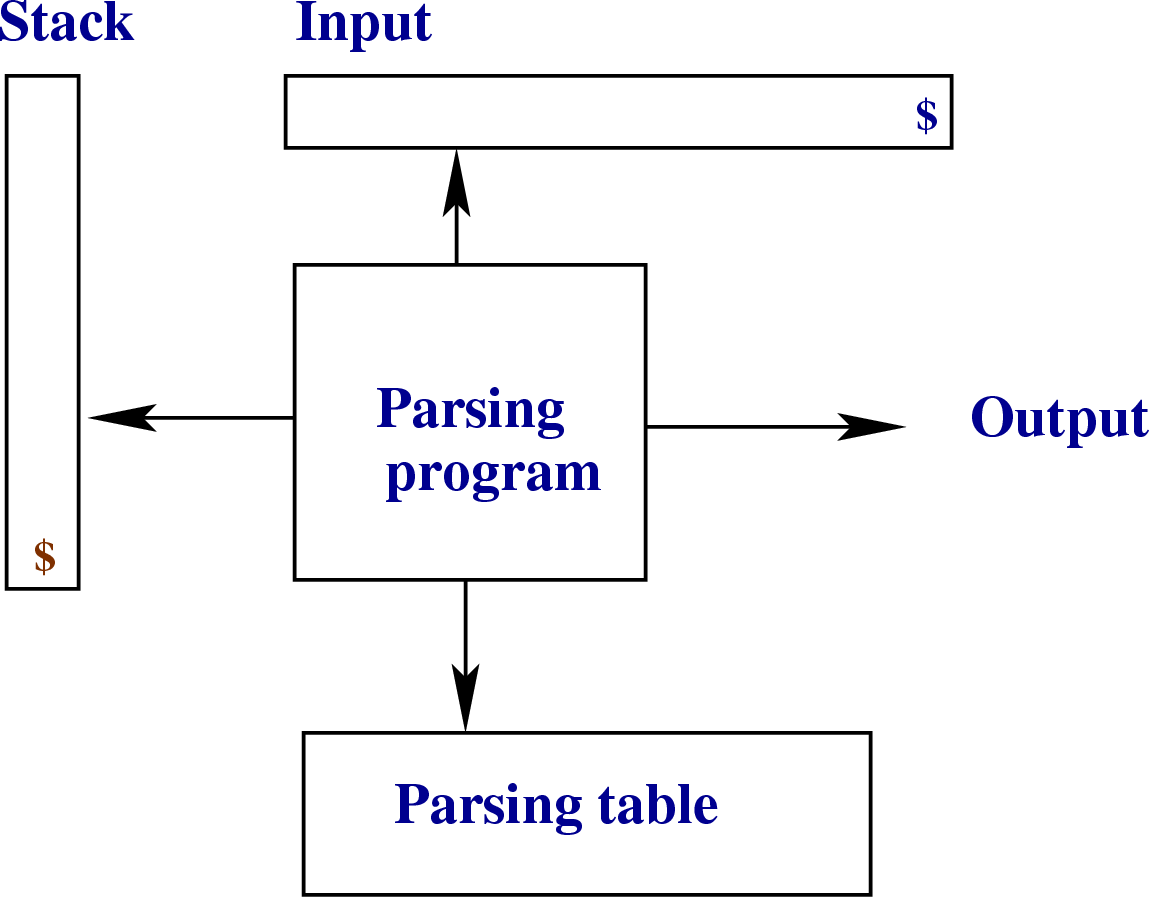


Рис. 1. Схема нисходящего синтаксического анализатора.

Программа читает по одному токену из входного потока (Input). В начале работы анализатора, в стеке (Stack) содержатся стартовый символ грамматики S и символ конца входного потока $, причём первый находится на вершине стэка, а головка указывает на первый символ входного потока. Чтение входа происходит слева-направо. Если символ на вершине стэка совпадает с символом входного потока, то символ удаляется из стэка, а анализатор читает следующий символ входного потока. Иначе, в таблице Parsing Table, полученной алгоритмом 1.6, происходит поиск соответствующий записи. Если такой записи нет, то происходит синтаксическая ошибка. Такое возможно, когда на вершине стэка оказывается терминальный символ, который не совпадает с текущим символом входного потока (т.е. невозможно обратиться к таблице), либо, когда в ячейке, указываемой символом на вершине стэка и входным символом ничего нет. Если запись была обнаружена, то в стэк добавляются все грамматические символы продукции, начиная справа-налево. Таким образом, на вершине стэка будет самый первый грамматический символ продукции. Перед добавлением, соответственно, со стэка снимается текущий нетерминал. Всё выше сказанное обобщает следующий алгоритм.

**Алгоритм 1.7.** Нисходящий разбор.

Вход: таблица синтаксического анализа М, полученная алгоритмом 1.6,

для грамматики *G*, а также входная строка *w*.

Выход: Если иначе сообщение об ошибке.

Метод:

**2. Постановка задачи.**

Рассматривается грамматика *G\_3* выражений отношения со следующим множеством правил:

| **realNum**

Согласно соглашениям из предыдущей лабораторной работы, стартовым символом грамматики является *E*, нетерминалами являются E, AE, T, F, PE, PLUSOP, MULLOP, RELOP. Терминалами являются: +, -, \*, /, %, < <=, >, >=, ==, <>, (, ), **num**, **id**. **Num** и **id**, представляют из себя отдельные лексические единицы (лексемы), определённые следующим образом:

**id =** [A-Za-z\_][A-Za-z\_0-9]+

**num =** [0-9]+

**realnum =** [0-9]+.[0-9]+(**(**(E | e)(- | )[0-9]+) | ).

Все выше представленные шаблоны представляют из себя регулярные выражения. Идентификатор (id) – есть строка, начинающаяся с символа английского алфавита либо подчёркивания, за которым идёт любое количество букв из английского алфавита, цифр, или знаков подчёркивания.

Число (num) – это непустая последовательность цифр от 0 до 9. Действительное число с плавающей точкой (realNum), это число, с дробной частью, отделяемой точкой (.), за которой следует непустая цепочка цифр, которая может оканчиваться необязательной экспоненциальной частью, причём число, следуемое за символом экспоненты может иметь необязательный минус (быть отрицательным).

Данная грамматика также дополняется дополнительными правилами. Кроме того, в ней изменён стартовый символ, теперь этот символ равен *P*.

В новой грамматике появились четыре новых терминала (; , { , } , =), и четыре новых нетерминала (P,B,OPLIST,OP).

Для дополненной грамматики *G\_3m* необходимо разработать программу нисходящего синтаксического анализа.

**3. Реализация.**

**3.1. Разработка лексического анализатора.**

Прежде чем начать разрабатывать синтаксический анализатор для данной грамматики, необходимо написать программу, которая бы отображала входные символы файла в терминальные символы грамматики. Поскольку во входном файле могут быть символы (последовательности символов), которые не представляют из себя лексемы (они не соответствуют ни одному шаблону для терминалов грамматики), то в таких случаях, анализатор должен вернуть специальный токен, обозначающий сигнал о лексической ошибке.

Тип токен представлен ниже. Он имеет следующие поля: *lexem, type, name*. Поле *lexem* представляет из себя исходную лексему (последовательность символов во входном файле). Поле *type* обозначает тип лексемы (терминал, нетерминал, ошибка) для синтаксического анализатора. Поле *name* хранит в себе имя грамматического символа, т.е.

**package** ru.osipov.labs.lab3.lexers;  
  
**public class** Token {  
 **private** String **name**;  
 **private** String **lexem**;  
 **private char type**;  
  
 **public** Token(String name, String lexem,**char** type){  
 **this**.**type** = type;  
 **this**.**name** = name;  
 **this**.**lexem** = lexem;  
 }  
  
 **public void** setType(**char** type) {  
 **this**.**type** = type;  
 }  
  
 **public void** setLexem(String lexem) {  
 **this**.**lexem** = lexem;  
 }  
  
 **public void** setName(String name) {  
 **this**.**name** = name;  
 }  
  
 **public char** getType() {  
 **return type**;  
 }  
  
 **public** String getLexem() {  
 **return lexem**;  
 }  
  
 **public** String getName() {  
 **return name**;  
 }  
  
 @Override  
 **public** String toString(){  
 **if**(**type** == **'t'**)  
 **return lexem** != **null** ? **lexem** : **name**;  
 **else if**(**type** == **'n'**)  
 **return name**;  
 **else  
 return name**+**lexem**;  
 }  
}

Листинг 1. Класс Token.

Для операций ввода, вывода (чтения, записи) определён интерфейс ILexerIO.

**package** ru.osipov.labs.lab3.lexers;  
  
**import** java.io.IOException;  
**import** java.io.InputStream;  
  
**public interface** LexerIO {  
 **int** ungetch(**char** c);  
 **int** getch(InputStream r) **throws** IOException;  
 **int** getFilech(InputStream r) **throws** IOException;  
 **void** clear();  
 **int** getLine();  
 **int** getCol();  
 **void** setLine(**int** l);  
 **void** setCol(**int** c);  
 String getFromBuffer();  
}

Листинг 2. Интерфейс ILexerIO.

Его реализует следующий класс LookAheadBufferedLexer:

**package** ru.osipov.labs.lab3.lexers;  
  
**import** java.io.IOException;  
**import** java.io.InputStream;  
  
**public class** LookAheadBufferedLexer **implements** LexerIO{  
 *//Lexer parts.* **protected char**[] **buf**;  
 **protected int bsize**;  
 **protected int bufp**;  
 **protected int EOL**;  
  
 **protected int line**;  
 **protected int col**;  
  
 **public** LookAheadBufferedLexer(**int** bsize){  
 **this**.**bsize** = bsize;  
 **this**.**buf** = **new char**[bsize + 1];  
 **this**.**bufp** = 0;  
 **this**.**EOL** = 0;  
 **this**.**line** = 1;  
 **this**.**col** = 0;  
 }  
  
 **public** LookAheadBufferedLexer(){  
 **this**(255);  
 }  
  
 @Override  
 **public int** getLine(){  
 **return line**;  
 }  
  
 @Override  
 **public void** setLine(**int** l){  
 **this**.**line** = l;  
 }  
  
 @Override  
 **public void** setCol(**int** c){  
 **this**.**col** = c;  
 }  
  
 @Override  
 **public int** getCol(){  
 **return col**;  
 }  
  
 @Override  
 **public int** ungetch(**char** c){  
 **if**(**bufp** > **bsize**){  
 System.***out***.println(**"Error ("**+**line**+**":"**+**col**+**"). ungetch: too many characters."**);  
 **return** 0;  
 }  
 **else**{  
 **buf**[**bufp**++] = c;  
 **return** 1;  
 }  
 }  
  
 @Override  
 **public int** getch(InputStream r) **throws** IOException {  
 **if**(**bufp** > 0)  
 **return buf**[--**bufp**];  
 **else**{  
 **col**++;  
 **char** c = (**char**)r.read();  
 **if**(c == **'\n'**){  
 **line**++; **col** = 0;  
 }  
 **return** c;  
 }  
 }  
  
 @Override  
 **public int** getFilech(InputStream r) **throws** IOException {  
 **char** c = (**char**)r.read();  
 **if**(c == **'\n'**){  
 **line** += 1;  
 **col** = 0;  
 }  
 **else**{  
 **col** += 1;  
 }  
 **return** c;  
 }  
  
 @Override  
 **public** String getFromBuffer(){  
 **return new** String(**buf**);  
 }  
  
 @Override  
 **public void** clear(){  
 **EOL** = 0;  
 **bufp** = 0;  
 **this**.**buf** = **null**;  
 **this**.**buf** = **new char**[**bsize**];  
 }  
}

Листинг 3. Реализация интерфейса ILexerIO, тип LookAheadBufferedLexer.

Сам лексический анализатор опирается на модель детерминированного конечного автомата (ДКА). Данный автомат получается следующим образом:

1) Для каждого шаблона строится недетерминированный конечный автомат (НКА). Для построения используется алгоритм “конструктор Томпсона”.

2) Создаётся новая вершина, которая становится новым начальным состоянием нового НКА. От неё проведено ребро с меткой символа пустой строки к каждому начальному состоянию НКА, построенного на предыдущем шаге для определённого шаблона.

3) Полученный НКА является *комбинированным*, т.е. он содержит несколько допускающих состояний (являющимися конечными состояниями НКА, построенных на основе шаблонов). Данный автомат представлен следующим типом:

**package** ru.osipov.labs.lab1.structures.automats;  
  
**import** ru.osipov.labs.lab1.structures.graphs.Vertex;  
  
**import** java.util.HashSet;  
**import** java.util.List;  
**import** java.util.Set;  
  
*//NFA that contains more than one final state.***public class** CNFA **extends** NFA{  
 **private** HashSet<Vertex> **comboF**;  
 **public** CNFA(){  
 **this**.**comboF** = **new** HashSet<>();  
 }  
  
 **public boolean** hasMultipleFinish(){  
 **return comboF**.size() > 1;  
 }  
  
 **public void** setComboStart(Vertex v){  
 addNode(v);  
 v.setStart(**true**);  
 **this**.**start** = v;  
 **for**(Vertex vd : **nodes**){  
 *//System.out.println(vd+" : "+vd.isFinish());* **if**(vd.isFinish()) {  
 **comboF**.add(vd);  
 **break**;  
 }  
 }  
 }  
  
 **public void** addToFinished(Vertex v){  
 **if**(v.isFinish())  
 **comboF**.add(v);  
 }  
  
 **public void** setFinished(List<Vertex> v){  
 **comboF**.addAll(v);  
 }  
  
 **public** Set<Vertex> getFinished(){  
 **return comboF**;  
 }  
  
 **public void** setFinish(Vertex v){  
 **this**.**finish** = v;  
 **comboF**.add(v);  
 }  
}

Листинг 4. Класс комбинированного НКА - CNFA.

Данный автомат преобразуется в детерминированный конечный автомат (ДКА). Преобразование осуществляет реализация алгоритма 3.20 [2]. Код данного алгоритма представлен ниже, и находится в классе, представляющего детерминированный конечный автомат DFA из предыдущих лабораторных работ [4].

**public** DFA(CNFA nfa){  
 **this**.**finished** = **new** HashSet<>();  
 **this**.**tranTable** = **new** HashMap<>();  
 LinkedStack<Set<Vertex>> ST = **new** LinkedStack<>();  
 HashSet<Vertex> s0 = **new** HashSet<>();  
 s0.add(nfa.getStart());  
  
 ST.push(NFA.*e\_closure*(s0));*//e-closure(s0) where s0 - start state.* **this**.**start** = **new** Vertex();*//label for e-closure(s0)* **this**.**start**.setStart(**true**);  
 **this**.**alpha** = nfa.getAlpha();  
 **this**.**start**.setName(**"1"**);  
 Vertex a = **this**.**start**;  
 Vertex b = **null**;  
 HashMap<Set<Vertex>,Vertex> mapped = **new** HashMap<>();*//1 iteration map s0 to a single state.* mapped.put(ST.top(),a);  
 **int** qn = 1;  
 **while**(!ST.isEmpty()){  
 Set<Vertex> T = ST.top();  
 ST.pop();  
 a = mapped.get(T);*//get mapped state. (after 1 iteration)* **for**(**char** c : **alpha**){  
 Set<Vertex> U = NFA.*e\_closure*(NFA.*move*(T,c));  
 **if**(!mapped.containsKey(U)){  
 ST.push(U);  
 qn++;  
 b = **new** Vertex();  
 b.setName(qn+**""**);  
 mapped.put(U,b);*//add new unmapped state.* }  
 **else** b = mapped.get(U);*//get mapped state.* Set<Vertex> FF = **new** HashSet<>(U);  
 FF.retainAll(nfa.getFinished());  
**if**(FF.size() > 0) {*//U.contains(nfa.getFinish())* b.setValue(FF.stream().findFirst().get().getValue());**finished**.add(b);  
 }  
 FF = **new** HashSet<>(T);  
 FF.retainAll(nfa.getFinished());  
 **if**(FF.size() > 0) {  
 a.setValue(FF.stream().findFirst().get().getValue());  
 **finished**.add(a);  
 }  
 Edge tran = **new** Edge(a,b,c);*//Implies tran\_function.* **this**.**edges**.add(tran);  
 **tranTable**.put(**new** Pair<>(a,c),b);  
 }  
 }  
 addNode(**this**.**start**);*//build graph* **for**(Vertex vv: **finished**){  
 vv.setFinish(**true**);  
 }  
 *//initiate dead\_state.* **for**(Vertex v : **nodes**){  
 **int** c = 0;  
 List<Pair<Vertex,Character>> k = **tranTable**.keySet().stream().filter(x -> x.getV1().equals(v)).collect(Collectors.*toList*());  
 **for**(Pair<Vertex,Character> kk : k){  
 **if**(**tranTable**.get(kk).equals(v))  
 c++;  
 }*//if any transition leads to the same vertex.* **if**(c == **alpha**.size()) {  
 v.setDead(**true**);  
 **this**.**dead** = v;  
 **break**;  
 }  
 }  
 System.***out***.println(**"States of DFA: "**+**this**.**nodes**.size());  
 System.***out***.println(**"Finished: "**+**this**.**finished**.size());  
 **if**(**dead** != **null**)  
 System.***out***.println(**"Dead: "**+**dead**);  
}

Листинг 5. Процедура преобразования комбинированного НКА в ДКА, представленная в виде конструктора класса DFA.

Каждое допускающее состояние ДКА содержит в поле *value,* имя распознанного терминального символа. Код класса вершины (Vertex) представлен ниже:

**package** ru.osipov.labs.lab1.structures.graphs;  
  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
**public class** Vertex {  
 **private** String **name**;  
 **private** List<Edge> **edges**;  
 **private boolean isStart**;  
 **private boolean isFinish**;  
 **private boolean isDead**;  
 **private** String **value**;*//label inside* **public** Vertex(){  
 **this**.**name** = **""**;  
 **this**.**isStart** = **false**;  
 **this**.**isFinish** = **false**;  
 **this**.**isDead** = **false**;  
 **this**.**edges** = **new** ArrayList<>();  
 **this**.**value** = **""**;  
 }  
  
 **public** Vertex(String name){  
 **this**.**name** = name;  
 **this**.**isStart** = **false**;  
 **this**.**isFinish** = **false**;  
 **this**.**isDead** = **false**;  
 **this**.**edges** = **new** ArrayList<>();  
 **this**.**value** = **""**;  
 }  
  
 **public void** setStart(**boolean** e){  
 **this**.**isStart** = e;  
 }  
  
 **public void** setFinish(**boolean** e) {  
 **this**.**isFinish** = e;  
 }  
  
 **public void** setName(String n){  
 **this**.**name** = n;  
 }  
  
 **public void** setDead(**boolean** dead) {  
 **isDead** = dead;  
 }  
  
 **public void** setEdges(List<Edge> edges) {  
 **this**.**edges** = edges;  
 }  
  
 **public void** addEdge(Edge e){  
 **if**(!**edges**.contains(e))  
 **edges**.add(e);  
 }  
  
 **public** List<Edge> getEdges() {  
 **return edges**;  
 }  
  
 **public boolean** isFinish() {  
 **return isFinish**;  
 }  
  
 **public boolean** isStart() {  
 **return isStart**;  
 }  
  
 **public** String getName(){  
 **return name**;  
 }  
  
 @Override  
 **public boolean** equals(Object obj){  
 **if**(obj **instanceof** Vertex) {  
 Vertex b = (Vertex) obj;  
 **return isStart** == b.**isStart** && **isFinish** == b.**isFinish** && **name**.equals(b.**name**);  
 }  
 **return false**;  
 }  
  
 @Override  
 **public int** hashCode(){  
 **return this**.getName().hashCode();  
 }  
  
 **public void** setValue(String v){  
 **this**.**value** = v;  
 }  
  
 **public** String getValue(){  
 **return value**;  
 }  
  
 **public boolean** isDead() {  
 **return isDead**;  
 }  
  
 @Override  
 **public** String toString(){  
 **return this**.**name**;  
 }  
}

Листинг 6. Класс вершины Vertex автомата.

4) Для минимизации состояний полученного ДКА используется алгоритм Хопкрофта [5], модифицированный следующим образом.

*//CREATE min-DFA based on DFA (minimize specified dfa)***public** DFA(DFA dfa,**boolean** isLexer) {  
 **this**.**alpha** = dfa.getAlpha();  
 Set<Vertex> F = dfa.getFinished();  
 Set<Vertex> NF = dfa.getNodes().stream().filter(vertex -> !F.contains(vertex)).collect(Collectors.*toSet*());*//Q - F.* ArrayList<Set<Vertex>> P = minimize(NF, F, dfa.getNodes(), dfa.getTranTable(),isLexer);  
 **this**.**tranTable** = **new** HashMap<Pair<Vertex, Character>, Vertex>();  
 HashMap<Pair<Vertex,Character>,Vertex> oldTran = dfa.getTranTable();  
 **this**.**dead** = **null**;  
 **this**.**start** = **null**;  
 **this**.**finished** = **new** HashSet<>();  
  
 Elem<Integer> nstatescount = **new** Elem<>(0);  
 HashMap<String,Vertex> mapped = **new** HashMap<>();  
 HashMap<String,Set<Vertex>> representers = **new** HashMap<>();  
 **for** (Set<Vertex> sv : P) {*//for each set -> make new state.* **for** (Vertex v : sv) {*//select a representer of the set  
 //System.out.println("Rep: "+v.getName());* Vertex n = **new** Vertex();  
 nstatescount.setV1(nstatescount.getV1() + 1);  
 makeRecord(P,oldTran,sv,v,n,nstatescount,mapped);  
 **break**;  
 }  
 }  
 addNode(**this**.**start**);  
}

*//CREATE min-DFA based on DFA (minimize specified dfa)***public** DFA(DFA dfa) {  
 **this**(dfa,**false**);  
}

*//isLexer :: ifTrue => build initial partition with patterns, dead state and others. ({p1},{p2}...{pn}{dead}{NF states})***private** ArrayList<Set<Vertex>> minimize(Set<Vertex> NF, Set<Vertex> F,List<Vertex> Q,HashMap<Pair<Vertex,Character>,Vertex> table,**boolean** isLexer){  
 ArrayList<Set<Vertex>> P = **new** ArrayList<>(Q.size());  
 HashMap<String,Integer> clz = **new** HashMap<>();*//indicies of class which state is belonging.* HashMap<Integer,Set<Vertex>> involved = **new** HashMap<>();*//classes which have states with edges to splitter.* LinkedQueue<Pair<Set<Vertex>,Character>> queue = **new** LinkedQueue<>();  
 ArrayList<Set<Vertex>> W = **new** ArrayList<>();  
 **if**(isLexer) {  
 List<Vertex> l = ColUtils.*fromSet*(F);  
 Vertex dead = **null**;  
 **for**(**int** i = 0; i < l.size();i++){  
 **if**(l.get(i).isDead()){  
 dead = l.get(i);  
 **break**;  
 }  
 }  
 **if**(dead == **null**){  
 **for**(Vertex v : NF){  
 **if**(v.isDead()){  
 dead = v;  
 **break**;  
 }  
 }  
 NF.remove(dead);  
 }  
 l.remove(dead);  
 LinkedStack<String> ids = **new** LinkedStack<>();  
 HashMap<String,Set<Vertex>> ig = **new** HashMap<>();  
 **for**(**int** i = 0; i < l.size();i++){  
 String gname = l.get(i).getValue();  
 **if**(!ids.contains(gname)){  
 ids.push(gname);  
 ig.put(gname,**new** HashSet<>());  
 }  
 }  
 **for**(**int** i = 0; i < l.size();i++){  
 Set<Vertex> group = ig.get(l.get(i).getValue());  
 group.add(l.get(i));  
 }  
 P.add(NF);  
 **for**(String k : ig.keySet()){  
 Set<Vertex> group = ig.get(k);  
 P.add(group);  
 }  
 **if**(dead != **null**) {  
 Set<Vertex> Dead = **new** HashSet<>();  
 Dead.add(dead);  
 P.add(Dead);  
 }  
 System.***out***.println(**"IP = "**+P);  
 W.addAll(P);  
 }  
 **else** {  
 P.add(F);*//F* P.add(NF);*//Q - F (Q\F)* W.add(F);  
 W.add(NF);  
 }  
 **while**(W.size() > 0){  
 Set<Vertex> C = W.get(0);  
 W.remove(C);  
 **for**(Character c : **alpha**){  
 List<Pair<Vertex,Character>> keys = table.keySet().stream().filter(x -> x.getV2() == c).collect(Collectors.*toList*());  
 Set<Vertex> X = **new** HashSet<>();  
 **for**(Pair<Vertex,Character> k : keys) {  
 **if**(C.contains(table.get(k))){  
 X.add(k.getV1());  
 }  
 }  
 **int** k = 0;  
 **while**(k < P.size()) {  
 Set<Vertex> Y = P.get(k);  
 Set<Vertex> Y\_i = **new** HashSet<>(Y);  
 Set<Vertex> X\_i = **new** HashSet<>(X);  
 Y\_i.removeAll(X);*//Y - X.* X\_i.retainAll(Y);*// X INTERSECT Y* **if** (X\_i.size() > 0 && Y\_i.size() > 0){  
 P.remove(Y);  
 P.add(X\_i);*//replaced current* P.add(Y\_i);  
 **if**(W.contains(Y)){  
 W.remove(Y);  
 W.add(X\_i);  
 W.add(Y\_i);  
 }  
 **else**{  
 **if**(X\_i.size() <= Y\_i.size()){  
 W.add(X\_i);  
 }  
 **else** W.add(Y\_i);  
 }  
 }  
 k++;  
 }  
 }  
 }

System.***out***.println(**"P = "**+P);  
 **return** P;  
}

Листинг 7. Алгоритм Хопкрофта для минимизации состояний в ДКА с опцией учёта шаблонов для терминалов грамматики.

Конструктор ДКА, принимающий на вход НКА переопределён следующим образом. Если кроме НКА ничего больше не подаётся на вход, то вызывается более общий конструктор ДКА, но уже со значением параметра *isLexer*, равным **false**. Данный параметр передается в модифицированную процедуру *minimize*, которая получает множество подразбиений одинаковых состояний ДКА. Параметр указывает процедуре, как построить начальное подразбиение. Если его значение равно **false**, то построить начальное подразбиение, разделив состояние ДКА на две группы: допускающие и не допускающие. Иначе, если *isLexer* равен **true**, то тупиковое состояние, а также допускающие состояния, которые содержат различные имена терминальных символов считаются в виде отдельных групп. Тупиковое состояние предварительно отыскивается во множествах допускающих и не допускающих состояний ДКА, и если оно найдено, то оно удаляется из этих множеств и добавляется в виде группы к начальному подразбиению. Во множестве допускающих состояний, каждое состояние, хранящее уникальное имя терминального символа, составляет отдельную группу. Такие группы добавляются в начальное подразбиение. Множество же недопускающих состояний добавляется также, как и в начале, в виде отдельной группы. Далее работа метода ничем не отличается от старой версии из [4]. Работа процедуры *makeRecord*, это составить новую таблицу переходов, с учётом полученного от метода *minimize* разбиения.

Полученный ДКА с минимальным количеством состояний также может содержать тупиковое состояние, т.е. состояние, которое не является допускающим и содержит только переходы в само себя. Это состояние можно спокойно отбросить, включив проверку на наличие записи в таблице переходов ДКА. Далее представлен код метода, удаляющий тупиковое состояние ДКА.

*//Delete dead state from DFA.***public void** deleteDeadState(){  
 **if**(**dead** != **null**){  
 List<Edge> conns = **new** ArrayList<>(**dead**.getEdges());  
 **this**.**nodes**.remove(**dead**);*//remove from nodes.* **for**(Edge e : conns){  
 Pair<Vertex,Character> p1 = **new** Pair<>(e.getSource(),e.getTag());  
 Pair<Vertex,Character> p2 = **new** Pair<>(e.getTarget(),e.getTag());  
 **if**(**tranTable**.get(p1) != **null**)*//remove from tranTable.* **tranTable**.remove(p1);  
 **if**(**tranTable**.get(p2) != **null**)  
 **tranTable**.remove(p2);  
 **this**.**edges**.remove(e);*//remove from edges.* e.disconnectNodes();  
 }  
 **this**.**dead** = **null**;  
 }  
}

Листинг 8. Процедура удаления тупикового состояния ДКА.

Наконец, для повышения эффективности работы был переопределён метод *addNode*, добавляющий переданную вершину *v*, а также всё что с ней связано (рёбра, вершины, которые соединены ребром c *v*, а также остальные смежные вершины и их рёбра и так далее).

*//Add nodes to the graph. All edges has already been added before building DFA.  
//(On step constructing DFA from NFA) or  
//(On step minimization DFA)  
//param: Vertex v is useless (used only to override method)*@Override  
**public void** addNode(Vertex v){  
 **for**(Pair<Vertex,Character> k : **tranTable**.keySet()){  
 Vertex dest = **tranTable**.get(k);  
 **if**(!**nodes**.contains(dest))  
 **this**.**nodes**.add(dest);  
 **if**(!**nodes**.contains(k.getV1()))  
 **this**.**nodes**.add(k.getV1());  
 }  
}

Листинг 9. Добавление вершин и рёбер во множества V,E графа, представляющий ДКА, на основе записей в таблице переходов

Вышеуказанный метод, вместо использования стэка, использует таблицу переходов, которая была получена в ходе процедуры детерминизации или минимизации.

Теперь, можно описать сам лексический анализатор. Центральное место в нём занимает метод *recognize*, принимающий в качестве параметра экземпляр класса InputStream, представляющий абстракцию входного потока. Данный метод описан в интерфейсе *ILexer*.

**package** ru.osipov.labs.lab3.lexers;  
  
**import** java.io.IOException;  
**import** java.io.InputStream;  
  
**public interface** ILexer {  
 Token recognize(InputStream f) **throws** IOException;  
}

Листинг 10. Интерфейс ILexer.

Метод *recognize* работает следующим образом. До тех пор, пока идут пробельные символы, происходит чтение следующего символа файла. Как только встречается не пробельный символ, то сохранить в стэке начальное состояние, и занести в экземпляр класса *StringBuilder*, представляющий динамически расширяемую строку, первый такой символ. Далее процедура *moveTo* принимает текущее состояние и входной символ, и либо возвращает новое состояние, либо возвращает **null**, если нет соответствующей записи в таблице переходов. Если было получено состояние, то до тех пор, пока не будет получено тупиковое состояние, или пока процедура *moveTo* не вернёт **null,** читать соответствующий файл посимвольно и сохранить результат в экземпляре класса StringBuilder. Если дальше нельзя осуществлять переходы, т.е. *moveTo* вернула **null** или тупиковое состояние, то пока не будет получено допускающее состояние, удалять справа налево символы из StringBuilder и сохранять их в буфер входного потока с помощью процедуры *ungetch*, добавляя их слева направо. В качестве указателя текущей свободной позиции в буфере используется переменная экземпляра класса *LookAheadBufferedLexer* *bufp*. Как только будет достигнуто допускающее состояние, то будет возвращён токен, представляющий терминал, имя которого и хранит допускающее состояние в поле *value*. Метод *getch,* который используется для чтения входного потока, проверяет не пуст ли буфер, и если он не пуст, то читать с этого буфера справа налево.

**package** ru.osipov.labs.lab3.lexers;  
  
**import** ru.osipov.labs.lab1.structures.automats.CNFA;  
**import** ru.osipov.labs.lab1.structures.automats.DFA;  
**import** ru.osipov.labs.lab1.structures.automats.NFA;  
**import** ru.osipov.labs.lab1.structures.graphs.Pair;  
**import** ru.osipov.labs.lab1.structures.graphs.Vertex;  
**import** ru.osipov.labs.lab1.structures.lists.LinkedStack;  
  
**import** java.io.IOException;  
**import** java.io.InputStream;  
**import** java.util.List;  
**import** java.util.stream.Collectors;  
  
**public class** DFALexer **extends** DFA **implements** ILexer {  
  
 **private** LexerIO **io**;  
  
 **public** DFALexer(DFA dfa, LexerIO io){  
 **super**(dfa,**true**);  
 **this**.deleteDeadState();  
 **this**.**io** = io;  
 }  
  
 **public** DFALexer(NFA nfa, LexerIO io){  
 **super**(nfa);  
 **this**.deleteDeadState();  
 **this**.**io** = io;  
 }  
  
 **public** DFALexer(CNFA nfa){  
 **super**(nfa);  
 **this**.deleteDeadState();  
 System.***out***.println(**"DFA States: "**+**this**.getNodes().size());  
 System.***out***.println(**"Patterns (F): "**+**this**.getFinished().size());  
 **this**.**io** = **new** LookAheadBufferedLexer();  
 }  
  
 **public** DFALexer(DFA dfa){  
 **super**(dfa,**true**);*//set Minimization for lexer true.* **this**.deleteDeadState();  
 System.***out***.println(**"MinDFA States: "**+**this**.getNodes().size());  
 System.***out***.println(**"Patterns (F): "**+**this**.getFinished().size());  
 **this**.**io** = **new** LookAheadBufferedLexer();  
 }  
  
 @Override  
 **public** Token recognize(InputStream f) **throws** IOException {  
 **char** cur = (**char**)**io**.getch(f);  
 **while**(cur == **' '** || cur == **'\t'** || cur == **'\n'** || cur == **'\r'**) {  
 **if** (cur == **'\n'**) {  
 **io**.setLine(**io**.getLine() + 1);  
 **io**.setCol(0);  
 }  
 cur = (**char**) **io**.getch(f);  
 }  
 **if**(((**int**)cur) == 65535)  
 **return new** Token(**"$"**,**"$"**,**'t'**);  
 Vertex s = **this**.**start**;  
 LinkedStack<Vertex> states = **new** LinkedStack<>();  
 states.push(s);  
 StringBuilder sb = **new** StringBuilder();  
 sb.append(cur);  
 **while**(((**int**)cur) != 65535 && !s.isDead()){*//while not EOF or not deadState.* s = moveTo(s,cur);  
 **if**(s == **null** || s.isDead()){  
 **while**(s == **null** || !s.isFinish()){

**if**(sb.length() == 0)  
 **return new** Token(**"Unrecognized"**,**"Error at ("**+**io**.getLine()+**":"**+**io**.getCol()+**") :: Unrecognized token: "**+sb.toString()+**"\n"**,**'e'**);  
  
 cur = sb.charAt(sb.length() - 1);  
 sb.deleteCharAt(sb.length() - 1);  
 **io**.ungetch(cur);  
 s = states.top();  
 states.pop();  
 }  
 **return new** Token(s.getValue(),sb.toString(),**'t'**);  
 }  
 **else** {  
 states.push(s);  
 cur = (**char**)**io**.getch(f);  
 sb.append(cur);  
 }  
 }  
 **if**(sb.length() > 0){  
 **if**((**int**)cur == 65535)  
 sb.deleteCharAt(sb.length() - 1);*//remove redundant read EOF ch.* **if**(s.isFinish())  
 **return new** Token(s.getValue(),sb.toString(),**'t'**);  
 **else  
 return new** Token(**"Unrecognized"**,**"Error at ("**+**io**.getLine()+**":"**+**io**.getCol()+**") :: Unrecognized token: "**+sb.toString()+**"\n"**,**'e'**);  
 }  
 **else  
 return new** Token(**"$"**,**"$"**,**'t'**);  
 }  
  
 **private** Vertex moveTo(Vertex v, **char** c){  
 List<Pair<Vertex,Character>> k = **tranTable**.keySet().stream().filter(x -> x.getV1().equals(v) && x.getV2() == c).collect(Collectors.*toList*());  
 **return** k.size() > 0 ? **tranTable**.get(k.get(0)) : **null**;  
 }

**public** Token generateError(String s1, String s2){  
 **return new** Token(**"Unrecognized"**,**"Error at ("**+**io**.getLine()+**":"**+**io**.getCol()+**") :: “ + ”Expected token: "**+s1+**" but actual: "**+s2+**"\n"**,**'e'**);  
}

}

Листинг 11. Класс DFALexer, представляющий лексический анализатор.

Для создания комбинированного НКА используется класс FALexerGenerator:

**package** ru.osipov.labs.lab3.lexers.generators;  
  
**import** org.omg.Messaging.SYNC\_WITH\_TRANSPORT;  
**import** ru.osipov.labs.lab1.Main;  
**import** ru.osipov.labs.lab1.structures.automats.CNFA;  
**import** ru.osipov.labs.lab1.structures.automats.DFA;  
**import** ru.osipov.labs.lab1.structures.automats.NFA;  
**import** ru.osipov.labs.lab1.structures.graphs.Edge;  
**import** ru.osipov.labs.lab1.structures.graphs.Elem;  
**import** ru.osipov.labs.lab1.structures.graphs.Vertex;  
**import** ru.osipov.labs.lab1.structures.lists.LinkedStack;  
**import** ru.osipov.labs.lab1.utils.RegexRPNParser;  
**import** ru.osipov.labs.lab2.grammars.Grammar;  
  
**import** java.util.\*;  
  
**public class** FALexerGenerator {  
 **public** CNFA buildNFA(Grammar G){  
 Map<String, List<String>> rules = G.getLexicalRules();  
 HashSet<Character> alpha = **new** HashSet<>();  
 RegexRPNParser parser = **new** RegexRPNParser();  
 Vertex vs = **new** Vertex(**"1"**);

vs.setStart(**true**);  
 **int** nfa\_i = 2;  
 Elem<Integer> idC = **new** Elem<>(1);  
 Map<String,String> ids\_p = **new** HashMap<>();  
 List<Vertex> Fs = **new** ArrayList<>();  
 **for**(String id : rules.keySet()){  
 List<String> patterns = rules.get(id);  
 **for**(String pattern : patterns){  
 LinkedStack<Character> rpn = **new** LinkedStack<>();  
 **if**(pattern.length() == 1) {  
 **char** c = pattern.charAt(0);  
 CNFA nfa = **new** CNFA();  
 Vertex v1 = **new** Vertex(nfa\_i+**""**);  
 nfa\_i++;  
 Vertex v2 = **new** Vertex(nfa\_i+**""**);  
 nfa\_i++;  
 Edge e = **new** Edge(v1,v2,c);  
 v1.setStart(**true**);  
 v2.setFinish(**true**);  
 nfa.setComboStart(v1);  
 nfa.setFinish(v2);  
 nfa.getFinish().setValue(id);  
 Fs.add(nfa.getFinish());  
 alpha.add(c);  
 idC.setV1(nfa\_i);  
 Edge e2 = **new** Edge(vs,nfa.getStart(),(**char**)1);

**continue**;  
 }  
 **else** {  
 String p\_i = **new** String(pattern.toCharArray());  
 p\_i = p\_i.replaceAll(G.getEmpty(),(**char**)1+**""**);

parser.setTerminals(p\_i.toCharArray());  
 p\_i = Main.*addConcat*(p\_i,parser);  
 parser.setTerminals(p\_i.toCharArray());rpn = parser.GetInput(p\_i);  
}  
 CNFA nfa = Main.*buildNFA*(rpn,parser,idC,Fs);  
 alpha.addAll(nfa.getAlpha());  
 nfa.getFinish().setValue(id);Fs.addAll(nfa.getFinished());  
 nfa\_i = idC.getV1();  
 nfa.getStart().setName(nfa\_i+**""**);  
 nfa\_i++;  
 idC.setV1(nfa\_i);Edge e = **new** Edge(vs,nfa.getStart(),(**char**)1);}  
 }  
 CNFA comboNFA = **new** CNFA();  
 comboNFA.setComboStart(vs);  
 comboNFA.setFinished(Fs);alpha.remove((**char**)1);  
 System.***out***.println(**"Alpha: "**+alpha);  
 comboNFA.setAlpha(alpha);  
 System.***out***.println(**"States of NFA: "**+comboNFA.getCountOfStates());  
 **return** comboNFA;  
 }  
}

Листинг 12. Создание комбинированного НКА.

Процедура buildNFA из класса Main, лежащий в пакете ru.osipov.labs.lab1, строит НКА, сохраняя количество полученных состояний, так как для создания комбинированного НКА, каждой вершине автомата НКА, распознающего определённый шаблон, установлено уникальное имя.

*//Algorithm: Mac Naughton-Yamada-Tompson (Мак-Нотона, Ямады, Томпсона)***public static** CNFA buildNFA(LinkedStack<Character> expr, RegexRPNParser parser, Elem<Integer> el,List<Vertex> Fe){  
 LinkedStack<CNFA> result = **new** LinkedStack<>();  
 HashSet<Character> alpha = **new** HashSet<>();  
 **int** c = el.getV1();  
 **for**(Character tok : expr){  
 **if**(parser.isUnaryOp(tok)){  
 CNFA g = result.top();  
 result.pop();  
 **for**(Vertex v: g.getNodes()){*//nullify finish* v.setFinish(**false**);  
 }  
 Vertex s = **new** Vertex(c+**""**);  
 c++;  
 Vertex t = **new** Vertex(c+**""**);  
 c++;  
 Edge iloop = **new** Edge(g.getFinish(), g.getStart(), (**char**) 1);  
 g.getFinish().setFinish(**false**);  
 g.getStart().setStart(**false**);  
 Edge se = **new** Edge(s, g.getStart(), (**char**) 1);  
 Edge fe = **new** Edge(g.getFinish(), t, (**char**) 1);  
 **if**(tok == **'\*'**) {*// '+' and '\*' differ only with one edge.* Edge loop = **new** Edge(s, t, (**char**) 1);*//for '\*' add empty from start to finish* }  
 CNFA R = **new** CNFA();  
 t.setFinish(**true**);  
 R.setStart(s);  
 R.setFinish(t);  
 result.push(R);  
 }  
 **else if**(parser.isOperator(tok)){  
 CNFA g2 = result.top();  
 result.pop();  
 CNFA g1 = result.top();  
 result.pop();  
 **for**(Vertex v: g2.getNodes()){*//nullify finish* v.setFinish(**false**);  
 }  
 **for**(Vertex v: g1.getNodes()){  
 v.setFinish(**false**);  
 }  
 **if**(tok == **'^'**) {  
 Vertex inter = g1.getFinish();  
 List<Edge> outE = g2.getStart().getEdges().stream().filter(edge -> edge.getSource().equals(g2.getStart())).collect(Collectors.*toList*());  
 List<Edge> outEi = g2.getStart().getEdges().stream().filter(edge -> edge.getTarget().equals(g2.getStart())).collect(Collectors.*toList*());  
 inter.setFinish(**false**);  
 **for**(Edge e: outE){*//union output edges.* Edge ae = **new** Edge(inter,e.getTarget(),e.getTag());  
 g2.disconnectVertexByEdge(e,g2.getStart(),e.getTarget());  
 }  
 **for**(Edge e : outEi){*//union input edges.* Edge ea = **new** Edge(e.getSource(),inter,e.getTag());  
 g2.disconnectVertexByEdge(e,e.getSource(),g2.getStart());  
 }  
 CNFA FC = **new** CNFA();  
 g2.getFinish().setFinish(**true**);  
 FC.setStart(g1.getStart());  
 FC.setFinish(g2.getFinish());  
 result.push(FC);  
 }  
 **else if**(tok == **'|'**){  
 Vertex s = **new** Vertex(c+**""**);  
 c++;  
 Vertex t = **new** Vertex(c+**""**);  
 c++;  
 Vertex s1 = g1.getStart();  
 Vertex s2 = g2.getStart();  
 Vertex t1 = g1.getFinish();  
 Vertex t2 = g2.getFinish();  
 Edge s\_s1 = **new** Edge(s,s1,(**char**)1);  
 Edge s\_s2 = **new** Edge(s,s2,(**char**)1);  
 Edge t\_t1 = **new** Edge(t1,t,(**char**)1);  
 Edge t\_t2 = **new** Edge(t2,t,(**char**)1);  
 s.setStart(**true**);  
 s1.setStart(**false**);  
 s2.setStart(**false**);  
 t1.setFinish(**false**);  
 t2.setFinish(**false**);  
 t.setFinish(**true**);  
 CNFA FU = **new** CNFA();  
 FU.setStart(s);  
 FU.setFinish(t);  
 result.push(FU);  
 }  
 }  
 **else**{*//token is not operator.* Vertex v1 = **new** Vertex(c+**""**);  
 c++;  
 Vertex v2 = **new** Vertex(c+**""**);  
 c++;  
 v2.setFinish(**true**);  
 Edge e = **new** Edge(v1,v2,tok);  
 CNFA F = **new** CNFA();  
 F.setComboStart(v1);  
 F.setFinish(v2);  
 alpha.add(tok);  
 result.push(F);  
 }  
 }  
 result.top().setAlpha(alpha);  
 el.setV1(c);  
 **return** result.top();  
}

Листинг 13. Метод построения комбинированного НКА.

Для использования сокращённой записи объединений многих регулярных выражений в этом же классе модифицирован метод addConcat.

**public static** String addConcat(String s,RegexRPNParser parser){  
 StringBuilder result = **new** StringBuilder();  
 **for**(**int** i = 0; i < s.length(); i++){  
 **if**(s.charAt(i) == **'['**){*// replace class [A-Z] with expression (A|B|...|Z) and add '^' if needed.* **char** t = s.charAt(i);  
 **if**(i > 0 && s.charAt(i - 1) != **'('**){  
 result.append(**'^'**);  
 }  
 result.append(**'('**);  
 **int** j = i + 1;  
 **boolean** wflag = **false**;  
 **while**(t != **']'** && j < s.length()){  
 t = s.charAt(j);  
 **if**(j + 1 < s.length() && s.charAt(j) == **'-'**){  
 **char** a = s.charAt(j - 1);  
 **char** b = s.charAt(j + 1);  
 **if**(a > b){  
 **char** temp = a;  
 a = b;  
 b = temp;  
 }  
 **while**(a != b){  
 a++;  
 result.append(a).append(**"|"**);  
 }  
 CharSequence subor = result.subSequence(0,result.length() - 1);  
 result = **new** StringBuilder().append(subor);  
 j = j + 2;  
 wflag = **true**;  
 **continue**;  
 }  
 **if**(wflag && t != **']'**){  
 result.append(**"|"**);  
 wflag = **false**;  
 }  
 String or = (j + 1 == s.length() || s.charAt(j + 1) == **']'**) ? s.charAt(j)+**""** : s.charAt(j)+**"|"**;  
 result.append(s.charAt(j) == **']'** ? **")"** : or);  
 **if**(s.charAt(j) == **']'**){  
 j++;  
 **break**;  
 }  
 **if**(s.charAt(j + 1) == **']'**){  
 result.append(**')'**);  
 j++;  
 j++;  
 **break**;  
 }  
 j++;  
 }  
 i = j;  
 **if**(i == s.length())  
 **return** result.toString();  
 **if**((s.charAt(i) == **')'** || s.charAt(i) == **'\*'** || s.charAt(i) == **'+'**)){  
 result.append(s.charAt(i));  
 **if**(i + 1 < s.length() && s.charAt(i + 1) != **')'** && s.charAt(i + 1) != **'+'** && s.charAt(i + 1) != **'\*'** && s.charAt(i + 1) != **'|'** && s.charAt(i + 1) != **'['**)  
 result.append(**'^'**);  
 }  
 **else if**(parser.isTerminal(s.charAt(i)) || s.charAt(i) == **'('**){  
 result.append(**'^'**).append(s.charAt(i));  
 }  
 **else if**(s.charAt(i) == **'['**)  
 i = j - 1;  
 **continue**;  
 }  
 result.append(s.charAt(i));  
 **if**(parser.isTerminal(s.charAt(i)) && i + 1 < s.length() && (parser.isTerminal(s.charAt(i + 1)) || s.charAt(i + 1) == **'('** ) ){  
 result.append(**'^'**);  
 }  
 **if**((s.charAt(i) == **')'** || s.charAt(i) == **'\*'** || s.charAt(i) == **'+'** ) && i + 1 < s.length() && (parser.isTerminal(s.charAt(i + 1)) || s.charAt(i + 1) == **'('**) ){  
 result.append(**'^'**);  
 }  
 }  
 **return** result.toString();  
}

Листинг 14. Процедура addConcat, раскрывающая сокращения вида [A-Za-z] в (A|…|z), а также расставляющая оператор конкатенации для строк вида ab => a^b, или (a|b)(c|d) => (a|b)^(c|d).

**3.2. Разработка синтаксического анализатора.**

Для создания таблицы синтаксического анализа LL-(1) грамматики, необходимо вычислить функции FIRST и FOLLOW. В классе LLParserGenerator определена процедура firstTable, возвращающая таблицу значений функции FIRST для одиночного грамматического символа (терминала или нетерминала).

**package** ru.osipov.labs.lab3.parsers.generators;  
  
**import** ru.osipov.labs.lab1.structures.graphs.Pair;  
**import** ru.osipov.labs.lab1.structures.lists.LinkedStack;  
**import** ru.osipov.labs.lab1.structures.observable.ObservableHashSet;  
**import** ru.osipov.labs.lab1.utils.ColUtils;  
**import** ru.osipov.labs.lab2.grammars.Grammar;  
**import** ru.osipov.labs.lab2.grammars.GrammarString;  
**import** ru.osipov.labs.lab2.grammars.GrammarSymbol;  
  
**import** java.util.\*;  
  
**public class** LLParserGenerator {

*//Compute FIRST for each terminal and non-terminal.***public** Map<String,Set<String>> firstTable(Grammar G){  
 Set<String> T = G.getTerminals();  
 Set<String> NT = G.getNonTerminals();  
 HashMap<String,Set<String>> res = **new** HashMap<String,Set<String>>();  
 LinkedStack<String> S = **new** LinkedStack<>();  
 **for**(String t : T){ *//for each terminal put first(t) = t.* HashSet<String> f = **new** HashSet<>();  
 f.add(t);  
 res.put(t,f);  
 }  
 **for**(String n : NT){  
 S.push(n);  
 }  
 S.push(G.getStart());  
 M1: **while**(!S.isEmpty()){*//compute first for each nonTerminal p.* Set<String> first\_i = **new** HashSet<>();  
 String p = S.top();  
 S.pop();  
 **if**(res.containsKey(p))*//computed.* **continue**;  
 Set<GrammarString> bodies = G.getProductions().get(p);  
 **for**(GrammarString str : bodies){  
 **int** l = str.getSymbols().size();  
 **int** ec = 0;  
 **boolean** addAll = **false**;  
 **for**(GrammarSymbol s : str.getSymbols()){  
 **if**(s.getType() == **'t'**){  
 **if**(s.getVal().equals(G.getEmpty()))  
 ec++;*//continue scanning string.* **else** {  
 first\_i.add(s.getVal());  
 **break**;*//end scanning after first terminal.* }  
 }  
 **else**{  
 **if**(res.containsKey(s.getVal())){  
 **if**(res.get(s.getVal()).contains(G.getEmpty())) {  
 Set<String> without\_empty = res.get(s.getVal());  
 without\_empty.remove(G.getEmpty());  
 first\_i.addAll(without\_empty);  
 ec++;*//continue scanning string.* **continue**;  
 }  
 first\_i.addAll(res.get(s.getVal()));*//add all symbols from first(X\_i) where X\_i is grammar symbol of str.* **break**;  
 }  
 addAll = **true**;  
 **break**;  
 }  
 }  
 **if**(addAll){  
 S.push(p);  
  
 List<GrammarSymbol> ns = str.getSymbols();  
 **for**(**int** i = ns.size() - 1; i >= 0; i--){  
 **if**(ns.get(i).getType() == **'n'**)  
 S.push(ns.get(i).getVal());  
 }  
 **continue** M1;  
 }  
 **else**{  
 **if**(ec == l)*//add empty if and only if all grammar symbols X in str contains eps in FIRST(X)* first\_i.add(G.getEmpty());  
 }  
 }  
 res.put(p,first\_i);  
 }  
 **return** res;  
}

…// Другие методы класса.

….

}

Листинг 15. Вычисление значений функции FIRST(X) для одиночного грамматического символа X.

Получив данную таблицу, легко написать функцию first, вычисляющее множество FIRST, для произвольной строки грамматических символов. Ниже дан код функции first:

*//Compute FIRST for GrammarString str. (list of GrammarSymbols)***private** Set<String> first(GrammarString str, Map<String,Set<String>> firstTable,String eps){  
 Set<String> res = **new** HashSet<>();  
 **int** ec = 0;  
 **for**(GrammarSymbol s : str.getSymbols()){  
 HashSet<String> ans = **new** HashSet<String>(firstTable.get(s.getVal()));  
 **if**(!ans.contains(eps)) {  
 res.addAll(ans);  
 **break**;  
 }  
 **else**{  
 ans.remove(eps);  
 res.addAll(ans);*//add all symbols except eps and continue.* ec++;  
 }  
 }  
 **if**(ec == str.getSymbols().size())  
 res.add(eps);  
 **return** res;  
}

Листинг 16. Функция FIRST(X1…Xn) для произвольной строки грамматических символов.

Для каждого нетерминала грамматики необходимо вычислить значение функции FOLLOW. Метод *followTable* выполняет данную задачу, реализуя алгоритм 1.2.

**public** Map<String,Set<String>> followTable(Grammar G,Map<String,Set<String>> firstTable){  
 List<String> NT = ColUtils.*fromSet*(G.getNonTerminals());  
 String empty = G.getEmpty();  
 HashMap<String,Set<String>> res = **new** HashMap<String,Set<String>>();  
 LinkedStack<String> S = **new** LinkedStack<>();  
 **for**(String N : NT) {  
 S.push(N);  
 res.put(N,**new** ObservableHashSet<String>());  
 }  
 res.get(G.getStart()).add(**"$"**);*//add $ to FOLLOW(S) where S = start symbol of G.* **while**(!S.isEmpty()){  
 String p = S.top();  
 S.pop();  
 Set<GrammarString> bodies = G.getProductions().get(p);  
 **for**(GrammarString str : bodies){  
 List<GrammarSymbol> l = str.getSymbols();  
 **for**(**int** i = 0; i < l.size()-1;i++){  
 GrammarSymbol sym = l.get(i);  
 **if**(sym.getType() != **'t'**){  
 GrammarString subStr = **new** GrammarString(**new** ArrayList<>(l.subList(i+1,l.size())));  
 Set<String> first = first(subStr,firstTable,empty);  
 **if**(first.contains(empty)) {  
 ((ObservableHashSet<String>) res.get(p)).attach((ObservableHashSet<String>) res.get(l.get(i).getVal()));  
 first.addAll(res.get(p));  
 }  
 first.remove(empty);  
*// if(sym.getVal().equals(G.getStart()))  
// first.add("$");* res.get(l.get(i).getVal()).addAll(first);  
 *//res.put(l.get(i).getVal(),first);* }  
 }  
 GrammarSymbol last = l.get(l.size() - 1);  
 **if**(last.getType() != **'t'**){  
 Set<String> first = res.get(p);  
 ((ObservableHashSet<String>) first).attach((ObservableHashSet<String>) res.get(last.getVal()));  
 *//res.put(last.getVal(),first);* }  
 }  
 }  
 **return** res;  
 }

Листинг 17. Функция FOLLOW(A) для любого нетерминала A грамматики G.

Для реализации механизма наследования в случае, когда все элементы из множества FOLLOW(A) для нетерминала А, представляющего заголовок продукции, идут во FOLLOW(B), где нетерминал В – либо является самым крайним правым символом продукции, либо множество first(Xk…Xn), символов грамматики, стоящих справа от B, содержит символ пустой строки, используется шаблон издатель – подписчик. Здесь, подписчиками и издателями являются сами множества FOLLOW. При добавлении элементов из FOLLOW(A) в FOLLOW(B), множество А подписывает подписчика, являющегося множеством В. Кроме того, множество А добавляет все элементы во множество В. В дальнейшем, когда во множество А добавятся новые элементы, эти элементы также добавятся во все множества, которые *подписались* на А. Здесь А – издатель, а В – подписчик. В данном случае В необходимо получать уведомления о добавлении новых элементов во множество А, и в такой ситуации, получить их от А. (В шагах листинга 17 ничего не удаляется из множеств, поэтому А уведомляет все множества подписчиков только при добавлении новых элементов).

С этой целью был создан класс ObservableHashSet, предоставляющий методы регистрации подписчика. Он хранит всех своих подписчиков в виде списка *observers*, элементы которого представлены тем же типом. Ниже дан код данного класса.

**package** ru.osipov.labs.lab1.structures.observable;  
  
**import** java.util.ArrayList;  
**import** java.util.Collection;  
**import** java.util.HashSet;  
**import** java.util.List;  
  
**public class** ObservableHashSet<T> **extends** HashSet<T> {  
  
 **private** List<ObservableHashSet<T>> **observers**;  
  
 **public** ObservableHashSet(){  
 **this**.**observers** = **new** ArrayList<>();  
 }  
  
 **public void** attach(ObservableHashSet<T> o) {  
 **observers**.add(o);  
 o.addAll(**this**);  
 }  
  
 **public void** detach(ObservableHashSet<T> o) {  
 **observers**.remove(o);  
 }  
  
 **public void** sendMessageAll() {  
 **for**(ObservableHashSet<T> o : **observers**){  
 o.addAll(**this**);  
 }  
 }  
  
 **public void** sendMessage(T ndata){  
 **for**(ObservableHashSet<T> o : **observers**){  
 o.add(ndata);  
 }  
 }  
  
 **public void** addWithoutNotify(T t){  
 **super**.add(t);  
 }  
  
 **public void** addAllWithoutNotify(Collection<? **extends** T> c){  
 **super**.addAll(c);  
 }  
  
 @Override  
 **public boolean** add(T t) {  
 **boolean** flag = **super**.add(t);  
 **if**(flag)  
 sendMessage(t);  
 **return** flag;  
 }  
  
 @Override  
 **public boolean** addAll(Collection<? **extends** T> c) {  
 **boolean** flag = **super**.addAll(c);  
 **if**(flag)  
 sendMessageAll();  
 **return** flag;  
 }  
}

Листинг 18. Класс ObseravbleHashSet.

Построив множества FIRST, FOLLOW, можно построить таблицу синтаксического анализа. Метод *getTable* экземпляра класса LLParserGenerator реализует алгоритм 1.6. Он возвращает таблицу синтаксического анализа в виде отображения. Отображение преобразует пару (нетерминал, терминал) в строку грамматических символов (т.е. в продукцию).

**public** Map<Pair<String,String>, GrammarString> getTable(Grammar G){  
 HashMap<Pair<String,String>, GrammarString> table = **new** HashMap<>();  
  
 Map<String,Set<String>> firstTable = firstTable(G);  
 Map<String,Set<String>> followTable = followTable(G,firstTable);  
 Set<String> ps = G.getProductions().keySet();  
 String empty = G.getEmpty();  
 **for**(String p : ps){  
 Set<GrammarString> bodies = G.getProductions().get(p);  
 **for**(GrammarString b: bodies){  
 Set<String> first = first(b,firstTable,empty);  
 **if**(first.contains(empty)) {  
 Set<String> follow = followTable.get(p);  
 **for** (String f : follow) {  
 Pair<String, String> rec = **new** Pair<>();  
 rec.setV1(p);  
 rec.setV2(f);  
 table.put(rec,b);*// new GrammarString(new ArrayList<>(b.getSymbols())));* }  
 }  
 **else  
 for**(String a : first){  
 Pair<String, String> rec = **new** Pair<>();  
 rec.setV1(p);  
 rec.setV2(a);  
 table.put(rec, b);*//new GrammarString(new ArrayList<>(b.getSymbols())));* }  
 }  
 }  
 **return** table;  
}

Листинг 19. Генерация таблицы синтаксического анализа для LL(1)-грамматики.

Экземпляр класса LLParser строит дерево разбора для файла, содержащее цепочку грамматических символов, принадлежащую языку, порождаемый грамматикой L(G). Если есть лексические или синтаксические ошибки, то данный синтаксический анализатор не сможет построить дерево разбора.

Полученное дерево разбора является левым порождением грамматики G.

**package** ru.osipov.labs.lab3.parsers;  
  
**import** org.omg.Messaging.SYNC\_WITH\_TRANSPORT;  
**import** ru.osipov.labs.lab1.structures.automats.DFA;  
**import** ru.osipov.labs.lab1.structures.graphs.Pair;  
**import** ru.osipov.labs.lab1.structures.lists.LinkedStack;  
**import** ru.osipov.labs.lab2.grammars.Grammar;  
**import** ru.osipov.labs.lab2.grammars.GrammarString;  
**import** ru.osipov.labs.lab2.grammars.GrammarSymbol;  
**import** ru.osipov.labs.lab3.lexers.DFALexer;  
**import** ru.osipov.labs.lab3.lexers.ILexer;  
**import** ru.osipov.labs.lab3.lexers.Token;  
**import** ru.osipov.labs.lab3.parsers.generators.LLParserGenerator;  
**import** ru.osipov.labs.lab3.trees.LinkedNode;  
**import** ru.osipov.labs.lab3.trees.LinkedTree;  
  
**import** java.io.\*;  
**import** java.util.ArrayList;  
**import** java.util.List;  
**import** java.util.Map;  
  
  
**public class** LLParser {  
 **private** LLParserGenerator **gen**;  
 **private** Map<Pair<String,String>, GrammarString> **table**;  
 **private** ILexer **lexer**;

**public** LLParser(Grammar G, ILexer lexer){  
 **gen** = **new** LLParserGenerator();  
 **this**.**table** = **gen**.getTable(G);  
 **this**.**lexer** = lexer;

}

*//Algorithm 4.20 with lexer module.***public** LinkedTree<Token> parse(Grammar G, String fname){  
 FileInputStream f;  
 **boolean** isParsed = **true**;  
 **try** {  
 f = **new** FileInputStream(**new** File(fname).getAbsolutePath());  
 InputStreamReader ch = **new** InputStreamReader(f);  
 LinkedStack<LinkedNode<Token>> S = **new** LinkedStack<>();  
 LinkedNode<Token> root = **new** LinkedNode<>();  
 LinkedNode<Token> EOF = **new** LinkedNode<>();  
 EOF.setValue(**new** Token(**"$"**,**"$"**,**'t'**));  
 root.setValue(**new** Token(G.getStart(),G.getStart(),**'n'**));  
 root.setIdx(1);  
 S.push(EOF);  
 S.push(root);*// Stack: S,$.* LinkedNode<Token> X = S.top();  
 Token tok = **lexer**.recognize(f);  
 String t = tok.getName();  
 String empty = G.getEmpty();  
 **int** nidx = 1;  
 **while**(!X.getValue().getName().equals(**"$"**)) {  
 **if**(X.getValue().getName().equals(t)){  
 S.top().getValue().setLexem(tok.getLexem());  
 S.pop();  
 tok = **lexer**.recognize(f);  
 *//System.out.println(tok);* t = tok.getName();  
 X = S.top();  
 **continue**;  
 }  
 **else if**(X.getValue().getName().equals(**"Unrecognized"**)){  
 System.***out***.println(X.getValue().getLexem());  
 isParsed = **false**;  
 **break**;  
 }  
 **else if**(G.getTerminals().contains(X.getValue().getName())) {  
 isParsed = **false**;  
 **break**;  
 }  
 GrammarString prod = **table**.get(**new** Pair<String,String>(X.getValue().getName(),t));  
 **if**(prod == **null**) {  
 isParsed = **false**;  
 **break**;  
 }  
 **else**{  
 List<GrammarSymbol> symbols = prod.getSymbols();  
 S.pop();  
 *//LinkedStack<LinkedNode<String>> RS = new LinkedStack<>();//used only to order children YK..Y1 -> Y1..Yk in brace notation* **for**(**int** i = symbols.size() - 1; i >= 0; i--){  
 LinkedNode<Token> node = **new** LinkedNode<>();  
 nidx++;  
 node.setValue(**new** Token(symbols.get(i).getVal(),**null**,symbols.get(i).getType()));  
 node.setIdx(nidx);  
 node.setParent(X);  
 X.getChildren().add(node);  
 **if**(!node.getValue().getName().equals(empty))  
 S.push(node);  
 }  
 }  
 X = S.top();  
 }  
 **if**(isParsed)  
 **return new** LinkedTree<Token>(root);  
 **else** {  
 **if**(tok.getType() != **'e'**) {  
 **lexer**.generateError(S.top().getValue().getName(),tok.getLexem());  
 }  
 **else** System.***out***.println(tok);  
 }  
 **return null**;*//if syntax error return null.* }  
 **catch** (FileNotFoundException e){  
 System.***out***.println(**"File not found. Specify file to read"**);  
 **return null**;  
 } **catch** (IOException e) {  
 System.***out***.println(**"File is not available now."**);  
 **return null**;  
 }  
 }

}

Листинг 20. Класс предиктивного нисходящего синтаксического анализатора.

Узлы дерева выражены классом Node. Поле *value* содержит экземпляры класса Token, а поле *idx* – уникальный индекс для вершины в пределах дерева.

**package** ru.osipov.labs.lab3.trees;  
  
**public class** Node<T> {  
 **protected** T **value**;  
 **protected int idx**;  
 **public void** setValue(T val) {  
 **this**.**value** = val;  
 }  
  
 **public** T getValue() {  
 **return value**;  
 }  
  
 **public void** setIdx(**int** i){  
 **this**.**idx** = i;  
 }  
  
 **public int** getIdx(){**return idx**;}  
  
 **public** Node(T val){  
 **this**.**value** = val;  
 }  
  
 **public** Node(){  
 **this**.**value** = **null**;  
 }  
}

Листинг 21. Класс Node.

Данный класс имеет подкласс LinkedNode, имеющий дополнительное поле *children*, представляющий список сыновей узла, а также поле *parent*, указывающее на родителя узла.

**package** ru.osipov.labs.lab3.trees;  
  
**import** ru.osipov.labs.lab1.structures.lists.LinkedList;  
  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
**public class** LinkedNode<T> **extends** Node<T> {  
 **private** List<LinkedNode<T>> **children**;  
 **private** LinkedNode<T> **parent**;  
 **public** LinkedNode(){  
 **children** = **new** ArrayList<>();  
 **idx** = -1;  
 }  
  
 **public void** setParent(LinkedNode<T> parent) {  
 **this**.**parent** = parent;  
 }  
  
 **public void** setChildren(List<LinkedNode<T>> children) {  
 **this**.**children** = children;  
 }  
  
 **public** LinkedNode<T> getParent() {  
 **return parent**;  
 }  
  
 **public** List<LinkedNode<T>> getChildren() {  
 **return children**;  
 }  
}

Листинг 22. Класс LinkedNode.

Для предоставления основных операций дерева, определены два интерфейса: Tree, и PositionalTree.

**package** ru.osipov.labs.lab3.trees;  
  
**public interface** Tree<T> {  
 Node<T> parent(Node<T> node);  
 Node<T> leftMostChild(Node<T> node);  
 Node<T> rightSibling(Node<T> node);  
 Node<T> root();  
 T value(Node<T> node);  
 **void** setVisitor(Visitor<T> visitor);  
 **int** getCount();  
 **void** clear();*//FROM ICollection<T>*}

**package** ru.osipov.labs.lab3.trees;  
  
**import** java.util.List;  
  
**public interface** PositionalTree<T> {  
 **void** addTo(Node<T> n, T item);  
 Node<T> rightMostChild(Node<T> n);  
 List<Node<T>> getChildren(Node<T> n);  
 **void** visit(VisitorMode order, Action<Node<T>> act);  
 PositionalTree<T> getSubTree(Node<T> n);  
}

Листинг 23. Интерфейсы Tree, PositionalTree.

Для посещения узлов и выполнения над ними определённой операции, были также созданы ещё два интерфейса, представляющие посетителя и действие: Visitor, Action.

**package** ru.osipov.labs.lab3.trees;  
  
**public interface** Action<T> {  
 **void** perform(T arg);  
}

**package** ru.osipov.labs.lab3.trees;  
  
**public interface** Visitor<T> {  
 **void** preOrder(Tree<T> n, Action<Node<T>> act);  
 **void** inOrder(Tree<T> t, Action<Node<T>> act);  
 **void** postOrder(Tree<T> n, Action<Node<T>> act);  
}

Листинг 24. Интерфейсы Action, Visitor.

Для выбора метода из интерфейса Vistior, определено перечисление VisitorMode.

**package** ru.osipov.labs.lab3.trees;  
  
**public enum** VisitorMode {  
 ***PRE***,***POST***,***IN***;  
}

Листинг 25. Перечисление VisitorMode.

Код дерева представлен ниже:

**package** ru.osipov.labs.lab3.trees;  
  
**import** ru.osipov.labs.lab1.structures.lists.LinkedList;  
**import** ru.osipov.labs.lab1.structures.lists.LinkedStack;  
  
**import** java.util.ArrayList;  
**import** java.util.HashSet;  
**import** java.util.List;  
  
**public class** LinkedTree<T> **implements** Tree<T>, PositionalTree<T> {  
 **private int \_count**;  
 **private** LinkedNode<T> **\_r**;  
 **private** Visitor<T> **\_visitor**;  
  
 **public** LinkedTree(){  
 **\_r** = **new** LinkedNode<T>();  
 **\_visitor** = **new** NRVisitor<T>();  
 **\_count** = 1;  
 }  
  
 **public** LinkedTree(LinkedNode<T> n){  
 n.setParent(**null**);  
 **\_r** = n;  
 **\_visitor** = **new** NRVisitor<T>();  
 **\_count** = 0;  
 Node<Integer> nc = **new** Node<>(**\_count**);  
 \_\_ComputeC(nc);  
 **\_count** = nc.getValue();  
 }  
  
  
 **private void** \_\_ComputeC(Node<Integer> val){  
 HashSet<Node<T>> hs = **new** HashSet<Node<T>>();  
 LinkedStack<Node<T>> STACK = **new** LinkedStack<>();  
 Node<T> n;  
 STACK.push(root());  
 **while**(!STACK.isEmpty()){  
 n = STACK.top();  
 **if**(hs.contains(n)) {  
 STACK.pop();  
 } **else** {  
 hs.add(n);  
 val.setValue(val.getValue() + 1);  
 List<Node<T>> children = getChildren(n);  
 **for**(**int** c = children.size() - 1; c >= 0; c--){  
 STACK.push(children.get(c));  
 }  
 }  
 }  
 }  
  
  
 **public void** add(T item){  
 LinkedNode<T> n = **\_r**;  
 **while**(n.getChildren().size() != 0){  
 n = n.getChildren().get(0);*//LEFTMOST\_CHILD* }  
 addTo(n,item);  
 }  
  
 */\*IPositionalTree implementation\*/* @Override  
 **public void** addTo(Node<T> n, T item){  
 LinkedNode<T> p = (LinkedNode<T>) n;  
 LinkedNode<T> it = **new** LinkedNode<T>();  
 it.setValue(item);  
 it.setParent(p);  
 **\_count**+= 1;  
 p.getChildren().add(it);  
 }  
  
 @Override  
 **public** Node<T> rightMostChild(Node<T> n){  
 LinkedNode<T> c = (LinkedNode<T>) n;  
 **if**(c == **null** || c.getChildren().size() == 0)  
 **return null**;  
 **return** c.getChildren().get(c.getChildren().size() - 1);  
 }  
  
  
  
 @Override  
 **public** List<Node<T>> getChildren(Node<T> n){  
 LinkedNode<T> c = (LinkedNode<T>) n;  
 **return new** ArrayList<>(c.getChildren());  
 }  
  
 @Override  
 **public** PositionalTree<T> getSubTree(Node<T> n){  
 LinkedNode<T> ln = (LinkedNode<T>) n;  
 **return new** LinkedTree<T>(ln);  
 }  
  
 @Override  
 **public void** visit(VisitorMode order, Action<Node<T>> act){  
 **switch**(order){  
 **case *PRE***:  
 **\_visitor**.preOrder(**this**,act);  
 **break**;  
 **case *POST***:  
 **\_visitor**.postOrder(**this**,act);  
 **break**;  
 **case *IN***:  
 **\_visitor**.inOrder(**this**,act);  
 **break**;  
 **default**:  
 **break**;  
 }  
 }  
  
 */\*Tree implementation\*/* @Override  
 **public** Node<T> root(){  
 **return \_r**;  
 }  
  
 @Override  
 **public** T value(Node<T> node){  
 **return** node.getValue();  
 }  
  
 @Override  
 **public** Node<T> parent(Node<T> node){  
 LinkedNode<T> np = (LinkedNode<T>) node;  
 **if**(np == **null** || np.getParent() == **null**){  
 **return null**;  
 }  
 **return** np.getParent();  
 }  
  
 @Override  
 **public** Node<T> leftMostChild(Node<T> node){  
 LinkedNode<T> np = (LinkedNode<T>)node;  
 **if**(np == **null** || np.getChildren() == **null** || np.getChildren().size() == 0){  
 **return null**;  
 }  
 **return** np.getChildren().get(0);  
 }  
  
 @Override  
 **public** Node<T> rightSibling(Node<T> node){  
 LinkedNode<T> np = (LinkedNode<T>)node;  
 **if**(np == **null** || np.getParent() == **null**){  
 **return null**;  
 }  
 LinkedNode<T> parent = np.getParent();  
 List<LinkedNode<T>> c = parent.getChildren();  
 **int** i = 0;*//own implementation of LinkedList.* **while**(!c.get(i).equals(np)){  
 i++;  
 }*//or zero.* **if**(i >= c.size() - 1)*//last element or more.* **return null**;  
 **return** c.get(i + 1);  
 }  
  
 @Override  
 **public int** getCount(){  
 **return this**.**\_count**;  
 }  
  
 @Override  
 **public void** clear(){  
 **\_r** = **null**;  
 **\_r** = **new** LinkedNode<T>();  
 **\_count** = 0;  
 }  
  
 @Override  
 **public void** setVisitor(Visitor<T> visitor){  
 **this**.**\_visitor** = visitor;  
 }  
  
 @Override  
 **public** String toString(){  
 StringBuilder sb = **new** StringBuilder();  
 HashSet<Node<T>> hs = **new** HashSet<Node<T>>();  
 LinkedStack<Node<T>> STACK = **new** LinkedStack<>();  
 Node<T> n;  
 STACK.push(root());  
 **while**(!STACK.isEmpty()){  
 n = STACK.top();  
 **if**(hs.contains(n)) {  
 STACK.pop();  
 sb.append(**"}"**);  
 } **else** {  
 hs.add(n);  
 sb.append(**"{"** + n.getValue().toString());  
 List<Node<T>> children = getChildren(n);  
 **for**(**int** c = children.size() - 1; c >= 0; c--){  
 STACK.push(children.get(c));  
 }  
 }  
 }  
 **return** sb.toString();  
 }  
  
 **public** String toDot(String fName){  
 StringBuilder sb = **new** StringBuilder();  
 HashSet<Node<T>> hs = **new** HashSet<Node<T>>();  
 LinkedStack<Node<T>> STACK = **new** LinkedStack<>();  
 Node<T> n;  
 STACK.push(root());  
 sb.append(**"digraph "**).append(fName).append(**" {"**);  
 **while**(!STACK.isEmpty()){  
 n = STACK.top();  
 **if**(hs.contains(n)) {  
 STACK.pop();  
 } **else** {  
 hs.add(n);  
 String name = n.getIdx()+**""**;  
 sb.append(name).append(**" [label=\""**).append(n.getValue().toString()).append(**"\"];"**);  
 List<Node<T>> children = getChildren(n);  
 **for**(**int** c = children.size() - 1; c >= 0; c--){  
 STACK.push(children.get(c));  
 sb.append(name).append(**" -> "**).append(children.get(c).getIdx()).append(**";"**);  
 }  
 }  
 }  
 sb.append(**"}"**);  
 **return** sb.toString();  
 }  
}

Листинг 26. Класс LinkedTree.

Метод *toString* представляет дерево в виде строки, оформленной в следующем виде:

* Для вершины дерева *a*, которое не имеет никаких сыновей, строковое представление имеет вид: “{*a*}”.
* Для вершины дерева *a,* имеющая прямых потомков *a1,…ak,* строковое представление имеет вид: “{*a*{}{}…{}}”.

Метод использует посетителя типа NRVisitor для обхода узлов дерева.

Код посетителя дан ниже:

**package** ru.osipov.labs.lab3.trees;  
  
**import** ru.osipov.labs.lab1.structures.lists.ArrStack;  
**import** ru.osipov.labs.lab1.structures.lists.LinkedStack;  
  
**public class** NRVisitor<T> **implements** Visitor<T> {  
  
 @Override  
 **public void** preOrder(Tree<T> tree, Action<Node<T>> act){  
 Node<T> m = tree.root();*//ROOT(T)* **if**(act == **null**){  
 act = (n) -> System.***out***.print(tree.value(n).toString()+**" "**);  
 }  
  
 ArrStack<Node<T>> STACK = **new** ArrStack<>(tree.getCount());  
  
 **while**(**true**){  
 **if**(m != **null**){  
 act.perform(m);*//LABEL(node,TREE)* STACK.push(m);  
 m = tree.leftMostChild(m);*//LEFTMOST\_CHILD(node,TREE)* }  
 **else**{  
 **if**(STACK.isEmpty()){  
 **return**;  
 }  
 m = tree.rightSibling(STACK.top());*//RIGHT\_SIBLING(TOP(S),TREE) where TOP(S) is node* STACK.pop();*//POP(S)* }  
 }  
 }  
  
 @Override  
 **public void** inOrder(Tree<T> t, Action<Node<T>> act) {  
 Node<T> m = t.root();*//ROOT(T)* **if**(act == **null**){  
 act = (n) -> System.***out***.print(t.value(n).toString()+**" "**);  
 }  
  
 ArrStack<Node<T>> STACK = **new** ArrStack<>(t.getCount());  
 ArrStack<Node<T>> STACK2 = **new** ArrStack<>(t.getCount());  
  
  
 **while**(**true**){  
 **if**(m != **null**){  
 STACK.push(m);  
 m = t.leftMostChild(m);*//LEFTMOST\_CHILD(node,TREE) while current != null current = current.leftson* }  
 **else**{  
 **if**(STACK.isEmpty()){  
 **return**;  
 }  
 *//Node<T> c = STACK.Top();  
 //1) 1,2 S: 4. 2)4 3 5 8 S: 4. 3) 4 6 10 S: empty.* act.perform(STACK.top());  
 m = t.rightSibling(t.leftMostChild(STACK.top()));*//right son of the STACK //current = top.rightson* STACK.pop();  
 **if**(m != **null** && t.rightSibling(m) != **null**){  
 STACK2.push(t.rightSibling(m));*//PUSH THIRD CHILD...* }  
  
 **if**(STACK.isEmpty() && m == **null** && !(STACK2.isEmpty())){  
 *//STACK.Push(STACK2.Top());  
 //STACK2.Pop();* m = STACK2.top();  
 STACK2.pop();  
 }  
 }  
 }  
 }  
  
 @Override  
 **public void** postOrder(Tree<T> tree, Action<Node<T>> act) {  
 Node<T> m = tree.root();*//ROOT(T)* **if**(act == **null**){  
 act = (n) -> System.***out***.print(tree.value(n).toString()+**" "**);  
 }  
  
 ArrStack<Node<T>> STACK = **new** ArrStack<>(tree.getCount());  
  
 **while**(**true**){  
 **if**(m != **null**){  
 *//act.perform(m);//LABEL(node,TREE)* STACK.push(m);  
 m = tree.leftMostChild(m);*//LEFTMOST\_CHILD(node,TREE)* }  
 **else**{  
 **if**(STACK.isEmpty()){  
 **return**;  
 }  
 act.perform(STACK.top());  
 m = tree.rightSibling(STACK.top());*//RIGHT\_SIBLING(TOP(S),TREE) where TOP(S) is node* STACK.pop();*//POP(S)* }  
 }  
 }  
}

Листинг 27. Класс посетителя NRVisitor.

Метод *toDot*, возвращает строковое описание ориентированного графа, в формате языка описания графов DOT. Данную строку читает модуль Graphviz, для визуализации полученного графа. Модуль лежит в библиотеке ***guru.nidi.graphviz-java*** версии 0.15.1. Для разбора языка DOT, библиотека использует программу dot. Программа dot не входит в набор модулей библиотеки и её необходимо устанавливать на компьютер отдельно.

**4. Результаты работы программы и краткие выводы.**

Для данной программы был разработан ряд тестов. Тестирование проводилось на трёх грамматиках. Первая грамматика представляет из себя грамматику арифметических выражений взятую из примера 2.27 [1]. Вторая грамматика представляет из себя грамматику *G\_3,* а третья грамматика – есть грамматика *G\_3m*. Программа читает грамматику из json файла. Формат такого файла описан в предыдущей работе [3]. Получив объект грамматики Grammar, данная грамматика преобразуется в приведённую грамматику (т.е. устраняются циклы, правила пустой строки, а также бесполезные символы). Из приведённой грамматики устраняется левая рекурсия. Затем происходит левая факторизация нелеворекурсивной грамматики. Затем, строится лексический анализатор с помощью экземпляра класса FALexerGenerator. Его метод buildNFA получает на вход грамматику (объект типа Grammar), а на выходе даёт комбинированный НКА. Из грамматики получают отображение: (имя\_терминала) –> список регулярных выражений, а также имя терминала, представляющего пустую строку (null). Эти значения хранятся в полях *lex\_rules, E*, объекта типа Grammar. Для каждого терминала и его каждого шаблона в шаблоне происходит замена всех вхождений подстроки, представляющей имя терминала пустой строки, на символ пустой строки. Затем, процедура addConcat раскрывает классы символов и конкатенацию. Наконец тип RegexRPNParser преобразует выражение в обратную польскую запись и эта запись с объектом, хранящим текущее возможное уникальное имя для новой вершины (класс Elem), подаётся на вход методу buildNFA класса Main из пакета lab1, для построения НКА для одиночного шаблона. Если шаблон является односимвольным, то вместо вызова Main.buildNFA, сразу строится НКА. К каждому допускающему состоянию НКА приписывается имя терминала, которое сохраняется в поле вершины автомата *value*. Получив комбинированный НКА, он подаётся на вход конструктору класса DFALexer, который преобразует его в ДКА с минимальным числом состояний. Грамматика и экземпляр класса DFALexer подаются на вход конструктору синтаксического анализатора, класса LLParser. Конструктор создаёт объект типа LLParserGenerator, и вызывает метод getTable, передав ему грамматику G. Метод getTable класса LLParserGenerator вычисляет множества FIRST и FOLLOW для грамматики G, и строит таблицу синтаксического анализа M которую он возвращает объекту класса LLParser. Загрузив таблицу синтаксического анализа, объект LLParser готов читать входные данные. Входные данные ему подаются в метод *parse*, который либо вовзращает дерево разбора в случае успеха, либо возвращает **null** в случаях синтаксических или лексических ошибок.

**package** ru.osipov.labs.lab3;  
  
**import** guru.nidi.graphviz.engine.Format;  
**import** guru.nidi.graphviz.engine.Graphviz;  
**import** org.junit.jupiter.api.Test;  
**import** org.junit.jupiter.api.extension.ExtendWith;  
**import** org.junit.runner.RunWith;  
**import** org.springframework.test.context.ContextConfiguration;  
**import** org.springframework.test.context.junit.jupiter.SpringExtension;  
**import** org.springframework.test.context.junit4.SpringJUnit4ClassRunner;  
**import** ru.osipov.labs.lab1.structures.automats.CNFA;  
**import** ru.osipov.labs.lab1.structures.automats.DFA;  
**import** ru.osipov.labs.lab2.grammars.Grammar;  
**import** ru.osipov.labs.lab2.jsonParser.SimpleJsonParser;  
**import** ru.osipov.labs.lab2.jsonParser.jsElements.JsonObject;  
**import** ru.osipov.labs.lab3.lexers.DFALexer;  
**import** ru.osipov.labs.lab3.lexers.Token;  
**import** ru.osipov.labs.lab3.lexers.generators.FALexerGenerator;  
**import** ru.osipov.labs.lab3.parsers.LLParser;  
**import** ru.osipov.labs.lab3.trees.LinkedTree;  
  
**import** java.io.File;  
**import** java.io.IOException;  
  
@ExtendWith(SpringExtension.**class**)  
@RunWith(SpringJUnit4ClassRunner.**class**)  
@ContextConfiguration(classes= TestLLParser.**class**)  
**public class** TestLLParser {  
  
 @Test  
 **public void** test1() **throws** IOException {  
 String p = System.*getProperty*(**"user.dir"**);  
 System.***out***.println(p);  
 p = p+**"\\src\\test\\java\\ru\\osipov\\labs\\lab3\\"**;  
 String dir = System.*getProperty*(**"user.dir"**) +**"\\src\\test\\java\\ru\\osipov\\labs\\lab3\\"**;  
 String s = System.*getProperty*(**"user.dir"**)+**"\\src\\test\\java\\ru\\osipov\\labs\\lab3\\"**;  
 s = s + **"input\\S\_G\_2\_27.txt"**;  
 p = p+**"grammarJson\\G\_2\_27.json"**;  
 SimpleJsonParser parser = **new** SimpleJsonParser();  
 JsonObject ob = parser.parse(p);  
 **assert** ob != **null**;  
 Grammar G = **new** Grammar(ob);  
 System.***out***.println(**"Source"**);  
 System.***out***.println(G);  
 G = Grammar.*deleteLeftRecursion*(G);  
 G = G.deleteLeftFactor();  
 **assert** G != **null**;  
 FALexerGenerator lg = **new** FALexerGenerator();  
 CNFA nfa = lg.buildNFA(G);  
 DFALexer lexer = **new** DFALexer(**new** DFA(nfa));  
 **assert** lexer.getFinished().size() == 5;  
 lexer.getImagefromStr(dir,**"lexer\_Test1"**);  
 LLParser sa = **new** LLParser(G,lexer);  
 LinkedTree<Token> t = sa.parse(G,s);  
 **assert** t != **null**;  
 System.***out***.println(t.toString());  
 String e = **"{E\_1{E\_1'1{E\_1'1{e}}{T\_2{T\_2'1{e}}{F\_3{a}}}{+}}{T\_2{T\_2'1{T\_2'1{e}}{F\_3{a}}{\*}}{F\_3{)}{E\_1{E\_1'1{E\_1'1{e}}{T\_2{T\_2'1{e}}{F\_3{a}}}{+}}{T\_2{T\_2'1{e}}{F\_3{a}}}}{(}}}}"**;  
 **assert** e.equals(t.toString());  
 Graphviz.*fromString*(t.toDot(**"parser\_test1"**)).render(Format.***PNG***).toFile(**new** File(dir+**"parser\_Test1"**));  
 }  
  
 @Test  
 **public void** test2() **throws** IOException {  
 String p = System.*getProperty*(**"user.dir"**);  
 System.***out***.println(p);  
 p = p+**"\\src\\test\\java\\ru\\osipov\\labs\\lab3\\"**;  
 String dir = System.*getProperty*(**"user.dir"**) +**"\\src\\test\\java\\ru\\osipov\\labs\\lab3\\"**;  
 String s = System.*getProperty*(**"user.dir"**)+**"\\src\\test\\java\\ru\\osipov\\labs\\lab3\\"**;  
 s = s + **"input\\S\_G\_lab3.txt"**;  
 p = p+**"grammarJson\\G\_Lab3\_3.json"**;  
 SimpleJsonParser parser = **new** SimpleJsonParser();  
 JsonObject ob = parser.parse(p);  
 **assert** ob != **null**;  
 Grammar G = **new** Grammar(ob);  
 System.***out***.println(**"Source"**);  
 System.***out***.println(G);  
 G = Grammar.*deleteLeftRecursion*(G);  
 G = G.deleteLeftFactor();  
 **assert** G != **null**;  
 FALexerGenerator lg = **new** FALexerGenerator();  
 CNFA nfa = lg.buildNFA(G);  
 DFALexer lexer = **new** DFALexer(**new** DFA(nfa));  
 **assert** lexer.getFinished().size() == 16;  
 lexer.getImagefromStr(dir,**"lexer\_Test2"**);  
 LLParser sa = **new** LLParser(G,lexer);  
 LinkedTree<Token> t = sa.parse(G,s);  
 **assert** t != **null**;  
 System.***out***.println(t.toString());  
 String e = **"{E\_1{AE\_2{AE\_2'2{AE\_2'2{e}}{T\_4{T\_4'1{e}}{F\_6{F\_6'1{e}}{PE\_8{num}}}}{PLUSOP\_5{-}}}{T\_4{T\_4'1{e}}{F\_6{F\_6'1{F\_6'1{e}}{PE\_8{id}}{^}}{PE\_8{num}}}}}{RELOP\_3{==}}{AE\_2{AE\_2'2{AE\_2'2{e}}{T\_4{T\_4'1{T\_4'1{e}}{F\_6{F\_6'1{e}}{PE\_8{id}}}{MULLOP\_7{\*}}}{F\_6{F\_6'1{e}}{PE\_8{id}}}}{PLUSOP\_5{+}}}{T\_4{T\_4'1{e}}{F\_6{F\_6'1{e}}{PE\_8{id}}}}}}"**;  
 **assert** e.equals(t.toString());  
 Graphviz.*fromString*(t.toDot(**"parser\_test2"**)).render(Format.***PNG***).toFile(**new** File(dir+**"parser\_Test2"**));  
 }  
  
 @Test  
 **public void** test3() **throws** IOException {  
 String p = System.*getProperty*(**"user.dir"**);  
 System.***out***.println(p);  
 p = p+**"\\src\\test\\java\\ru\\osipov\\labs\\lab3\\"**;  
 String dir = System.*getProperty*(**"user.dir"**) +**"\\src\\test\\java\\ru\\osipov\\labs\\lab3\\"**;  
 String s = System.*getProperty*(**"user.dir"**)+**"\\src\\test\\java\\ru\\osipov\\labs\\lab3\\"**;  
 s = s + **"input\\S\_G\_lab3\_mod.txt"**;  
 p = p+**"grammarJson\\G\_Lab3\_3\_m.json"**;  
 SimpleJsonParser parser = **new** SimpleJsonParser();  
 JsonObject ob = parser.parse(p);  
 **assert** ob != **null**;  
 Grammar G = **new** Grammar(ob);  
 System.***out***.println(**"Source"**);  
 System.***out***.println(G);  
 G = Grammar.*deleteLeftRecursion*(G);  
 G = G.deleteLeftFactor();  
 **assert** G != **null**;  
 FALexerGenerator lg = **new** FALexerGenerator();  
 CNFA nfa = lg.buildNFA(G);  
 DFALexer lexer = **new** DFALexer(**new** DFA(nfa));  
 **assert** lexer.getFinished().size() == 22;  
 lexer.getImagefromStr(dir,**"lexer\_Test3"**);  
 LLParser sa = **new** LLParser(G,lexer);  
 LinkedTree<Token> t = sa.parse(G,s);  
 **assert** t != **null**;  
 System.***out***.println(t.toString());  
 String e = **"{P\_1{}}{OPLIST\_3{OPLIST\_3'1{OPLIST\_3'1{OPLIST\_3'1{empty}}{OP\_4{}}{OPLIST\_3{OPLIST\_3'1{empty}}{OP\_4{E\_5{E\_5'1"** +  
 **"{AE\_6{AE\_6'2{AE\_6'2{empty}}{T\_8{T\_8'1{T\_8'1{empty}}{F\_10{F\_10'1{empty}}{PE\_12{id}}}{MULLOP\_11{\*}}}"** +  
 **"{F\_10{F\_10'1{empty}}{PE\_12{id}}}}{PLUSOP\_9{+}}}{T\_8{T\_8'1{empty}}{F\_10{F\_10'1{empty}}{PE\_12{25}}}}}{RELOP\_7{<=}}}{AE\_6{AE\_6'2{empty}}"** +  
 **"{T\_8{T\_8'1{empty}}{F\_10{F\_10'1{empty}}{PE\_12{25.34}}}}}}{=}{id}}}{{}}{;}}{OP\_4{E\_5{E\_5'1{empty}}{AE\_6{AE\_6'2{empty}}{T\_8{T\_8'1{empty}}"** +  
 **"{F\_10{F\_10'1{F\_10'1{empty}}{PE\_12{25}}{^}}{PE\_12{id}}}}}}{=}{id}}{;}}{OP\_4{E\_5{E\_5'1{AE\_6{AE\_6'2{empty}}{T\_8{T\_8'1{empty}}{F\_10{F\_10'1{F\_10'1{empty}}"** +  
 **"{PE\_12{i}}{^}}{PE\_12{25.2E-11}}}}}{RELOP\_7{<>}}}{AE\_6{AE\_6'2{AE\_6'2{empty}}{T\_8{T\_8'1{T\_8'1{empty}}{F\_10{F\_10'1{empty}}{PE\_12{id}}}"** +  
 **"{MULLOP\_11{\*}}}{F\_10{F\_10'1{empty}}{PE\_12{id}}}}{PLUSOP\_9{+}}}{T\_8{T\_8'1{empty}}{F\_10{F\_10'1{empty}}{PE\_12{id}}}}}}{=}{id}}}{{}}"**;  
 **assert** e.equals(t.toString());  
 Graphviz.*fromString*(t.toDot(**"parser\_test3"**)).render(Format.***PNG***).toFile(**new** File(dir+**"parser\_Test3"**));  
 }  
}

Листинг 28. Тестирование синтаксического и лексического анализаторов.

Все тесты успешно выполняются, строя правильное дерево разбора, а также минимальный ДКА для лексического анализатора. Для дополненной грамматики *G\_3m* эквивалентная левофакторизованная грамматика без левой рекурсии имеет следующие правила:

P\_1 -> { OPLIST\_3 }

RELOP\_7 -> == | <= | <> | < | > | >=

PE\_12 -> ( AE\_6 ) | **realNum** | **num** | **id**

AE\_6 -> T\_8 AE\_6'2 | PLUSOP\_9 T\_8 AE\_6'2

AE\_6'2 -> PLUSOP\_9 T\_8 AE\_6'2 |

F\_10'1 -> **^** PE\_12 F\_10'1 |

OPLIST\_3'1 -> **;** OP\_4 OPLIST\_3'1 |

T\_8 -> F\_10 T\_8'1

PLUSOP\_9 -> + | -

OP\_4 -> id = E\_5 | { OPLIST\_3 }

MULLOP\_11 -> % | \* | /

OPLIST\_3 -> OP\_4 OPLIST\_3'1

F\_10 -> PE\_12 F\_10'1

E\_5 -> AE\_6 E\_5'1

T\_8'1 -> MULLOP\_11 F\_10 T\_8'1 |

E\_5'1 -> RELOP\_7 AE\_6 |

Для шаблонов каждого терминального символа получим следующий ДКА.

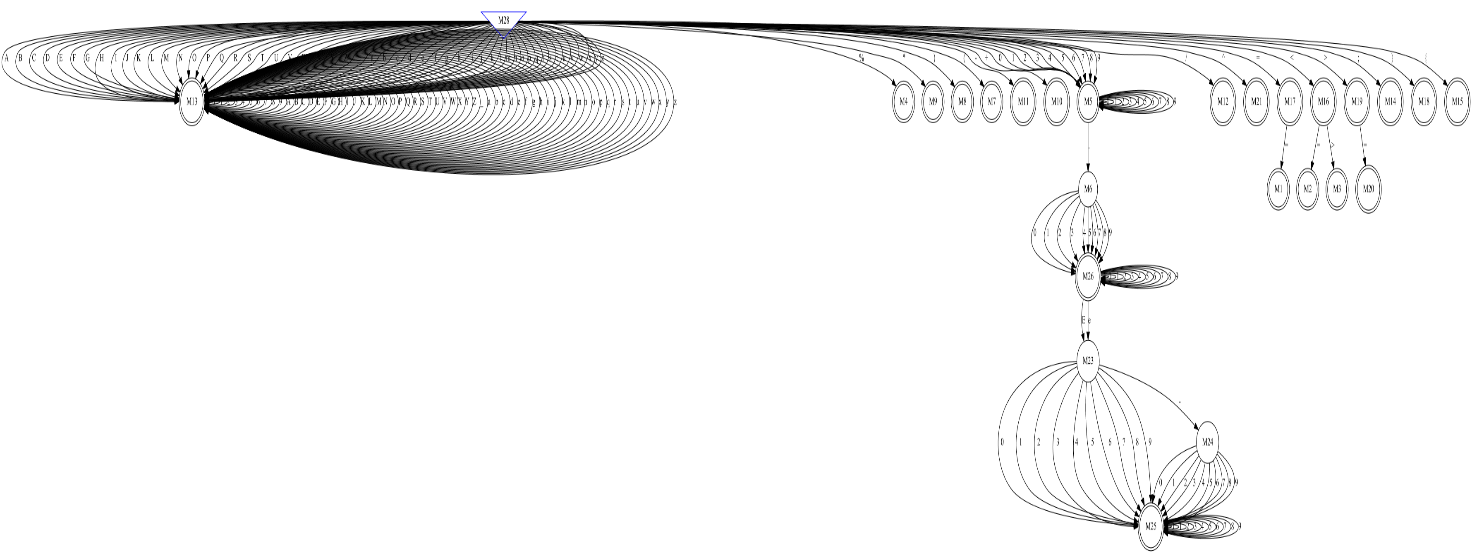


Рис 2. ДКА для лексического анализатора.

Для входного текстового файла S\_G\_lab3\_mod.txt, содержимое которое имеет вид

{  
 id = id + id \* id <> 25.2E-11 ^ i ;  
 id = id ^ 25 ;  
 {  
 id = 25.34 <= 25 + id \* id  
 }  
}

синтаксический анализатор построит следующее дерево разбора.

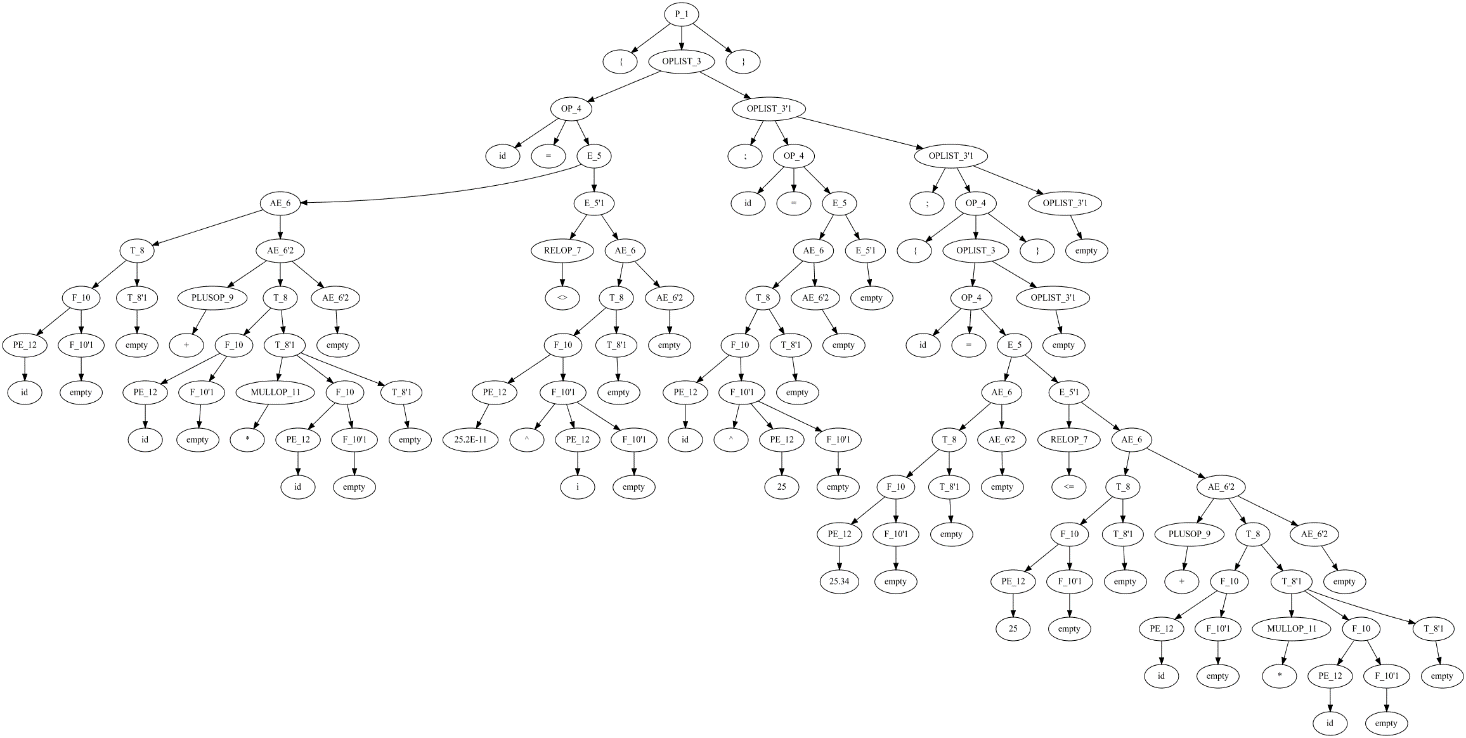


Рис. 3. Дерево разбора для входного файла S\_G\_lab3\_mod.txt.

У программы есть одно ограничение. Она работает только с LL-(1) грамматиками. У неё нет процедуры проверки того, является ли контекстно-свободная грамматика LL(1)-грамматикой. Для грамматик, которые не являются LL(1)-грамматиками программа может давать неправильный результат. Более того, проверка того, что для произвольной грамматики (в частности, для грамматик, которые не являются LL(1)), существует эквивалентная LL(1)-грамматика является алгоритмически неразрешимой.

**Список литературы и использованных источников.**

1. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции: В 2-х томах. Т.1.:Синтаксический анализ. –М.: Мир,1978.

2. Ахо А.В., Лам М.С., Сети Р., Ульман Дж. Д. Компиляторы: принципы, технологии и инструменты. –М.: Вильямс, 2008.

3. Осипов О.К. Отчёт по лабораторной работе №2.

4. Осипов О.К. Отчёт по лабораторной работе №1.

5. DFA minimization.

URL: <https://en.wikipedia.org/wiki/DFA_minimization#Hopcroft's_algorithm>

6. D. Grune, C. H. J. Jacobs "Parsing Techniques - A Practical Guide", Ellis Horwood, 1990.

URL: http://www.cs.vu.nl/~dick/PTAPG.html

7. Introduction to Recursive Descent Parsing.

URL: http://ag-kastens.uni-paderborn.de/lehre/material/uebi/parsdemo/recintro.html

8. Parsing Expressions by Recursive Descent.

URL: http://www.engr.mun.ca/~theo/Misc/exp\_parsing.htm