****

**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 № 2

по курсу

“Конструирование компиляторов”

Тема: Преобразование грамматик Устранение левой рекурсии, левая факторизация грамматики, устранение – правил из грамматики.

Вариант 3.

Выполнил студент группы ИУ7-21М:

Осипов Олег Константинович.

Преподаватель:

Ступников А.А.

Москва 2020 г.

**Преобразование грамматик. Преобразование грамматики в грамматику без – правил. Устранение левой рекурсии и левая факторизация грамматики.**

**1. Основные понятия.**

Формальные языки могут быть определены грамматиками. Существуют два типа грамматик: порождающие и распознающие. Любая порождающая грамматика характеризуется множеством терминалов, нетерминалов, правилами вывода, а также первым или начальным правилом вывода. Ниже дано формальное определение грамматики.

**Грамматикой *G*** называется четвёрка где:

* *N –* конечное множество *нетерминальных символов*, или

*нетерминалов (*иногда называемых вспомогательными символами, синтаксическими переменными);

* *T –* не пересекающееся с *N* конечное множество *терминальных*

*символов* или *терминалов*;

* *P –* конечное подмножество множества:

(элемент множества *P* называется *правилом* (или *продукцией*) и обозначается как ;

* *S –* выделенный символ из *N,* называемый *стартовым*

(*начальным*) *символом* грамматики*.*

Левую часть правила (первый элемент из пары ) называют *заголовком* правила (продукции), а правую часть – телом.

Для одной и той же левой части правила может существовать множество строк . Такие строки называются *альтернативами.* При наличии альтернатив можно использовать ниже приведённую сокращённую запись.

Пример 1.1.

Для множества правил

правила можно представить в следующем виде:

Грамматика порождает множество *выводимых цепочек,* состоящих из грамматических символов – терминалов и нетерминалов. Процесс порождения состоит в получении такого множества. Он определён следующим рекурсивным способом:

1) *S* – выводимая цепочка.

2) Если – выводимая цепочка и существует правило то

– тоже является выводимой цепочкой.

Выводимая цепочка грамматики *G,* состоящая только из терминальных символов называется *терминальной цепочкой* или *предложением.*

**Язык *L(G)*** порождаемый грамматикой *G –* это множество предложений данной грамматики.

Согласно иерархии Хомского, существуют следующие типы грамматик:

(1) **Праволинейной грамматикой *G*** называется грамматика, в которой каждое её правило из множества *P* имеет вид: Иными словами, левые части всех правил состоят ровно из одного нетерминала, а правые части представляют из себя либо строку из терминалов, либо строку из терминалов и нетерминала, стоящего в самом конце правила.

(2) **Контекстно-свободной грамматикой *G*** называют грамматику, в которой каждое её правило из множества *P* имеет вид: .

Иными словами, левые части всех правил такой грамматики состоят ровно из одного нетерминала.

(3) **Контекстно-зависимой грамматикой *G*** называют грамматику, в которой каждое её правило из множества *P* имеет вид: , и не существует правил вида .

(4) **Грамматикой общего вида *G***(*грамматика без ограничений*)называются грамматики, не удовлетворяющие ни одному из условий (1), (2), (3). Отметим, что контектно-свободная грамматика, содержащая правила вида , не является контекстно-зависимой. Грамматики типа (1) являются также грамматиками типа (2), (3) (если нет -правил). Т.е. каждая грамматика лежащая выше других, подходит под типы, определённые ниже, но не наоборот (не все контекстно-свободные языки являются регулярными).

Здесь и далее под грамматикой будет пониматься именно контекстно-свободная грамматика.

Для разбора грамматик ипользуют автоматы с магазинной памятью, представленные семёркой (T, Z, Q, F, q0, z0, f) – где T – алфавит входных символов, Z – алфавит памяти, состоящий из символов алфавита, состояний из множеств Q и Z, Q – множество состояний, составляющих основную память управляющего устройства, Z – стэк, или магазин, т.е. множество состояний вспомогательной памяти, q0 и z0 – начальные состояния из множеств Q и Z соответственно, F – множество допускающих состояний, f – функция перехода, принимающая один элемент из множества и возвращает подмножество множества . Входная цепочка принадлежит языку тогда, когда она полностью прочитана, а автомат находится в одном из допускающих состояний, причём его память опусташена, т.е. содержимое Z = .

Ниже приведены следующие обозначения, которые будут использоваться в дальнейшем:

1) Терминалы обозначаются:

* строчными буквами английского алфавита (a,b,с…)
* именами, выделенными жирным шрифтом: **id, num**
* символами операций (+, \*, / ^)
* символами пунктуации: (,), {, }, ,, ;,
* Арабскими цифрами 0…9.

2) Нетерминалы обозначаются:

* Заглавными буквами английского алфавита: A,B,C..
* Именами, выделенными курсивным шрифтом: *expr, stmt*
* S – имя начального символа грамматики.

3) Грамматические символы (как терминалы, так и нетерминалы) обозначаются заглавными буквами английского алфавита (c конца), выделенными курсивом *U, V, ..., Z .*

4) Цепочки терминалов обозначаются строчными буквами английского алфавита (с конца), выделенными курсивом *t, u, v, w, x, у, z*

5) Цепочки грамматических символов обозначаются строчными буквами греческого алфавита (с начала)

6) Правило вывода обозначаются:

Для первого правила, A = *S.* (Т.е. заголовок первого правила – есть начальный символ грамматики).

7) Отношение вывода строки из строки обозначается как . Для обозначения транзитивного замыкания вместо “стрелки” пишут а для транзитивно-рефлексивного замыкания - Вместо “звёздочки” и “плюса” можно использовать число, обозначающее степень отношения. К примеру, пусть даны следующие правила вывода: . Тогда следующие утверждения истины:

Грамматики будут описываться текстовыми файлами в формате JSON. Документ в формате JSON, описывающий грамматику, имеет следующую структуру:

{

“terms” : {

“a” : “id”,

“b” : [“0”,”1”],

“e” : null

},

“nonTerms” : [“S”],

“productions” : [{“S” : null}, {“S” : [”b”,”S”]}, {“S” : “a”}],

“start” : “S”

}

Листинг 1. Файл грамматики G1.json.

Значением свойства “terms” является объект, имена свойств которого

являются терминальными символами, а значения таких свойств – шаблоны для этих символов, которые могут быть использованы лексическим анализатором. Шаблон представляет из себя либо одиночную строку, либо список строк, которые обозначают регулярные выражения. Для шаблона пустой строки используется литерал *null*. В целях простоты, грамматика принимает только один терминальный символ для обозначения пустых строк. Если такой символ уже есть, то нельзя объявить ещё один. Иначе говоря, пустая строка соответствует ровно одному терминалу. В дальнейшем, можно потребовать, чтобы этому правилу соответствовали все шаблоны. Т.е. по одному терминалу для каждого шаблона. Списки же можно использовать, если они перечисляют части регулярного выражения, т.е. служат в роли оператора объединения двух или более регулярных выражений.

Значением свойства “nonTerms” является список строк, представляющих имена нетерминальных символов. Для обозначения начального символа грамматики используется значение свойства “start”, представляющее строку.

Значением свойства “productions” является список объектов, обозначающих продукции грамматики. Объект для продукции представлен в следующем виде: {“загаловок\_Продукции” : [<список грамматических символов>] }.

Имя свойства такого объекта обозначает заголовок продукции, состоящий ровно из одного нетерминального символа, значение такого свойства – список строк, обозначающих имена нетерминалов или терминалов. Для одиночного грамматического символа, необязательно использовать список, достаточно использовать строку. Для обозначения правила , можно использовать литерал *null*.

Учитывая выше сделанные предположения, приведённый документ

описывает следующую грамматику *G1 = (T = {a,b,e}, N = {S}, P = {}, S);*

Нетерминал *A* грамматики *G* называется *рекурсивным,* если

для некоторых . Если , то *A* называется *леворекурсивным.* Аналогично, если , то *A* называется *праворекурсивным.* Грамматика, имеющая хотя бы один *леворекурсивный* нетерминал называется *леворекурсивной грамматикой* или грамматика с левой рекурсией. Аналогично определяется *праворекурсивная грамматика.*

***-правилом*** грамматики называется правило вида: , где – терминал, обозначающий пустую строку. Здесь и далее, если не оговорено иное, символы – эквивалентны.

Грамматика называется грамматикой без *-правил* или *неукорачивающей грамматикой,* если она либо не имеет таких правил, либо существует одно правило вида: , и *S* не встречается в правых частях в никаких других правилах грамматики.

**Левая факторизация грамматики** – преобразование грамматики в эквивалентную грамматику, в которой все альтернативы каждой продукции из множества *P*, не имеет общих префиксов, т.е. , где .

**2. Постановка задачи.**

Дана контекстно-свободная грамматика *G = (T, N, P, S).* Необходимо её преобразовать в грамматику без -правил*,* без левой рекурсий, и в полученной грамматике *G’ = (T, N’, P’, S’)* провести левую факторизацию. Причём, полученная грамматика должна содержать минимальное количество новых правил, т.е. алгоритм не должен порождать избыточные правила.

**3. Программная реализация.**

**3.1. Структура проекта.**

Для выполнения задачи, будем использовать язык программирования Java 8. Создадим проект консольного приложения с использованием системы сборки проектов - gradle. Данная система позволяет загружать множество различных библиотек и модулей, и собирать их в единое монолитное приложение. В файле gradle.settings укажем имя приложения, добавив следующую строку: rootProject.name = **'CompilerCoursePr'.**

В файле build.gradle определены задачи, которые необходимо выполнять в процессе сборки. В частности, там есть задачи для очистки старых сборок, такие как clean, а есть задачи для создания jar-файла, например, jar. Для запуска проекта подключим фреймворк SpringBoot версии 2.1.7. а для тестов будем использовать библиотеку junit.jupiter версии 4.12.

Подключим также плагин application для указания главного класса, т.е. точки входа в приложение. Также определим задачи jar и bootJar для создания и запуска jar-файла приложения. В итоге содержимое файла build.gradle будет иметь следующий вид:

plugins **{** id **'java'** id **'application'** id **"org.springframework.boot"** version **"2.1.7.RELEASE"** id **"io.spring.dependency-management"** version **"1.0.8.RELEASE"  
}**group **'ru.osipov.labs'**version **'1.0-SNAPSHOT'**apply **plugin**: **'java'**apply **plugin**: **'idea'**apply **plugin**: **'application'**mainClassName = **'ru.osipov.labs.lab2.Main'**apply **plugin**: **'org.springframework.boot'**apply **plugin**: **'io.spring.dependency-management'**sourceCompatibility = 1.8  
targetCompatibility = 1.8  
repositories **{** mavenCentral()  
**}**idea **{** module **{** downloadJavadoc = **true** downloadSources = **true  
 }  
}**build.mustRunAfter clean  
task stage(**dependsOn**: [**'build'**, **'clean'**])  
  
task copyToLib(**type**: Copy) **{** into **"**$buildDir**/libs"** from(configurations.**compile**)  
**}**stage.dependsOn(copyToLib)  
  
  
dependencies **{** implementation **'org.springframework.boot:spring-boot-dependencies:2.1.7.RELEASE'** implementation **'org.springframework.boot:spring-boot-starter'** testImplementation **'org.springframework.boot:spring-boot-starter-test'** testImplementation **"org.junit.jupiter:junit-jupiter-api"** testImplementation **"org.junit.jupiter:junit-jupiter-engine"** testCompile **group**: **'junit'**, **name**: **'junit'**, **version**: **'4.12'** compile **group**: **'org.springframework.boot'**, **name**: **'spring-boot-starter-parent'**, **version**: **'2.1.8.RELEASE'**, **ext**: **'pom'  
}**test**{** useJUnitPlatform()  
**}**jar **{** manifest **{** attributes(**'Implementation-Title'**: **'CompilerCourse: Gradle Jar File '**,  
 **'Implementation-Version'**: **'1.0.0'**,  
 **'Main-Class'**: **"ru.osipov.labs.lab2.Main"**)  
 **}** archiveFileName = **"**$**{**project.name**}.jar"** enabled = **true** apply **plugin**: **'org.springframework.boot'  
}**bootJar **{** archiveFileName = **"**$**{**project.name**}.jar"  
}**

Листинг 2. Файл build.gradle.

В папке resources определим файл application.properties, для того, чтобы указать библиотеке SpringBoot, что это не веб-приложение, а консольное. Содержимое такого файла представлено ниже.

**spring.main.web-application-type**=**none  
mock**=**true**

Листинг 3. Файл application.properties.

**3.2. Создание JSON парсера.**

Ввиду того, что для представления грамматики используется формат текстовых данных JSON, то необходим модуль, который читает файлы такого формата, и преобразует их в нужную структуру. В данном случае, надо преобразовать json-файл в объект типа JsonObject, который представляет структуру документа в виде *нагруженного* дерева. Здесь вместо символов используются имена свойств, а массивы можно рассматривать как промежуточные узлы, которые по номеру элемента в массиве введут в дочерние узлы, принадлежащие массиву. Корнем такого дерева является объект типа JsonObject. Листьями являются объекты, представляющие элементарные скалярные значения. Они выражены типами JsonString, JsonBoolean, JsonNull, JsonNumber, JsonRealNumber, для строк, логических значений (true, false), для литерала null, и для чисел соответственно. Промежуточными узлами являются объекты типа JsonArray и JsonObject, представляющие массивы и вложенные объекты. Все эти типы имеют общего родителя – базового типа JsonElement<T>, представляющего элементы json документа. Для отображения свойств в виде строк вида “имя\_свойства” : значение, тип LinkedHashMap<String,V> переопределён типом StrLinkedHashMap<V>. Далее представлен исходный код вышеописанных типов.

**package** ru.osipov.labs.lab2.jsonParser.jsElements;  
  
**import** java.util.ArrayList;  
  
**public abstract class** JsonElement<T> {  
 **public abstract** T getValue();  
  
 @Override  
 **public** String toString(){  
 **return** getValue().toString();  
 }  
  
}

Листинг 4. Класс JsonElement<T>.

**package** ru.osipov.labs.lab2.jsonParser.jsElements;  
  
**public class** JsonBoolean **extends** JsonElement<Boolean> {  
  
 **private boolean v**;  
 **public** JsonBoolean(Character c){  
 **v** = (c == **'t'**);  
 }  
  
 @Override  
 **public** Boolean getValue() {  
 **return v**;  
 }  
  
}

Листинг 5. Класс JsonBoolean.

**package** ru.osipov.labs.lab2.jsonParser.jsElements;  
  
**public class** JsonNull **extends** JsonElement<String> {  
 @Override  
 **public** String getValue() {  
 **return "null"**;  
 }  
}

Листинг 6. Класс JsonNull.

**package** ru.osipov.labs.lab2.jsonParser.jsElements;  
  
**public class** JsonString **extends** JsonElement<String> {  
 **private** String **val**;  
 **public** JsonString(String v){  
 **this**.**val** = v;  
 }  
  
 **public void** setVal(String val) {  
 **this**.**val** = val;  
 }  
  
 @Override  
 **public** String getValue() {  
 **return val**;  
 }  
  
 @Override  
 **public** String toString(){  
 **return val**;  
 }  
}

Листинг 7. Класс JsonString.

**package** ru.osipov.labs.lab2.jsonParser.jsElements;  
  
**public class** JsonNumber **extends** JsonElement<Number> {  
  
 **private** Number **n**;  
  
 **public** JsonNumber(Number n){  
 **this**.**n** = n;  
 }  
  
 @Override  
 **public** String toString() {  
 **return n**.toString();  
 }  
  
 @Override  
 **public** Number getValue() {  
 **return n**;  
 }  
}

Листинг 8. Класс JsonNumber.

**package** ru.osipov.labs.lab2.jsonParser.jsElements;  
  
**public class** JsonRealNumber **extends** JsonNumber {  
  
 **public** JsonRealNumber(Float n){  
 **super**(n);  
 }  
 **public** JsonRealNumber(Double n) {  
 **super**(n);  
 }  
}

Листинг 9. Класс JsonRealNumber.

**package** ru.osipov.labs.lab2.jsonParser.jsElements;  
  
**import** java.util.ArrayList;  
  
**public class** JsonArray **extends** JsonElement<ArrayList<JsonElement>> {  
 **private** ArrayList<JsonElement> **elements**;  
 **private int c**;  
  
 **public** JsonArray(){  
 **this**.**elements** = **new** ArrayList<>();  
 **this**.**c** = 0;  
 }  
  
 **public** JsonArray(ArrayList<JsonElement> elements){  
 **this**.**elements** = elements;  
 **this**.**c** = elements.size();  
 }  
  
 **public void** setElements(ArrayList<JsonElement> elements) {  
 **this**.**elements** = elements;  
 }  
  
 **public** <T> **void** add(JsonElement<T> e){  
 **elements**.add(e);  
 **c**++;  
 }  
  
 **public** ArrayList<JsonElement> getElements() {  
 **return elements**;  
 }  
  
 **public int** getC() {  
 **return c**;  
 }  
  
 @Override  
 **public** ArrayList<JsonElement> getValue() {  
 **return elements**;  
 }  
}

Листинг 10. Класс JsonArray.

**package** ru.osipov.labs.lab2.jsonParser.jsElements;  
  
**import** ru.osipov.labs.lab2.jsonParser.JsParserState;  
**import** ru.osipov.labs.lab1.structures.lists.KeyValuePair;  
**import** ru.osipov.labs.lab1.structures.lists.LinkedStack;  
  
**import** java.util.\*;  
  
**public class** JsonObject **extends** JsonElement<StrLinkedHashMap<JsonElement>> {  
 **private** StrLinkedHashMap<JsonElement> **table**;  
  
 **public** JsonObject(){  
 **this**.**table** = **new** StrLinkedHashMap<>();  
 }  
  
 **public void** Put(String k, JsonElement id){  
 **this**.**table**.put(k,id);  
 }  
  
 **public** JsonElement Get(String k){*//get element on the root level.* **return table**.get(k);  
 }  
  
  
  
 **public** JsonElement getElement(String path){  
 String[] p = path.split(**"\\."**);  
 StrLinkedHashMap<JsonElement> cnode = **table**;*//ROOT.* JsonElement n = **null**;  
 **for**(String c : p){  
 n = cnode.get(c);  
 **try** {*//is interior node* **if**(n == **null**)  
 **return** n;  
 cnode = (StrLinkedHashMap<JsonElement>) n.getValue();*//isTable.* }**catch** (ClassCastException e){*//that was leaf.* **return** cnode.get(c);  
 }  
 }  
 **return** n;  
 }  
  
 @Override  
 **public** StrLinkedHashMap<JsonElement> getValue() {  
 **return table**;  
 }

}

Листинг 11. Класс JsonObject.

**package** ru.osipov.labs.lab2.jsonParser.jsElements;  
**import** java.util.Iterator;  
**import** java.util.LinkedHashMap;  
**import** java.util.Set;  
**public class** StrLinkedHashMap<V> **extends** LinkedHashMap<String,V> {  
 @Override  
 **public** String toString(){  
 Set<String> keys = keySet();  
 StringBuilder sb = **new** StringBuilder();  
 sb.append(**"{\n\t"**);  
 Iterator<String> t = keySet().iterator();  
 **while**(t.hasNext()){  
 String k = t.next();  
 sb.append(k+**" : "**+get(k).toString());  
 **if**(t.hasNext())  
 sb.append(**",\n\t"**);  
 }  
 sb.append(**"}"**);  
 **return** sb.toString();  
 }  
}

Листинг 12. Класс StrLinkedHashMap<V>.

Поскольку для грамматики JSON невозможно построить детерминированный конечный автомат из-за наличия правил структурной вложенности (объект может содержать другие объекты, а также массивы, кроме того, массивы также могут содержать объекты и массивы), то для него требуется построить автомат с магазинной памятью, который бы отслеживал текущую вложенность. Для такого автомата определён следующий алфавит памяти в виде перечисления, код которого представлен ниже:

**package** ru.osipov.labs.lab2.jsonParser;  
  
**public enum** JsParserState {  
 ***START***,***OPENROOT***,***CLOSEROOT***,***OPENBRACE***,***CLOSEBRACE***,***OPENARR***,***ARRELEM***,***CLOSEARR***,***OPENQ***,***CLOSEQ***,***OPENQP***,***CLOSEQP***,***COLON***,***ERR***,***FINISH***;

}

Листинг 13. Состояния автомата, в виде перечисления JsParserState.

Множество состояний автомата Q, вложено в алфавит памяти, т.е.

Алфавитом входных символов T, который читает автомат из json-файла, являются все символы, которые можно использовать в текстовых файлах. Автомат является детерминированным в том смысле, что для конечного подмножества элементов из множества сопоставлен ровно один элемент из множества , а для тех элементов, которых нет в данном множестве, автомат переходит в состояние ошибки (тупиковое состояние), которое означает, что документ не является json-документом. Начальным состоянием является состояние OPENROOT, конечным, т.е. допускающим состоянием является состояние CLOSEROOT. Автомат использует два стека помимо стека вспомогательной памяти, один для объектов, другой для массивов. Если автомат принял документ (состояние CLOSEROOT), то результатом его работы, будет объект типа JsonObject, находящийся на вершине стека объектов. Заметим, что объект также будет содержать все упомянутые в документе массивы. Функция перехода задана в виде метода changeState, которая принимает на вход текущий прочитанный символ *c*, текущее состояние *state*, файл в котором надо прочитать следующий символ *f,* стеки *objects arrays* для объектов и массивов, а также имя текущего свойства *propName,* и объект типа JsonString *arflag,* который хранит информацию о том, был ли прочитан массив, или нет. Она также работает с входным буфером автомата, и его стеком памяти. Перед циклом, который её вызывает до тех пор, пока не будет достигнуто либо принимающее либо состояние ошибки, происходит проверка того, что первый символ в документе является ‘{‘. Конечно, перед ним могло быть несколько пробельных символов, однако, здесь множество распознаваемых json-документов ограничено этим ограничением, в целях упрощения разработки парсера.

Таким образом, начальная и конечная конфигурация автомата может быть описана следующим образом.

c\_start = (OPENROOT, ‘{‘, OPENROOT),

c\_end = (CLOSEROOT, , ).

Конфигурация автомата хранит текущее состояние из множества *Q,* текущий входной символ и текущее состояние в памяти т.е. состояние из множества *Z.*

Документ принимается автоматом, если он завершает свою работу конечной конфигурацией (а не находится в состоянии ошибки). В случае тупикового состояния, автомат тут же прекращает свою работу, возвращая null, как сигнал о том, что не удалось распознать и построить структуру документа формата json.

Ниже представлен код данного автомата.

**package** ru.osipov.labs.lab2.jsonParser;  
  
**import** ru.osipov.labs.lab2.jsonParser.jsElements.\*;  
**import** ru.osipov.labs.lab1.structures.lists.\*;  
**import** ru.osipov.labs.lab1.structures.graphs.Pair;  
  
**import** java.io.\*;  
  
*//THIS UNIT HAS ONLY Json\_Parser functions yet.  
//****TODO: Parse specific JSON file to produce List of tiles.*public class** SimpleJsonParser {  
 **private** JsParserState **state**;  
 **private char**[] **buf**;  
 **private int bsize**;  
 **private int bufp**;  
 **private int EOL**;  
  
 **private int line**;  
 **private int col**;  
  
  
 **private** LinkedStack<JsParserState> **S1**;  
  
 **public** SimpleJsonParser(){  
 **this**.**state** = JsParserState.***START***;  
 **this**.**buf** = **new char**[100];  
 **this**.**bsize** = 100;  
 **this**.**bufp** = 0;  
  
 **this**.**EOL** = 0;  
 **this**.**line** = 1;  
 **this**.**col** = 0;  
 **this**.**S1** = **new** LinkedStack<>();  
 }  
  
  
 **public** JsonObject parse(String fileName) {  
 LinkedStack<JsonObject> obs = **new** LinkedStack<>();  
 obs.push(**new** JsonObject());  
 FileInputStream f;  
 LinkedStack<JsonArray> arrays = **new** LinkedStack<>();  
 **try** {  
 f = **new** FileInputStream(**new** File(fileName).getAbsolutePath());  
 InputStreamReader ch = **new** InputStreamReader(f);  
 **int** c = f.read();  
  
 **if**((**char**)c == **'{'**) {  
 **state** = JsParserState.***OPENROOT***;  
 **this**.**S1**.push(**state**);  
 **col**++;  
 }  
 **else** err((**char**)c,**"{"**);*//root not found!* JsonString pname = **new** JsonString(**""**);  
 JsonString pflag = **new** JsonString(**""**);  
 **while**(**state** != JsParserState.***CLOSEROOT*** && **state** != JsParserState.***ERR***){*//read characters until last '}' or error is found.* **char** cur;  
 **while**((cur = (**char**)getch(f)) == **' '** || cur == **'\t'** || cur == **'\n'** || cur == **'\r'**){  
 **if**(cur == **'\n'**) {  
 **line**++;  
 **col** = 0;  
 }  
 }  
 changeState(**state**,cur,f,obs,arrays,pname,pflag);  
 }  
 **if**(**state** == JsParserState.***ERR***)  
 **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**;  
 }  
 **return** obs.top();  
 }  
  
 **private void** changeState(JsParserState state, **char** c, InputStream f, LinkedStack<JsonObject> objects, LinkedStack<JsonArray> arrays, JsonString propName, JsonString arflag) **throws** IOException {  
 *//System.out.println("Current state: "+state+ " symbol -> "+c);  
 //System.out.println("Stack: "+S1+"\n");* JsonObject top = objects.top();  
 **if**(state == JsParserState.***OPENROOT*** ){*//|| state == JsParserState.OPENBRACE){* **if**(c == **'}'** )*//&& state == JsParserState.OPENROOT)* **this**.**state** = JsParserState.***CLOSEROOT***;  
 **else if**(c != **'\"'**) {  
 err(c, **"\""**);  
 }  
 **else** {  
 **this**.**state** = JsParserState.***OPENQP***;  
 }  
 }  
 **else if**(state == JsParserState.***OPENQP***){  
 **if**(c == **'\"'**) {  
 err(c, **"Any character except \" and EOF"**);  
 **return**;  
 }  
 **int** l = 1;  
 ungetch(c);  
 **while**((c = (**char**)getFilech(f)) != **'\"'**){ungetch(c);l++; **col**++;}  
 StringBuilder b = **new** StringBuilder();  
 **for**(**int** i = 0; i < **buf**.**length** && i < l; i++)  
 b.append(**buf**[i]);  
 propName.setVal(b.toString());  
 *//System.out.println("Property: "+propName);* b = **null**;  
 clear();  
 **this**.**state** = JsParserState.***CLOSEQP***;  
 }  
 **else if**(state == JsParserState.***CLOSEQP***){  
 **if**(c != **':'**)  
 err(c,**":"**);  
 **else** {  
 **this**.**state** = JsParserState.***COLON***;  
 }  
 }  
 **else if**(state == JsParserState.***COLON***){  
 **if**(c == **'['**) {  
 **this**.**state** = JsParserState.***OPENARR***;  
 arrays.push(**new** JsonArray());  
 top.Put(propName.getValue(),arrays.top());  
 **this**.**S1**.push(**this**.**state**);  
 }  
 **else if**(c == **'{'**) {  
 **this**.**state** = JsParserState.***START***;  
 objects.push(**new** JsonObject());  
 top.Put(propName.getValue(),objects.top());  
 **this**.**S1**.push(JsParserState.***OPENBRACE***);  
 }  
 **else if**(c == **'\"'**)  
 **this**.**state** = JsParserState.***OPENQ***;  
 **else if**(Character.*isDigit*(c) || c == **'-'**)  
 readNum(f,c,propName,top,**null**);  
 **else if**(c != **','** && c != **':'** && c != **'}'** && c != **']'**)*//read id (feature extension)* readId(f,c,propName,top,**null**);  
 **else** err(c,**"One of [ { \" d+ or true or false"**);  
 }  
 **else if**(state == JsParserState.***OPENARR***){  
 **this**.**S1**.push(JsParserState.***ARRELEM***);  
 **if**(c == **'['**) {  
 **this**.**state** = JsParserState.***OPENARR***;  
 JsonArray l = arrays.top();  
 arrays.push(**new** JsonArray());  
 l.add(arrays.top());  
 **this**.**S1**.push(**this**.**state**);  
 }  
 **else if**(c == **'{'**) {  
 **this**.**state** = JsParserState.***START***;  
 JsonArray l = arrays.top();  
 objects.push(**new** JsonObject());  
 l.add(objects.top());  
 **this**.**S1**.push(JsParserState.***OPENBRACE***);  
 }  
 **else if**(c == **'\"'**) {  
 arflag.setVal(**"A"**);  
 **this**.**state** = JsParserState.***OPENQ***;  
 }  
 **else if**(Character.*isDigit*(c))  
 readNum(f,c,**null**,**null**,arrays.top());*//read number [binary,octal,hex formats] and return CLOSEQ* **else if**(c != **','** && c != **':'** && c != **'}'** && c != **']'**)*//read id (feature extension)* readId(f,c,**null**,top,arrays.top());  
 **else** err(c,**"One of [ { \" d+ or true or false"**);  
 }  
  
 **else if**(state == JsParserState.***OPENQ***){  
 ungetch(c);  
 **int** l = 1;  
 **while**((c = (**char**)getFilech(f)) != **'\"'**){ungetch(c);l++;}  
 StringBuilder b = **new** StringBuilder();  
 **for**(**int** i = 0; i < **buf**.**length** && i < l; i++)  
 b.append(**buf**[i]);  
 String val = b.toString();  
 *//System.out.println(propName.getValue());* **if**(!arflag.getValue().equals(**"A"**))  
 top.Put(propName.getValue(),**new** JsonString(val));  
 **else**{  
 JsonArray array = arrays.top();  
 array.add(**new** JsonString(val));  
 }  
  
 b = **null**;  
 clear();  
 **this**.**state** = JsParserState.***CLOSEQ***;  
 }  
 **else if**(state == JsParserState.***CLOSEQ***){  
 **if**(c == **','** && **S1**.top() == JsParserState.***ARRELEM***){  
 **S1**.pop();  
 arflag.setVal(**""**);  
 **this**.**state** = **S1**.top();  
 }  
 **else if**(c == **','** && **S1**.top() == JsParserState.***OPENBRACE***){  
 **while**((c = (**char**)getch(f)) == **' '** || c == **'\t'** || c == **'\n'** || c == **'\r'**);*// skip spaces* **if**(c != **'\"'**) {  
 err(c, **"\""**);  
 }  
 **else  
 this**.**state** = JsParserState.***OPENQP***;  
 }  
 **else if**(c == **']'** && **S1**.top() == JsParserState.***ARRELEM***){  
 **S1**.pop();*//end of element* **S1**.pop();*//end of array.* arflag.setVal(**""**);  
 arrays.pop();  
 **if**(**S1**.top() == JsParserState.***ARRELEM***)  
 **this**.**state** = JsParserState.***CLOSEQ***;  
 **else** {  
 **S1**.push(JsParserState.***CLOSEARR***);  
 **this**.**state** = **S1**.top();  
 }  
 }  
 **else if**(c == **'}'** && **S1**.top() == JsParserState.***OPENROOT***){  
 **S1**.pop();  
 **this**.**state** = JsParserState.***CLOSEROOT***;  
 }  
 **else if**(c == **'}'** && **S1**.top() == JsParserState.***OPENBRACE***){  
 **S1**.pop();  
 **S1**.push(JsParserState.***CLOSEBRACE***);  
 **this**.**state** = **S1**.top();  
 }  
 **else if**(c == **','**){  
 **while**((c = (**char**)getch(f)) == **' '** || c == **'\t'** || c == **'\n'** || c == **'\r'**);*// skip spaces* **if**(c != **'\"'**) {  
 err(c, **"\""**);  
 }  
 **else  
 this**.**state** = JsParserState.***OPENQP***;  
 }  
 **else** err(c,**", or ] or }"**);  
 }  
 **else if**(state == JsParserState.***OPENBRACE***){  
 **if**(c == **'}'** && **S1**.top() == JsParserState.***OPENBRACE***){  
 **S1**.pop();  
 **S1**.push(JsParserState.***CLOSEBRACE***);  
 **this**.**state** = **S1**.top();  
 }  
 **else if** (c == **','**) {  
 **while** ((c = (**char**) getch(f)) == **' '** || c == **'\t'** || c == **'\n'** || c == **'\r'**) ;*// skip spaces* **if** (c != **'\"'**) {  
 err(c, **"\""**);  
 } **else  
 this**.**state** = JsParserState.***OPENQP***;  
 }  
 }  
 **else if**(state == JsParserState.***START***){  
 **if**(c == **'}'**){  
 **S1**.pop();  
 **S1**.push(JsParserState.***CLOSEBRACE***);  
 **this**.**state** = **S1**.top();  
 }  
 **else if**(c != **'\"'**){  
 err(c,**"Expected \""**);  
 }  
 **else  
 this**.**state** = JsParserState.***OPENQP***;  
 }  
 **else if**(state == JsParserState.***CLOSEARR***) {*//closearr openbrace arrelem* **S1**.pop();  
 arflag.setVal(**""**);  
 **if** (c == **','**) {  
 **while** ((c = (**char**) getch(f)) == **' '** || c == **'\t'** || c == **'\n'** || c == **'\r'**) ;*// skip spaces* **if** (c != **'\"'**) {  
 err(c, **"\""**);  
 } **else  
 this**.**state** = JsParserState.***OPENQP***;  
 }  
 **else if**(c == **'}'** && **S1**.top() == JsParserState.***OPENBRACE***){  
 **this**.**state** = JsParserState.***CLOSEBRACE***;  
 **this**.**S1**.pop();  
 **this**.**S1**.push(JsParserState.***CLOSEBRACE***);  
 ungetch(c);  
 }  
 }  
 **else if**(state == JsParserState.***CLOSEBRACE***){  
 **S1**.pop();  
 objects.pop();  
 top = objects.top();  
 **if**(c == **'}'** && **S1**.top() == JsParserState.***OPENROOT***)  
 **this**.**state** = JsParserState.***CLOSEROOT***;  
 **else if**(c == **'}'** && **S1**.top() == JsParserState.***OPENBRACE***){  
 **S1**.pop();  
 **S1**.push(JsParserState.***CLOSEBRACE***);  
 **this**.**state** = **S1**.top();  
 }  
 **else if**(c == **'}'** && **S1**.top() == JsParserState.***ARRELEM***){  
 **this**.**state** = JsParserState.***CLOSEQ***;  
 }  
 **else if**(c == **','** && **S1**.top() == JsParserState.***ARRELEM***){  
 **S1**.pop();  
 **this**.**state** = **S1**.top();  
 }  
 **else if**(c == **']'** && **S1**.top() == JsParserState.***ARRELEM***){  
 **S1**.pop();*//end of element* **S1**.pop();*//end of array.* **if**(**S1**.top() == JsParserState.***ARRELEM***)  
 **this**.**state** = JsParserState.***CLOSEQ***;  
 **else** {  
 **if**(**S1**.top() == JsParserState.***OPENROOT***){  
 **char** nc = (**char**) getch(f);  
 ungetch(nc);  
 **if**(nc == **','**)  
 **this**.**state** = JsParserState.***CLOSEQ***;  
 **else if**(nc == **'}'**)  
 **this**.**state** = JsParserState.***CLOSEROOT***;  
 **else  
 this**.**state** = **S1**.top();  
 }  
 **else  
 this**.**state** = **S1**.top();  
 }  
 }  
 **else if**(c == **','**){  
 **while** ((c = (**char**) getch(f)) == **' '** || c == **'\t'** || c == **'\n'** || c == **'\r'**) ;*// skip spaces* **if** (c != **'\"'**) {  
 err(c, **"\""**);  
 } **else  
 this**.**state** = JsParserState.***OPENQP***;  
 }  
 }  
 }  
  
 **private void** err(**char** f,String msg){  
 **state** = JsParserState.***ERR***;  
 System.***out***.println(**"Founded illegal symbol at ("**+**line**+**":"**+**col**+**"). Expected: "**+msg);  
 System.***out***.println(**"But founded: "**+f);  
 }  
  
 **private 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;  
 }  
 }  
  
 **private 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;  
 }  
 }  
  
 **private int** getFilech(InputStream r) **throws** IOException {  
 **char** c = (**char**)r.read();  
 **if**(c == **'\n'**){  
 **line** += 1;  
 **col** = 0;  
 }  
 **else**{  
 **col** += 1;  
 }  
 **return** c;  
 }  
  
 **private void** clear(){  
 **EOL** = 0;  
 **bufp** = 0;  
 **this**.**buf** = **null**;  
 **this**.**buf** = **new char**[**bsize**];  
 }  
  
 *//Read named token or name of the object (ob).* **private void** readId(InputStream f, **char** c,JsonString propName, JsonObject ob, JsonArray arr) **throws** IOException {  
 StringBuilder b = **new** StringBuilder();  
 **int** i = 0;  
 b.append(c);  
 **while**(i < 2147483647){  
 c = (**char**)getch(f);  
 **if**(c == **','** || c == **'\t'** || c ==**'\n'** || c == **' '** || c == **'}'** || c == **']'** || c == **'\r'**)  
 **break**;  
 b.append(c);  
 i++;  
 }  
 String v = b.toString();  
 **if**(v.equals(**"true"**)){  
 addLiteral(ob,propName,arr,**new** JsonBoolean(**'t'**));  
 }  
 **else if**(v.equals(**"false"**)){  
 addLiteral(ob,propName,arr,**new** JsonBoolean(**'f'**));  
 }  
 **else if**(v.equals(**"null"**)){  
 addLiteral(ob,propName,arr,**new** JsonNull());  
 }  
 **else**{  
 JsonElement l = ob.getElement(v);  
 addLiteral(ob,propName,arr,(l == **null**) ? **new** JsonString(v) : l);  
 }  
 **this**.**state** = JsParserState.***CLOSEQ***;  
 ungetch(c);  
 }  
  
 *//read numeric literal [Real: hex,binary,octal; Integer: hex,binary,octal;]* **private void** readNum(InputStream f, **char** c,JsonString propName, JsonObject ob, JsonArray arr) **throws** IOException {  
 **int** realf = 0, zerof = 1,ex = 0;  
 Pair<Integer,Integer> nflags = **new** Pair<>(0,0);  
 StringBuilder b = **new** StringBuilder();  
 **int** sign = 1;  
 **if**(c == **'-'**){  
 sign = -1;  
 c = (**char**)getch(f);  
 }  
 **if**(c == **'0'**) {  
 ungetch(c);*//save 0.* **if**((c = (**char**)getFilech(f)) == **'x'** || c == **'X'**){*//0x => hex-decimal preffix.* getch(f);*//preffix was read.* nflags.setV1(2);  
 zerof = 0;  
 c = (**char**)readNumericHex(f,c,nflags,b);  
 **int** che = nflags.getV2();  
 **if**(c == **'.'**){  
 b.append(c);  
 nflags.setV1(nflags.getV1() + 1);*//i++* **char** c1 = (**char**)getch(f);  
 ungetch(c1);  
 **if**((c1 >= **'0'** && c1 <= **'9'**) || (c1 >= **'A'** && c1 <= **'F'**) || (c1 >= **'a'** && c1 <= **'f'**)){  
 c = (**char**)readNumericHex(f,c,nflags,b);  
 nflags.setV2(1);  
 realf = 1;  
 }  
 **else if**(nflags.getV2() > 0){*//after dot there is no any hex digit.  
 //if non-exponent* err(c1,**"0-9 or A-F or a-f"**);  
 **return**;  
 }  
 }  
 **if**(nflags.getV2() == 0){  
 err(c,**"0-9 or A-F or a-f"**);  
 **return**;  
 }  
 **if**(c == **'P'** || c == **'p'** || c == **'H'** || c == **'h'**){  
 String n = b.toString();  
 **char** exp = c;  
 b = **null**;  
 b = **new** StringBuilder();  
 nflags.setV1(0);  
 nflags.setV2(0);  
 c = (**char**)readExponent(f,c,nflags,b);  
 **if**(nflags.getV2() == 0){*//exponent was read?* err(c,**"0-9"**);  
 **return**;  
 }  
 String e = b.toString();  
 b = **null**;  
 b = **new** StringBuilder();  
 b.append(ProcessExp.*parse*(n,e,exp,16,sign));  
 ex = 1;  
 }  
 **else**{  
 String n = b.toString();  
 b = **null**;  
 b = **new** StringBuilder();  
 b.append(ProcessExp.*parse*(n,**null**,**'O'**,16,sign));*//without exponent. parse it to decimal format (valid JSON)* }  
 **if**(c == **'L'** || c == **'l'** || c == **'S'** || c == **'s'**){  
 String num = b.toString();  
 **if**((c == **'L'** || c == **'l'**)) {  
 **if**(realf == 1 || num.contains(**"."**))  
 num = num.substring(0,num.indexOf(**'.'**));  
 addLiteral(ob,propName,arr,**new** JsonNumber(Long.*parseLong*(num)));  
 **this**.**state** = JsParserState.***CLOSEQ***;  
 }  
 **else**{  
 addLiteral(ob,propName,arr,**new** JsonRealNumber(Float.*parseFloat*(num)));  
 **this**.**state** = JsParserState.***CLOSEQ***;  
 }  
 **return**;  
 }  
 **else**{  
 addLiteral(ob,propName,arr,**new** JsonRealNumber(Double.*parseDouble*(b.toString())));  
 **this**.**state** = JsParserState.***CLOSEQ***;  
 *// if(c == ',' || c == '\t' || c =='\n' || c == ' ')* ungetch(c);  
 *// else if(c != 'D' && c != 'd')  
 // err(c,"Expected currect suffix [DdSsLl]");* **return**;  
 }  
 }  
 **else if**(c == **'B'** || c == **'b'**){ *//0b => binary preffix.* getch(f);*//preffix was read.* nflags.setV1(2);  
 zerof = 0;  
 c = (**char**)readNumericBin(f,c,nflags,b);  
 **if**(c == **'.'**){*//0b.2* b.append(c);  
 nflags.setV1(nflags.getV1() + 1);*//i++* **char** c1 = (**char**)getch(f);  
 ungetch(c1);  
 **if**((c1 >= **'0'** && c1 <= **'1'**)){  
 c = (**char**)readNumericBin(f,c,nflags,b);  
 nflags.setV2(1);  
 realf = 1;  
 }  
 **else if**(nflags.getV2() > 0){*//after dot there is no any hex digit.* err(c1,**"0,1"**);  
 **return**;  
 }  
 }  
 **if**(nflags.getV2() == 0){  
 err(c,**"0,1"**);  
 **return**;  
 }  
 **if**(c == **'P'** || c == **'p'** || c == **'E'** || c == **'e'**){  
 String n = b.toString();  
 **char** exp = c;  
 b = **null**;  
 b = **new** StringBuilder();  
 nflags.setV1(0);  
 nflags.setV2(0);  
 c = (**char**)readExponent(f,c,nflags,b);  
 **if**(nflags.getV2() == 0){*//exponent was read?* err(c,**"0-9"**);  
 **return**;  
 }  
 String e = b.toString();  
 b = **null**;  
 b = **new** StringBuilder();  
 b.append(ProcessExp.*parse*(n,e,exp,2,sign));  
 ex = 1;  
 }  
 **else**{  
 String n = b.toString();  
 b = **null**;  
 b = **new** StringBuilder();  
 b.append(ProcessExp.*parse*(n,**null**,**'O'**,2,sign));*//without exponent. parse it to decimal format (valid JSON)* }  
 **if**(c == **'L'** || c == **'l'** || c == **'F'** || c == **'f'**){  
 String num = b.toString();  
 **if**((c == **'L'** || c == **'l'**)) {  
 **if**(realf == 1 || num.contains(**"."**))  
 num = num.substring(0,num.indexOf(**'.'**));  
 addLiteral(ob,propName,arr,**new** JsonNumber(Long.*parseLong*(num)));  
 **this**.**state** = JsParserState.***CLOSEQ***;  
 }  
 **else**{  
 addLiteral(ob,propName,arr,**new** JsonRealNumber(Float.*parseFloat*(num)));  
 **this**.**state** = JsParserState.***CLOSEQ***;  
 }  
 **return**;  
 }  
 **else**{  
 addLiteral(ob,propName,arr,**new** JsonRealNumber(Double.*parseDouble*(b.toString())));  
 **this**.**state** = JsParserState.***CLOSEQ***;  
 *// if(c == ',' || c == '\t' || c =='\n' || c == ' ')* ungetch(c);  
 *// else if(c != 'D' && c != 'd')  
 // err(c,"Expected currect suffix [DdFfLl]");* **return**;  
 }  
 }  
 **else if**(c == **'C'** || c == **'c'**){  
 getch(f);*//preffix was read '0c'* nflags.setV1(2);  
 zerof = 0;  
 c = (**char**)readNumericOct(f,c,nflags,b);  
 **if**(c == **'.'**){  
 b.append(c);  
 nflags.setV1(nflags.getV1() + 1);*//i++* **char** c1 = (**char**)getch(f);  
 ungetch(c1);  
 **if**((c1 >= **'0'** && c1 <= **'7'**)){  
 c = (**char**)readNumericOct(f,c,nflags,b);  
 nflags.setV2(1);  
 realf = 1;  
 }  
 **else if**(nflags.getV2() > 0){*//after dot there is no any hex digit.* err(c1,**"0-7"**);  
 **return**;  
 }  
 }  
 **if**(nflags.getV2() == 0){  
 err(c,**"0-7"**);  
 **return**;  
 }  
 **if**(c == **'P'** || c == **'p'** || c == **'E'** || c == **'e'**){  
 String n = b.toString();  
 **char** exp = c;  
 b = **null**;  
 b = **new** StringBuilder();  
 nflags.setV1(0);  
 nflags.setV2(0);  
 c = (**char**)readExponent(f,c,nflags,b);  
 **if**(nflags.getV2() == 0){*//exponent was read?* err(c,**"0-9"**);  
 **return**;  
 }  
 String e = b.toString();  
 b = **null**;  
 b = **new** StringBuilder();  
 b.append(ProcessExp.*parse*(n,e,exp,8,sign));  
 ex = 1;  
 }  
 **else**{  
 String n = b.toString();  
 b = **null**;  
 b = **new** StringBuilder();  
 b.append(ProcessExp.*parse*(n,**null**,**'O'**,8,sign));*//without exponent. parse it to decimal format (valid JSON)* }  
 **if**(c == **'L'** || c == **'l'** || c == **'F'** || c == **'f'**){  
 String num = b.toString();  
 **if**((c == **'L'** || c == **'l'**)) {  
 **if**(realf == 1 || num.contains(**"."**))  
 num = num.substring(0,num.indexOf(**'.'**));  
 addLiteral(ob,propName,arr,**new** JsonNumber(Long.*parseLong*(num)));  
 **this**.**state** = JsParserState.***CLOSEQ***;  
 }  
 **else**{  
 addLiteral(ob,propName,arr,**new** JsonRealNumber(Float.*parseFloat*(num)));  
 **this**.**state** = JsParserState.***CLOSEQ***;  
 }  
 **return**;  
 }  
 **else**{  
 addLiteral(ob,propName,arr,**new** JsonRealNumber(Double.*parseDouble*(b.toString())));  
 **this**.**state** = JsParserState.***CLOSEQ***;  
 *// if(c == ',' || c == '\t' || c =='\n' || c == ' ')* ungetch(c);  
 *//else if(c != 'D' && c != 'd')  
 // err(c,"Expected currect suffix [DdFfLl]");* **return**;  
 }  
 }  
 getch(f);*//free zero.* b.append(**'0'**);  
 ungetch(c);  
 zerof = 1;  
 }  
 **else**{  
 **if**((c >= **'1'** && c <= **'9'**) || c == **'.'**) {  
 b.append(c);  
 ex = 1;  
 }  
 **else** {  
 err(c, **"Expected 0-9"**);  
 **return**;  
 }  
 }  
 **if**(c != **'.'**){*//decimal numeric literal.* c = (**char**)readNumeric(f,c,nflags,b);  
 **if**(c == **'.'**){  
 b.append(**'.'**);  
 nflags.setV1(nflags.getV1() + 1);  
 **char** c1 = (**char**)getch(f);  
 ungetch(c1);  
 **if**((c1 >= **'0'** && c1 <= **'9'**)){  
 c = (**char**)readNumeric(f,c,nflags,b);  
 nflags.setV2(1);  
 realf = 1;  
 }  
 **else if**(nflags.getV2() > 0){*//after dot there is no any hex digit.* err(c1,**"0-9"**);  
 **return**;  
 }  
 **else if**(ex == 1){  
 err(c1,**"0-9"**);  
 **return**;  
 }  
 }  
 *//exponent part* **if**(c == **'P'** || c == **'p'** || c == **'E'** || c == **'e'**){  
 String n = b.toString();  
 **char** exp = c;  
 b = **null**;  
 b = **new** StringBuilder();  
 nflags.setV1(0);  
 nflags.setV2(0);  
 c = (**char**)readExponent(f,c,nflags,b);  
 **if**(nflags.getV2() == 0){*//exponent was read?* err(c,**"0-9"**);  
 **return**;  
 }  
 String e = b.toString();  
 b = **null**;  
 b = **new** StringBuilder();  
 b.append(ProcessExp.*parse*(n,e,exp,10,sign));  
 ex = 1;  
 }  
 **else**{  
 String n = b.toString();  
 b = **null**;  
 b = **new** StringBuilder();  
 b.append(ProcessExp.*parse*(n,**null**,**'O'**,10,sign));*//without exponent. parse it to decimal format (valid JSON)* }  
 **if**(c == **'L'** || c == **'l'** || c == **'F'** || c == **'f'**){  
 String num = b.toString();  
 **if**((c == **'L'** || c == **'l'**)) {  
 **if**(realf == 1 || num.contains(**"."**))  
 num = num.substring(0,num.indexOf(**'.'**));  
 addLiteral(ob,propName,arr,**new** JsonNumber(Long.*parseLong*(num)));  
 **this**.**state** = JsParserState.***CLOSEQ***;  
 }  
 **else**{  
 addLiteral(ob,propName,arr,**new** JsonRealNumber(Float.*parseFloat*(num)));  
 **this**.**state** = JsParserState.***CLOSEQ***;  
 }  
 **return**;  
 }  
 **else** {  
 addLiteral(ob,propName,arr,**new** JsonRealNumber(Double.*parseDouble*(b.toString())));  
 **this**.**state** = JsParserState.***CLOSEQ***;  
 *// if(c == ',' || c == '\t' || c =='\n' || c == ' ' || c == ']' || c == '}')* ungetch(c);  
 *//else if(c != 'D' && c != 'd')  
 // err(c,"Expected currect suffix [DdFfLl]");* **return**;  
 }  
 }  
 **else** {*//dot token -> decimal number without integral part [i.e .123 or .5e3 or .95]* **if**(zerof == 1) {  
 getch(f);*//remove '.' from buffer if it was 0.* }  
 b.append(c);  
 **char** c1 = (**char**)getch(f);  
 ungetch(c1);  
 **if**((c1 >= **'0'** && c1 <= **'9'**)){  
 c = (**char**)readNumeric(f,c,nflags,b);  
 nflags.setV2(1);  
 realf = 1;  
 }  
 **else if**(zerof == 1){*//after dot there is no any hex digit.* err(c1,**"0-9"**);  
 **return**;  
 }  
 *//exponent part* **if**(c == **'P'** || c == **'p'** || c == **'E'** || c == **'e'**){  
 String n = b.toString();  
 **char** exp = c;  
 b = **null**;  
 b = **new** StringBuilder();  
 nflags.setV1(0);  
 nflags.setV2(0);  
 c = (**char**)readExponent(f,c,nflags,b);  
 **if**(nflags.getV2() == 0){*//exponent was read?* err(c,**"0-9"**);  
 **return**;  
 }  
 String e = b.toString();  
 b = **null**;  
 b = **new** StringBuilder();  
 b.append(ProcessExp.*parse*(n,e,exp,10,sign));  
 ex = 1;  
 }  
 **else**{  
 String n = b.toString();  
 b = **null**;  
 b = **new** StringBuilder();  
 b.append(ProcessExp.*parse*(n,**null**,**'O'**,10,sign));*//without exponent. parse it to decimal format (valid JSON)* }  
 **if**(c == **'L'** || c == **'l'** || c == **'F'** || c == **'f'**){  
 String num = b.toString();  
 **if**((c == **'L'** || c == **'l'**)) {  
 **if**(realf == 1 || num.contains(**"."**))  
 num = num.substring(0,num.indexOf(**'.'**));  
 addLiteral(ob,propName,arr,**new** JsonNumber(Long.*parseLong*(num)));  
 **this**.**state** = JsParserState.***CLOSEQ***;  
 }  
 **else**{  
 addLiteral(ob,propName,arr,**new** JsonRealNumber(Float.*parseFloat*(num)));  
 **this**.**state** = JsParserState.***CLOSEQ***;  
 }  
 **return**;  
 }  
 **else**{  
 addLiteral(ob,propName,arr,**new** JsonRealNumber(Double.*parseDouble*(b.toString())));  
 **this**.**state** = JsParserState.***CLOSEQ***;  
 *//if(c == ',' || c == '\t' || c =='\n' || c == ' ')* ungetch(c);  
 *//else if(c != 'D' && c != 'd')  
 // err(c,"Expected currect suffix [DdFfLl]");* **return**;  
 }  
 }  
 }  
  
 **private int** readNumeric(InputStream f, **char** c, Pair<Integer,Integer> flags, StringBuilder builder) **throws** IOException {  
 **int** cur = flags.getV1();  
 **int** nflag = 0;  
 **while**(cur < 2147483647){  
 **int** flag\_ = 1;  
 **while**(flag\_ > 0){  
 **while**( ((c = (**char**)getch(f)) >= **'0'** && c <= **'9'**) ){  
 builder.append(c);  
 cur += 1;  
 nflag = 1;  
 }  
 **if**(c == **'\_'**){  
 **while**(c == **'\_'**) c = (**char**)getFilech(f);  
 ungetch(c);*//after \_ it is digit or other symbol.* }  
 **else**{  
 flag\_ = 0;  
 }  
 }  
 flags.setV1(cur);  
 flags.setV2(nflag);  
 **return** c;  
 }  
 flags.setV1(cur);  
 flags.setV2(0);  
 *//this.state = JsParserState.ERR;* **return** -1;  
 }  
  
 **private int** readNumericHex(InputStream f, **char** c, Pair<Integer,Integer> flags, StringBuilder builder) **throws** IOException {  
 **int** cur = flags.getV1();  
 **int** nflag = 0;  
 **while**(cur < 2147483647){  
 **int** flag\_ = 1;  
 **while**(flag\_ > 0){  
 **while**( ((c = (**char**)getch(f)) >= **'0'** && c <= **'9'**) || (c >= **'A'** && c <= **'F'**) || (c >= **'a'** && c <= **'f'**) ){  
 builder.append(c);  
 cur += 1;  
 nflag = 1;  
 }  
 **if**(c == **'\_'**){  
 **while**(c == **'\_'**) c = (**char**)getFilech(f);  
 ungetch(c);*//after \_ it is digit or other symbol.* }  
 **else**{  
 flag\_ = 0;  
 }  
 }  
 flags.setV1(cur);  
 flags.setV2(nflag);  
 **return** c;  
 }  
 flags.setV2(0);  
 *//this.state = JsParserState.ERR;* **return** -1;  
 }  
  
 **private int** readNumericOct(InputStream f, **char** c, Pair<Integer,Integer> flags, StringBuilder builder) **throws** IOException {  
 **int** cur = flags.getV1();  
 **int** nflag = 0;  
 **while**(cur < 2147483647){  
 **int** flag\_ = 1;  
 **while**(flag\_ > 0){  
 **while**( ((c = (**char**)getch(f)) >= **'0'** && c <= **'7'**)){  
 builder.append(c);  
 cur += 1;  
 nflag = 1;  
 }  
 **if**(c == **'\_'**){  
 **while**(c == **'\_'**) c = (**char**)getFilech(f);  
 ungetch(c);*//after \_ it is digit or other symbol.* }  
 **else**{  
 flag\_ = 0;  
 }  
 }  
 flags.setV1(cur);  
 flags.setV2(nflag);  
 **return** c;  
 }  
 flags.setV2(0);  
 *//this.state = JsParserState.ERR;* **return** -1;  
 }  
  
 **private int** readNumericBin(InputStream f, **char** c, Pair<Integer,Integer> flags, StringBuilder builder) **throws** IOException {  
 **int** cur = flags.getV1();  
 **int** nflag = 0;  
 **while**(cur < 2147483647){  
 **int** flag\_ = 1;  
 **while**(flag\_ > 0){  
 **while**( ((c = (**char**)getch(f)) >= **'0'** && c <= **'1'**)){  
 builder.append(c);  
 cur += 1;  
 nflag = 1;  
 }  
 **if**(c == **'\_'**){  
 **while**(c == **'\_'**) c = (**char**)getFilech(f);  
 ungetch(c);*//after \_ it is digit or other symbol.* }  
 **else**{  
 flag\_ = 0;  
 }  
 }  
 flags.setV1(cur);  
 flags.setV2(nflag);  
 **return** c;  
 }  
 flags.setV2(0);  
 *//this.state = JsParserState.ERR;* **return** -1;  
 }  
  
 **private int** readExponent(InputStream f, **char** c, Pair<Integer,Integer> flags, StringBuilder builder) **throws** IOException {  
 **int** cur = flags.getV1();  
 **int** err = 0;  
 **while**(cur < 2147483647 && err < 2){  
 **if**((c = (**char**)getFilech(f)) == **'+'** || c == **'-'**){*//read sign of the exponent.* c = c == **'+'** ? **'+'** : **'-'**;  
 builder.append(c);  
 cur += 1;  
 err += 1;  
 }  
 **else** ungetch(c);  
 flags.setV1(cur);  
 c = (**char**)readNumeric(f, c,flags,builder);  
 **return** c;*//or suffix\_type or illegal symbol* }  
 flags.setV2(0);  
 **return** -1;  
 }  
  
 **private** <T> **void** addLiteral(JsonObject ob, JsonString propName, JsonArray arr,JsonElement<T> el){  
 **if**(propName == **null** || ob == **null**){  
 arr.add(el);  
 }  
 **else** ob.Put(propName.getValue(),el);  
 }  
}

Листинг 14 Json-парсер.

Отметим, что для хранения имён свойств и их скалярных значений используется входной буфер ограниченного размера. Для записи и извлечения символа из буфера используются функции *ungetch(char c)*, и *getch()*. Последняя будет читать из файла, если в буфере ничего нет. Для чтения символов из файла также определена отдельная функция *getFilech(InputStream r)*. Стандартом [1] определяется json-документ. Однако этот стандарт допускает только числа, которые представлены либо в виде целых чисел, либо в виде десятичных чисел с точкой и необязательной десятичной экспонентой. Если число начинается с нуля, то после него может идти только точка, с последующими цифрами от 0 до 9 и экспонентой. Он не допускает чисел представимых в виде двоичных, восьмеричных, шестнадцатеричных литералов. Кроме того, между цифрами не допускается использовать нижнее подчёркивание в качестве разделителя. Этот автомат позволяет читать такие числа. Кроме того, помимо десятичной экспоненты E, в автомате определена двоичная экспонента P, умножающее число на 2 в указанной степени после P. Для шестнадцатеричных чисел определена десятичная экспонента в виде буквы H (т.к. буквы А-Е используются в качестве цифр числа). Все прочитанные числа трансформируются в обычные десятичные числа (как целые, так и действительные), и сохраняются в объекте JsonObject. Для работы с экспонентами определён дополнительный тип ProcessExp.

**package** ru.osipov.labs.lab2.jsonParser;  
  
*//Process octal,hex,binary,decimal scientific literals to decimal number.***public class** ProcessExp {  
 **public static double** parse(String num, String e, **char** c,**int** base,**int** sign){*//c - type of exponent.* **double** me = 1;  
 **if**(c == **'P'** || c == **'p'**)*//binary exponent.* me = Math.*pow*(2.0,Double.*parseDouble*(e));  
 **else if**(c == **'E'** || c == **'e'** || c == **'H'** || c == **'h'**)*//H and h are exponent for hex numbers.* me = Math.*pow*(10.0,Double.*parseDouble*(e));  
 **else** me = 1;  
 **for**(**int** i = 0; i < num.length(); i++){  
 **char** s = num.charAt(i);  
 }  
 **if**(base == 16)  
 **return** *parse16*(num)\*me\*sign;  
 **else if**(base == 8)  
 **return** *parse8*(num)\*me\*sign;  
 **else if**(base == 2)  
 **return** *parse2*(num)\*me\*sign;  
 **else if**(base == 10)  
 **return** *parse10*(num)\*me\*sign;  
 **return** Double.***NaN***;  
 }  
 **private static double** parse16(String hex){  
 String digits = **"0123456789ABCDEF"**;  
 hex = hex.toUpperCase();  
 **double** val = 0;  
 **int** i = 0;  
 **while**(i < hex.length())  
 {  
 **char** c = hex.charAt(i);  
 **if**(c == **'.'**) {  
 i++;  
 **break**;  
 }  
 **int** d = digits.indexOf(c);  
 val = 16\*val + d;  
 i++;  
 }  
 **int** power = 1;  
 **while**(i < hex.length()){  
 **char** c = hex.charAt(i);  
 **int** d = digits.indexOf(c);  
 power \*= 16;  
 val = 16\*val + d;  
 i++;  
 }  
 **return** val/power;  
 }  
 **private static double** parse8(String num){  
 String digits = **"01234567"**;  
 num = num.toUpperCase();  
 **double** val = 0;  
 **int** i = 0;  
 **while**(i < num.length())  
 {  
 **char** c = num.charAt(i);  
 **if**(c == **'.'**) {  
 i++;  
 **break**;  
 }  
 **int** d = digits.indexOf(c);  
 val = 8\*val + d;  
 i++;  
 }  
 **int** power = 1;  
 **while**(i < num.length()){  
 **char** c = num.charAt(i);  
 **int** d = digits.indexOf(c);  
 power \*= 8;  
 val = 8\*val + d;  
 i++;  
 }  
 **return** val/power;  
 }  
 **private static double** parse2(String num){  
 String digits = **"01"**;  
 num = num.toUpperCase();  
 **double** val = 0;  
 **int** i = 0;  
 **while**(i < num.length())  
 {  
 **char** c = num.charAt(i);  
 **if**(c == **'.'**) {  
 i++;  
 **break**;  
 }  
 **int** d = digits.indexOf(c);  
 val = 2\*val + d;  
 i++;  
 }  
 **int** power = 1;  
 **while**(i < num.length()){  
 **char** c = num.charAt(i);  
 **int** d = digits.indexOf(c);  
 power \*= 2;  
 val = 2\*val + d;  
 i++;  
 }  
 **return** val/power;  
 }  
 **private static double** parse10(String num){  
 String digits = **"0123456789"**;  
 num = num.toUpperCase();  
 **double** val = 0;  
 **int** i = 0;  
 **while**(i < num.length())  
 {  
 **char** c = num.charAt(i);  
 **if**(c == **'.'**) {  
 i++;  
 **break**;  
 }  
 **int** d = digits.indexOf(c);  
 val = 10\*val + d;  
 i++;  
 }  
 **int** power = 1;  
 **while**(i < num.length()){  
 **char** c = num.charAt(i);  
 **int** d = digits.indexOf(c);  
 power \*= 10;  
 val = 10\*val + d;  
 i++;  
 }  
 **return** val/power;  
 }  
}

Листинг 15. Класс ProcessExp.

Второе отклонение от стандарта [1], это использование в качестве значения свойств регулярные определения, т.е. идентификаторы (имена) ранее определённых свойств, которые уже хранят определённые значения. При этом, в объекте сохраняется свойство со значением, на которое указывает регулярное определение (т.е. значение ранее определённого свойства).

Несмотря на такую широкую возможность, данный парсер не способен ещё обрабатывать управляющие последовательности символов, такие как ‘\t’, ‘\n’, ‘\r’, которые встречаются в именах свойств и в строковых литералах. Что касается символов ‘\n’, \t’, ‘\r’, и ‘ ‘, то они пропускаются автоматом (т.е. автомат читает следующий символ до тех пор, пока не встретит символ, не являющийся из выше перечисленных).

**3.3 Создание грамматики.**

Для представления типа грамматики, используется информация, полученная после обработки json-документа, и хранимая в виде экземпляра класса JsonObject. Для представления грамматического символа (как терминала, так и нетерминала) используется тип GrammarSymbol, хранящий имя символа в виде строки, хранимой в поле *val*, и типа символа (терминал, нетерминал) в виде одного символа (‘t’, ‘n’) в поле *type*.

**package** ru.osipov.labs.lab2.grammars;  
  
**public class** GrammarSymbol {  
 **private char type**;  
 **private** String **val**;  
  
 **public** GrammarSymbol(**char** type, String val){  
 **this**.**type** = type;  
 **this**.**val** = val;  
 }  
  
 **public void** setVal(String val) {  
 **this**.**val** = val;  
 }  
  
 **public** String getVal() {  
 **return val**;  
 }  
  
 **public void** setType(**char** type) {  
 **this**.**type** = type;  
 }  
  
 **public char** getType() {  
 **return type**;  
 }  
  
 @Override  
 **public** String toString() {  
 **return val**;  
 }  
  
 @Override  
 **public boolean** equals(Object o){  
 **try**{  
 **if**(o == **null**)  
 **return false**;  
 GrammarSymbol s = (GrammarSymbol) o;  
 **return** s.getType() == **type** && s.getVal().equals(**val**);  
 }  
 **catch** (ClassCastException e){  
 **return false**;  
 }  
 }  
}

Листинг 16. Класс GrammarSymbol.

Строки грамматических символов представлены типом GrammarString, хранящий список объектов типа GrammarSymbol в поле *symbols*.

**package** ru.osipov.labs.lab2.grammars;  
  
**import** java.util.ArrayList;  
**import** java.util.List;  
**import** java.util.Map;  
  
**public class** GrammarString {  
 **private** List<GrammarSymbol> **symbols**;  
  
 **public** GrammarString(){  
 **this**.**symbols** = **new** ArrayList<>(40);  
 }  
  
 **public** GrammarString(List<GrammarSymbol> symbols){  
 **this**.**symbols** = symbols;  
 }  
  
 **public void** setSymbols(List<GrammarSymbol> symbols) {  
 **this**.**symbols** = symbols;  
 }  
  
 **public** List<GrammarSymbol> getSymbols() {  
 **return symbols**;  
 }  
  
 **public void** addSymbol(GrammarSymbol s){  
 **symbols**.add(s);  
 }  
  
 **public** String getTypedStr(){  
 StringBuilder b = **new** StringBuilder();  
 **for**(GrammarSymbol s : **symbols**){  
 b.append(s.getType());  
 }  
 **return** b.toString();  
 }  
  
 @Override  
 **public boolean** equals(Object o){  
 **try**{  
 **if**(o == **null**)  
 **return false**;  
 GrammarString gs = (GrammarString) o;  
 **return** gs.getSymbols().equals(**symbols**);  
 }  
 **catch** (ClassCastException e){  
 **return false**;  
 }  
 }  
  
 @Override  
 **public int** hashCode(){  
 StringBuilder b = **new** StringBuilder();  
 **for**(GrammarSymbol s : **symbols**){  
 b.append(s.getVal());  
 }  
 **return** b.toString().hashCode();  
 }  
  
 @Override  
 **public** String toString(){  
 StringBuilder b = **new** StringBuilder();  
 **for**(GrammarSymbol s : **symbols**){  
 b.append(s.getVal());  
 }  
 **return** b.toString();  
 }  
  
 *//not very yet.* **public** String toString(Map<String,List<String>> lex\_rules){  
 StringBuilder b = **new** StringBuilder();  
 **for**(GrammarSymbol s : **symbols**){  
 b.append(s.getType() != **'t'** ? s.getVal() : lex\_rules.get(s.getVal()).get(0));  
 }  
 **return** b.toString();  
 }  
  
}

Листинг 17. Класс GrammarString.

Грамматика представлена типом Grammar. Она хранит множество терминалов *T*, множество нетерминалов *N*, отображение имён нетерминалов во множество грамматических строк GrammarString *P,* начальный символ грамматики *S*, терминал, представляющий пустую строку *E*, а также множества нетерминалов порождающих терминальные цепочки *N\_g*, и порождающих только пустые строки *N\_e* (т.е. порождающие ). Хоть это и не обязательно, но в классе Grammar определено дополнительное поле *lex\_rules*, представляющее отображение имён терминалов на список шаблонов. В дальнейшем оно может быть использовано для лексического анализатора.

Для каждого поля определён get-метод, возвращающий значение поля. Ниже представлены все поля и конструкторы класса Grammar.

**private** Set<String> **T**;  
**private** Set<String> **N**;  
**private** Map<String,Set<GrammarString>> **P**;*//left part of production is Non-terminal. (because it is only Context-Free-Grammar)***private** String **S**;  
**private** String **E**;*//terminal which means empty String.***private** Set<String> **N\_g**;*//Non-terminals which generates words.***private** Set<String> **N\_e**;*//Non-terminals which generates empty words.***private** Map<String, List<String>> **lex\_rules**; *//terminals with their patterns. (Lexical rules)***public** Grammar(Set<String> T, Set<String> N, Map<String,Set<GrammarString>> P,String start, String em, Map<String,List<String>> lexs){  
 **this**.**T** = T;  
 **this**.**N** = N;  
 **this**.**P** = P;  
 **this**.**S** = start;  
 **this**.**E** = em;  
 **this**.**lex\_rules** = lexs;  
 computeN\_g();  
 computeN\_e();  
}  
  
**public** Grammar(JsonObject jsonG){  
 JsonElement S = jsonG.getElement(**"start"**);  
 **if**(S **instanceof** JsonString)  
 **this**.**S** = ((JsonString) S).getValue();  
 **else  
 throw new** InvalidJsonGrammarException(**"Expected start property with String value!"**,**null**);  
  
 **this**.**E** = **""**;  
 JsonElement T = jsonG.getElement(**"terms"**);  
 **if**(T **instanceof** JsonObject){  
 **this**.**T** = **new** HashSet<>();  
 **lex\_rules** = **new** HashMap<>();  
 JsonObject o = (JsonObject) T;  
 Set<String> termNames = o.getValue().keySet();  
 **for**(String t : termNames){  
 List<String> l = **new** LinkedList<>();  
 **this**.**T**.add(t);  
 JsonElement el = o.getElement(t);  
 **if**(el **instanceof** JsonNull) {  
 **if**(!**this**.**E**.equals(**""**))  
 **throw new** InvalidJsonGrammarException(**"Only one empty-terminal is allowed. Found two or more"**,**null**);  
 **this**.**E** = t;  
 l.add(**""**);  
 }  
 **else if**(el **instanceof** JsonString) {  
 String val = ((JsonString) el).getValue();  
 **if**(val.equals(**this**.**E**)){*//value is empty string.* **if**(**this**.**E**.equals(**""**))  
 **this**.**E** = t;  
 **else  
 throw new** InvalidJsonGrammarException(**"Only one empty-terminal is allowed. Found two or more."**,**null**);  
 }  
 l.add(((JsonString) el).getValue());  
 }  
 **else if**(el **instanceof** JsonArray){  
 ArrayList<JsonElement> patterns = ((JsonArray) el).getElements();  
 **for**(JsonElement pattern : patterns){  
 **if**(pattern **instanceof** JsonString){  
 l.add(((JsonString) pattern).getValue());  
 }  
 **else  
 throw new** InvalidJsonGrammarException(**"Value of terminal "**+t+**" is not a String pattern"**,**null**);  
 }  
 }  
 **else  
 throw new** InvalidJsonGrammarException(**"Value of terminal "**+t+**" is not a valid pattern"**,**null**);  
 **lex\_rules**.put(t,l);  
 }  
 }  
 **else  
 throw new** InvalidJsonGrammarException(**"Expected terms property with value of the JsonObject with names of terminals and their values (list of String patterns)!\n\t terms : {term\_i : string | null | [string\_i...] , ... n} "**,**null**);  
  
 JsonElement N = jsonG.getElement(**"nonTerms"**);  
 **if**(N **instanceof** JsonArray){  
 **this**.**N** = **new** HashSet<>();  
 ArrayList<JsonElement> nterms = ((JsonArray) N).getElements();  
 **for**(JsonElement e : nterms){  
 **if**(e **instanceof** JsonString) {  
 **this**.**N**.add(((JsonString) e).getValue());  
 }  
 **else  
 throw new** InvalidJsonGrammarException(**"Expected String name of non-terminals."**,**null**);  
 }  
 }  
 **else  
 throw new** InvalidJsonGrammarException(**"Expected nonTerms property with list of String names of non-terminals"**,**null**);  
  
 JsonElement P = jsonG.getElement(**"productions"**);  
 **if**(P **instanceof** JsonArray){  
 **this**.**P** = **new** HashMap<>();  
 ArrayList<JsonElement> rules = ((JsonArray) P).getElements();  
 **for**(JsonElement e : rules){  
 **try** {  
 JsonObject o = (JsonObject) e;  
 **if**(o.getValue().keySet().size() == 0)  
 **continue**;  
 String key = o.getValue().keySet().iterator().next();*//get first property of object (name of production)* JsonElement rpart = o.Get(key);*//get list of grammar symbols related to production.* Set<GrammarString> product\_rules = **this**.**P**.get(key);*//try get information about production to identify alternatives* **if**(product\_rules == **null**)  
 product\_rules = **new** HashSet<>();  
 **if**(rpart **instanceof** JsonArray){  
 GrammarString alpha = **new** GrammarString();  
 ArrayList<JsonElement> symbols = ((JsonArray) rpart).getElements();  
 **for**(JsonElement symbol : symbols){  
 **if**(symbol **instanceof** JsonString){  
 String X = ((JsonString) symbol).getValue();  
 GrammarSymbol entity;  
 **if**(**this**.**T**.contains(X))  
 entity = **new** GrammarSymbol(**'t'**,X);  
 **else if**(**this**.**N**.contains(X)) {  
 entity = **new** GrammarSymbol(**'n'**, X);  
 }  
 **else  
 throw new** InvalidJsonGrammarException(**"Illegal grammar symbol!"** +  
 **"\nExpected non-terminal or termnial of Grammar."**,**null**);  
 alpha.addSymbol(entity);  
 }  
 **else  
 throw new** InvalidJsonGrammarException(**"Grammar Symbol must be String!"**,**null**);  
 }  
 product\_rules.add(alpha);  
 }  
 **else if**(rpart **instanceof** JsonString){  
 GrammarString alpha = **new** GrammarString();  
 String X = ((JsonString) rpart).getValue();  
 GrammarSymbol entity;  
 **if**(**this**.**T**.contains(X))  
 entity = **new** GrammarSymbol(**'t'**,X);  
 **else if**(**this**.**N**.contains(X))  
 entity = **new** GrammarSymbol(**'n'**,X);  
 **else  
 throw new** InvalidJsonGrammarException(**"Illegal grammar symbol!"** +  
 **"\nExpected non-terminal or termnial of Grammar."**,**null**);  
 alpha.addSymbol(entity);  
 product\_rules.add(alpha);  
 }  
 **else if**(rpart **instanceof** JsonNull){  
 GrammarSymbol s = **new** GrammarSymbol(**'t'**, **this**.**E**);  
 GrammarString alpha = **new** GrammarString();  
 alpha.addSymbol(s);  
 product\_rules.add(alpha);  
 }  
 **else  
 throw new** InvalidJsonGrammarException(**"Illegal grammar string! Expected String, Null or list of Strings"**,**null**);  
 **this**.**P**.put(key,product\_rules);  
 }  
 **catch** (ClassCastException err){  
 **throw new** InvalidJsonGrammarException(**"productions is not a list of objects!"**,err);  
 }  
 }  
 }  
 **else  
 throw new** InvalidJsonGrammarException(**"productions is not a list!"**,**null**);  
 computeN\_g();  
 computeN\_e();  
}

Листинг 18. Поля и конструкторы класса Grammar.

Для построения множеств *N\_g*, и *N\_e* используются соответствующие процедуры *computeN\_g* и *computeN\_e.* Они реализуют алгоритм 2.7 из [2]. Для вычисления *N\_e* в алгоритме 2.7 необходимо лишь положить, что множество терминалов *T* = {}.

*//Compute set of non-terminals which generates empty-strings***private void** computeN\_e(){  
 Set<String> N\_0 = **new** HashSet<>();  
 Set<String> N\_i = **new** HashSet<>();  
 LinkedStack<Set<String>> S = **new** LinkedStack<>();  
 S.push(N\_0);  
 **boolean** t1 = **true**;  
 **while**(**true**){  
 N\_0 = S.top();  
 Set<String> rules = **P**.keySet();  
 **for** (String p : rules) {  
 Set<GrammarString> a = **P**.get(p);*//get all grammar strings with rule p.* **for** (GrammarString alpha : a) {*//for each alternative* List<GrammarSymbol> symbols = alpha.getSymbols();  
 **for** (GrammarSymbol s : symbols) {*//check that alpha is (N\_i-1 U T)\** **if** (!s.getVal().equals(**this**.**E**) && !(N\_0.contains(s.getVal()))) {  
 t1 = **false**;  
 **break**;*//stop scanning string* }  
 }  
 **if**(t1) { *//valid grammar string -> add rule.* N\_i.add(p);  
 **break**;*//stop scanning alternatives.* }  
 t1 = **true**;*//continue scanning rule P for valid alternatives (grammar strings)* }  
 }  
 S.pop();  
 S.push(N\_i);  
 **if**(N\_i.equals(N\_0)) {  
 **N\_e** = N\_i;  
 **return**;  
 }  
 N\_i = **new** HashSet<>();  
 }  
}  
  
*//Compute set of non-terminals which generate strings of Language L(G).***private void** computeN\_g(){  
 Set<String> N\_0 = **new** HashSet<>();  
 Set<String> N\_i = **new** HashSet<>();  
 LinkedStack<Set<String>> S = **new** LinkedStack<>();  
 S.push(N\_0);  
 **boolean** t1 = **true**;  
 **while**(**true**){  
 N\_0 = S.top();  
 N\_i.addAll(N\_0);  
 Set<String> rules = **P**.keySet();  
 **for** (String p : rules) {  
 Set<GrammarString> a = **P**.get(p);*//get all grammar strings with rule p.* **for** (GrammarString alpha : a) {*//for each alternative* List<GrammarSymbol> symbols = alpha.getSymbols();  
 **for** (GrammarSymbol s : symbols) {*//check that alpha is (N\_i-1 U T)\** **if** (s.getType() != **'t'** && !(N\_0.contains(s.getVal()))) {  
 t1 = **false**;  
 **break**;*//stop scanning string* }  
 }  
 **if**(t1) { *//valid grammar string -> add rule.* N\_i.add(p);  
 **break**;*//stop scanning alternatives.* }  
 t1 = **true**;*//continue scanning rule P for valid alternatives (grammar strings)* }  
 }  
 S.pop();  
 S.push(N\_i);  
 **if**(N\_i.equals(N\_0)) {  
 **N\_g** = N\_i;  
 **return**;  
 }  
 N\_i = **new** HashSet<>();  
 }  
}

Листинг 19. Методы computeN\_g и computeN\_e.

Ниже приведён код процедуры, реализующий алгоритм 2.10 из [2], для устранения -правил из грамматики *G*. Процедура getNonEmptyWordsGrammar использует методы *getAllESubsetOf*, *getERulesFor* и *isEmptyString*. Первая возвращает множество всех подмножеств для грамматической строки, состоящих из таких комбинаций, в которых либо присутствует, либо удалён один из нетерминалов принадлежащий множеству *N\_e*. Вторая возвращает количество нетерминалов из множества *N\_e* лежащих в данной строке. Третья проверяет, состоит ли строка только лишь из символов из *N\_e,* либо состоит из одного терминала для пустой строки .

**public** Grammar getNonEmptyWordsGrammar(){  
 Set<String> rules = **P**.keySet();  
 Map<String,Set<GrammarString>> newP = **new** HashMap<>();  
 String newS = **this**.**S**;  
 Set<String> NN = **new** HashSet<>();  
  
 *//Check whether S -> eps. (ifTrue then add Rule S' -> S | eps).* Set<GrammarString> S = **this**.**P**.get(**this**.**S**);  
 **for**(GrammarString al : S){  
 List<GrammarSymbol> str = al.getSymbols();  
 **if**( (str.size() == 1 && str.get(0).getVal().equals(**this**.**E**)) || isEmptyString(al)){  
 newS = newS+**"\'"**;  
 GrammarString al1 = **new** GrammarString();  
 GrammarString al2 = **new** GrammarString();  
 al1.addSymbol(**new** GrammarSymbol(**'n'**,**this**.**S**));  
 al2.addSymbol(**new** GrammarSymbol(**'t'**,**this**.**E**));  
 Set<GrammarString> srbody = **new** HashSet<>();  
 srbody.add(al1);  
 srbody.add(al2);  
 newP.put(newS,srbody);  
 NN.add(newS);  
 **break**;  
 }  
  
 }  
  
 *//Generate new set of rules.* **for** (String p : rules) {*//for each production.* Set<GrammarString> rule = **P**.get(p);  
 Set<GrammarString> nbody = **new** HashSet<>();  
 **for**(GrammarString r : rule){  
 Set<GrammarString> rbody = getAllESubsetOf(r);  
 nbody.addAll(rbody);  
 }  
 **if**(nbody.size() > 0) {  
 newP.put(p, nbody);  
 NN.add(p);  
 }  
 }  
 **return new** Grammar(**this**.**T**,NN,newP,newS,**this**.**E**,**this**.**lex\_rules**);  
}  
  
*//Get all subsets of rule which consists of combinations of N\_e.***private** Set<GrammarString> getAllESubsetOf(GrammarString rule){  
 Set<GrammarString> result = **new** HashSet<>();  
 LinkedStack<GrammarString> S = **new** LinkedStack<>();  
 S.push(rule);  
 **while**(!S.isEmpty()) {  
 GrammarString i = S.top();  
 S.pop();  
 **int** k = getERulesFor(i);  
 *//Do not add rules A -> empty (A -> eps).* **if**(k == 0 && i.getSymbols().size() == 1 && i.getSymbols().get(0).getVal().equals(**this**.**E**))  
 **continue**;  
 **else if**(k == 1 && i.getSymbols().size() == 1){  
 result.add(i);  
 **continue**;  
 }  
 result.add(i);*//add new combo into set.* ArrayList<Integer> passed = **new** ArrayList<>();  
 **while** (k > 0) {*//split on combinations.* GrammarString nr = **new** GrammarString();  
 **boolean** flag = **true**;  
 **int** j1 = 0;  
 **for** (GrammarSymbol sym : i.getSymbols()){  
 **if**(flag && **N\_e**.contains(sym.getVal()) && !passed.contains(j1)){  
 flag = **false**;  
 passed.add(j1);  
 j1++;  
 **continue**;  
 }  
 GrammarSymbol s = **new** GrammarSymbol(sym.getType(),sym.getVal());  
 nr.addSymbol(s);  
 j1++;  
 }  
 *//do not add rules A -> empty. (A -> eps)* **if**(nr.getSymbols().size() == 1 && nr.getSymbols().get(0).getType() == **'t'** && nr.getSymbols().get(0).getVal().equals(**this**.**E**)){  
 k++;  
 **continue**;  
 }  
 S.push(nr);  
 k--;  
 }  
 }  
 **return** result;  
}  
  
*//for grammarString count N\_e non-terminals.***public int** getERulesFor(GrammarString rule){  
 List<GrammarSymbol> l = rule.getSymbols();  
 **int** i = 0;  
 **for**(GrammarSymbol s : l){  
 **if**(**N\_e**.contains(s.getVal()))  
 i++;  
 }  
 **return** i;  
}  
  
**private boolean** isEmptyString(GrammarString r){  
 List<GrammarSymbol> l = r.getSymbols();  
 **for**(GrammarSymbol s : l){  
 **if**(s.getType() == **'t'** && s.getVal().equals(**this**.**E**))  
 **continue**;  
 **if**(!**N\_e**.contains(s.getVal()))  
 **return false**;  
 }  
 **return true**;  
}

Листинг 20. Устранение eps-правил из грамматики.

Поскольку множество всех подмножеств содержит до 2^m элементов, где m – количество нетерминалов из *N\_e* в строке грамматических символов, представляющих одну из альтернатив продукции, то алгоритм имеет экспоненциальную сложность. Для её устранения, можно преобразовать грамматику таким образом, чтобы каждое её правило состояло не более чем из двух грамматических символов. Для этого можно воспользоваться алгоритмом устранения длинных правил из грамматики. Идея его заключается в том что для данного правила , создать k – 2 нетерминалов, удалить исходное правило *А*, заменив его новыми k – 1 правилами вида: .

Пример 3.1

Правило заменяется на правила

Ниже представлен код, реализующий вышеописанный алгоритм.

*//Shorten rules which have more than 2 GrammarSymbols. (A -> aBbB, B -> c => A -> aB1, B1 -> BB2, B2 -> bB, B -> c)***public** Grammar getShortenedGrammar(){  
 Set<String> rules = **P**.keySet();  
 Set<String> NN = **new** HashSet<>();  
 Map<String,Set<GrammarString>> newP = **new** HashMap<>();  
 **for**(String p : rules) {  
 Set<GrammarString> r = **P**.get(p);  
 Set<GrammarString> oldr = **new** HashSet<>();  
 List<GrammarSymbol> str = **null**;  
 **int** j = 1;  
 Optional<GrammarString> op = r.stream().filter(x -> x.getSymbols().size() > 2).findFirst();  
 **if** (op.isPresent()) {*//has at least one alternative with 2 or more symbols* str = op.get().getSymbols();  
 List<GrammarSymbol> finalStr = str;  
 r = r.stream().filter(x -> !x.getSymbols().equals(finalStr)).collect(Collectors.*toSet*());*//r its all except first alternative which is scanning now  
 //if (str.size() > 2) {//if A1 consists of three or more symbols...* String pr = p;  
 **for** (**int** i = 0; i < str.size() - 2; i++) {  
 GrammarString n = **new** GrammarString();  
 n.addSymbol(str.get(i));  
 n.addSymbol(**new** GrammarSymbol(**'n'**, p + j));  
 NN.add(p + j);  
 Set<GrammarString> nbody = **new** HashSet<>();  
 nbody.add(n);  
 newP.put(pr, nbody);  
 pr = p + j;  
 j++;  
 }  
 GrammarString last = **new** GrammarString();  
 last.addSymbol(str.get(j - 1));  
 last.addSymbol(str.get(j));  
 NN.add(pr);  
 Set<GrammarString> nbody = **new** HashSet<>();  
 nbody.add(last);  
 newP.put(pr, nbody);  
 *//}//end scaning first alternative. (else add oldr alternative)* Iterator<GrammarString> it = r.iterator();  
 **while** (it.hasNext()) {*//scanning other alternatives.  
 //for (int k = 0; k < r.size(); k++) {* **int** str2\_idx = 1;  
 GrammarString rk = it.next();  
 List<GrammarSymbol> str2 = rk.getSymbols();*//r.get(k).getSymbols();* Set<GrammarString> nbody2 = newP.get(p);  
 **if** (str2.size() > 2) {  
 GrammarString sn = **new** GrammarString();  
 sn.addSymbol(str2.get(0));  
 sn.addSymbol(**new** GrammarSymbol(**'n'**, p + j));  
 nbody2.add(sn);  
 nbody2 = **new** HashSet<>();  
 String pr2 = p + j;  
 NN.add(pr2);  
 j++;  
 str2\_idx++;  
 **for** (**int** i = 1; i < str2.size() - 2; i++) {  
 GrammarString n = **new** GrammarString();  
 n.addSymbol(str2.get(i));  
 n.addSymbol(**new** GrammarSymbol(**'n'**, p + j));  
 nbody2.add(n);  
 newP.put(pr2, nbody2);  
 nbody2 = **new** HashSet<>();  
 pr2 = p + j;  
 NN.add(pr2);  
 j++;  
 str2\_idx++;  
 }  
 GrammarString last2 = **new** GrammarString();  
 last2.addSymbol(str2.get(str2\_idx - 1));  
 last2.addSymbol(str2.get(str2\_idx));  
 nbody2.add(last2);  
 newP.put(pr2, nbody2);  
 } **else** {  
 oldr.add(rk);  
 }  
 }*//end scaning alternatives.* } **else** {  
 oldr.addAll(r);*//r.get(0);* }  
 Set<GrammarString> fl = newP.get(p);  
 **if**(fl != **null**){  
 fl.addAll(oldr);*//if some alternatives were short. (they are not producing new rules)* }  
 **else**{  
 newP.put(p,oldr);*//if all alternatives were short.* }  
 NN.add(p);  
 }  
 **return new** Grammar(**this**.**T**,NN,newP,**this**.**S**,**this**.**E**,**this**.**lex\_rules**);  
}

Листинг 21. Устранение длинных правил из грамматики.

После получения грамматики из исходной посредством удаления из исходной длинных правил (метод *getShortenedGrammar*), в грамматике будут правила, не превышающие длину в 2 символа. Применив к ней уже процедуру *getNonEmptyWordsGrammar*, получим эквивалентную к исходной грамматику *G’,* не содержащую -правила. Заметим, что такая грамматика очень близка к нормальной форме Хомского. Она не соответствует по определению нормальной формы Хомского лишь наличием правил, которые содержат два терминальных символа, а также правил, содержащих один терминальный символ и один нетерминальный символ и правил, содержащих ровно один нетерминальный символ. Для правил, содержащих ровно один нетерминальный символ, можно применить алгоритм 2.11 из [2], удаляющий цепные правила из грамматики. Для остальных правил можно произвести замену терминального символа *a* на новый нетерминальный символ *,* для которого определено правило Естественно, в ходе замены, можно проверять, был ли создан нетерминал или нет, для определения дальнейших действий (замены или замены со вставкой нового правила ).

Для грамматики, у которой все правила не превышают длину в 2 символа, метод *getNonEmptyWordsGrammar* не превышает сложность O(n), где n = k\*m, где k – количество нетерминалов, а m = максимально возможное число альтернатив для правил грамматики. Т.е. алгоритм будет иметь линейную полиномиальную сложность.

В грамматике могут быть недостижимые и бесполезные символы.

Символ грамматики *G* называется *недостижимым,* если он не появляется ни в какой любой выводимой цепочке.

Символ грамматики *G* называется *бесполезным*, если в грамматике нет вывода вида . Т.е. символ X не порождает никакую терминальную цепочку, хотя может встречаться в выводимой цепочке.

Ниже представлена процедура *deleteUselessSymbols*, получающая новую грамматику из исходной посредством удаления бесполезных символов. Она реализует алгоритм 2.9 из [2], и использует метод *deleteUnreachableSymbols* для устранения недостижимых символов. *deleteUnreachableSymbols* в свою очередь реализует алгоритм 2.8. из [2].

*//Algorithm 2.8***public** Grammar deleteNonReachableSymbols(){  
 Set<String> V\_0 = **new** HashSet<>();  
 Set<String> V\_i = **new** HashSet<>();  
 LinkedStack<Set<String>> S = **new** LinkedStack<>();  
 Map<String,Set<GrammarString>> newP = **new** HashMap<>();  
 V\_0.add(**this**.**S**);  
 S.push(V\_0);  
 **while**(**true**){  
 V\_0 = S.top();*//i++* V\_i.addAll(V\_0);  
 **for**(String p : V\_0){  
 Set<GrammarString> a = **P**.get(p);*//get all grammar strings with rule p.* **if**(a == **null**)  
 **continue**;  
 Set<GrammarString> np = **new** HashSet<>();  
 **for** (GrammarString alpha : a) {*//for each alternative* List<GrammarSymbol> symbols = alpha.getSymbols();  
 GrammarString g = **new** GrammarString();  
 **for** (GrammarSymbol s : symbols) {*//FOR symbol Xi in A -> X1...XN for A in N\_i-1* V\_i.add(s.getVal());  
 g.addSymbol(s);  
 }  
 np.add(g);  
 }  
 **if**(**N**.contains(p)){  
 newP.put(p,np);  
 }  
 }  
 S.pop();  
 S.push(V\_i);  
 **if**(V\_i.equals(V\_0)) {  
 **break**;  
 }  
 V\_i = **new** HashSet<>();  
 }  
 Set<String> NN = **new** HashSet<>(V\_i);  
 NN.retainAll(**N**);  
 Set<String> NT = **new** HashSet<>(V\_i);  
 NT.retainAll(**T**);  
  
 **return new** Grammar(NT,NN,newP,**this**.**S**,**this**.**E**,**lex\_rules**);  
}  
  
*//Algorithm 2.9***public** Grammar deleteUselessSymbols(){  
 Map<String,Set<GrammarString>> newP = **new** HashMap<>();  
 Set<String> NN = **new** HashSet<>(**N**);  
 NN.retainAll(**this**.**N\_g**);  
 Set<String> ps = **P**.keySet();  
 **for**(String p : ps){  
 Set<GrammarString> bodies = **P**.get(p);  
 Set<GrammarString> nbodies = **new** HashSet<>();  
 **for**(GrammarString str : bodies){  
 **boolean** f = **true**;  
 GrammarString g = **new** GrammarString();  
 **for**(GrammarSymbol s : str.getSymbols()){  
 **if**(s.getType() != **'t'** && !NN.contains(s.getVal())){  
 f = **false**;  
 **break**;  
 }  
 g.addSymbol(**new** GrammarSymbol(s.getType(),s.getVal()));  
 }  
 **if**(f){  
 nbodies.add(g);  
 }  
 }  
 **if**(nbodies.size() > 0)  
 newP.put(p,nbodies);  
 }  
 Grammar NG = **new** Grammar(**this**.**T**,NN,newP,**this**.**S**,**this**.**E**,**lex\_rules**);  
 **return** NG.deleteNonReachableSymbols();  
}

Листинг 22. Удаление бесполезных символов из грамматики.

Правило вида является *цепным правилом.* Ниже приведён метод, *getNonCycledGrammar,* возвращающий новую грамматику без цепных правил. Метод реализует алгоритм 2.11 из [2] и работает только с грамматиками без -правил и без бесполезных символов.

**public** Set<String> compute\_allGeneratedN(String N){  
 Set<String> V\_0 = **new** HashSet<>();  
 Set<String> V\_i = **new** HashSet<>();  
 LinkedStack<Set<String>> S = **new** LinkedStack<>();  
 V\_0.add(N);  
 S.push(V\_0);  
 **while**(**true**){  
 V\_0 = S.top();*//i++* V\_i.addAll(V\_0);  
 **for**(String p : V\_0){  
 Set<GrammarString> a = **P**.get(p);*//get all grammar strings with rule p.* **if**(a == **null**)  
 **continue**;  
 Set<GrammarString> np = **new** HashSet<>();  
 **for** (GrammarString alpha : a) {*//for each alternative* **if**(alpha.getSymbols().size() == 1 && alpha.getSymbols().get(0).getType() != **'t'**)  
 V\_i.add(alpha.getSymbols().get(0).getVal());  
 }  
 }  
 S.pop();  
 S.push(V\_i);  
 **if**(V\_i.equals(V\_0)) {  
 **break**;  
 }  
 V\_i = **new** HashSet<>();  
 }  
 **return** V\_i;  
}  
  
*//Algorithm 2.11 Delete rules A -> B, where A,B are non-terminals and length of the body == 1.***public** Grammar getNonCycledGrammar(){

**if**(**N\_e**.size() > 0)  
 **return** getNonEmptyWordsGrammar().getNonCycledGrammar();

Map<String,Set<GrammarString>> newP = **new** HashMap<>();  
 Map<String,Set<String>> NG = **new** HashMap<>();  
 **for**(String n : **N**){  
 NG.put(n,compute\_allGeneratedN(n));  
 }  
 Set<String> ps = **P**.keySet();  
 **for**(String p : ps){  
 Set<GrammarString> rules = **P**.get(p);  
 Set<String> np = **new** HashSet<>();  
 **for**(String k : NG.keySet()){  
 **if**(NG.get(k).contains(p))  
 np.add(k);  
 }  
 **for**(GrammarString rule : rules){  
 *//if rule is not like A -> B.* **if**( (rule.getSymbols().size() == 1 && rule.getSymbols().get(0).getType() == **'t'**) || rule.getSymbols().size() > 1){  
 GrammarString nalt = **new** GrammarString(**new** ArrayList<>(rule.getSymbols()));  
 **for**(String nprod : np){  
 Set<GrammarString> body = newP.get(nprod);  
 **if**(body != **null**){  
 body.add(nalt);  
 }  
 **else**{  
 body = **new** HashSet<>();  
 body.add(nalt);  
 newP.put(nprod,body);  
 }  
 }  
 }  
 }  
 }  
 **return new** Grammar(**this**.**T**,**this**.**N**,newP,**this**.**S**,**this**.**E**,**this**.**lex\_rules**);  
}

Листинг 23. Удаление цепных правил из неукорачивающей грамматики.

В выше описанной процедуре имеется одно важное допущение, а именно: в ней не проверяется, имеет ли грамматика eps-правила. Оно обосновано тем, что первый этап устранения левой рекурсии [4] состоит именно в удалении таких правил. Второй этап, который устраняет цепные правила из грамматики уже имеет на входе неукорачивающую грамматику, полученную на предыдущем этапе.

Циклом в грамматике *G* называется отношение вывода , вытекающее из правил граммитики для некоторого нетерминала *A.* Сам нетерминал является *циклическим.*

Очевидно, что в грамматике без – правил и цепных правил, а также без бесполезных символов невозможно вывести цикл.

Грамматика *G* называется *приведённой,* если в ней нет бесполезных символов, -правил, и циклов. Ниже приведён статический метод *deleteLeftRecursion*, принимающий на вход грамматику в виде объекта типа Grammar, и вовзращающий новую приведённую грамматику без левой рекурсии, полученную из исходной, с помощью устранения – правил и цепных правил, а также бесполезных символов и самой рекурсии. Перед тем, как начать устранять рекурсию, необходимо убедиться в том, что грамматика является приведённой, поскольку алгоритм работает именно с этими грамматиками. Метод реализует алгоритм 4.8 из [3] и алгоритм 2.13 из [2]. Для проверки того, что одна из альтернатив правила имеет непосредствен-ную левую рекурсию, используется метод *hasImmediateLeftRecursion.* Для выполнения шага замены правил , используется процедура *replaceAlternativesWithFirstN,* принимающая на вход альтернативы правила (*al1*), (al2), и имени нетерминала, подлежащего замене, т.е. . Для упорядочивания множества нетерминалов *N,* используется метод *getIndexedGrammar,* возвращающий новую грамматику, в которой всем нетерминалам добавлен номер в порядке его появления в выводимой цепочке, начиная со стартового символа грамматики *S*, который имеет новое имя *S\_1.* Если один и тот же нетерминал встречается несколько раз в правых частях различных правил, то он получит новый номер только при первой встрече в правой части какого-либо правила, в последующих же встречах, он будет заменяться на своё новое имя.

*//Get indexed grammar where each non-terminal is ordered. (by specifying unique number begining with start production S)  
//Example: S -> aSbS | bSaS | e => S1 -> aS1bS1 | bS1aS1 | e.***public** Grammar getIndexedGrammar(){  
 Map<String,String> nNames = **new** HashMap<>();  
 Map<String,Set<GrammarString>> newP = **new** HashMap<>();  
 Set<String> NN = **new** HashSet<>();  
 **int** idx = 2;  
 LinkedStack<String> pr = **new** LinkedStack<>();  
 pr.push(**this**.**S**);  
 NN.add(**this**.**S**+**"\_1"**);  
 nNames.put(**this**.**S**,**this**.**S**+**"\_1"**);  
 String start = **this**.**S**+**"\_1"**;  
 **while**(!pr.isEmpty()){  
 String p = pr.top();  
 String np = nNames.get(p);  
 pr.pop();  
 Set<GrammarString> alts = **P**.get(p);  
 Set<GrammarString> nalts = **new** HashSet<>();  
 **for**(GrammarString body : alts){  
 GrammarString nb = **new** GrammarString();  
 List<GrammarSymbol> symbols = body.getSymbols();  
 **for**(GrammarSymbol symbol : symbols){  
 **if**(symbol.getType() == **'t'**) {  
 nb.addSymbol(**new** GrammarSymbol(**'t'**,symbol.getVal()));  
 **continue**;  
 }  
 String name = nNames.get(symbol.getVal());  
 **if**(name != **null**){  
 nb.addSymbol(**new** GrammarSymbol(**'n'**,name));  
 }  
 **else**{  
 pr.push(symbol.getVal());  
 nNames.put(symbol.getVal(),symbol.getVal()+**"\_"**+idx);  
 nb.addSymbol(**new** GrammarSymbol(**'n'**,symbol.getVal()+**"\_"**+idx));  
 NN.add(symbol.getVal()+**"\_"**+idx);  
 idx++;  
 }  
 }  
 nalts.add(nb);  
 }  
 newP.put(np,nalts);  
 }  
 **return new** Grammar(**this**.**T**,NN,newP,start,**this**.**E**,**lex\_rules**);  
}

*//ELIMINATE LEFT RECURSION. (all type)***public static** Grammar deleteLeftRecursion(Grammar G){  
 G = G.deleteUselessSymbols();  
 G = G.getNonEmptyWordsGrammar();  
 G = G.getNonCycledGrammar();  
 G = G.getIndexedGrammar();  
 **int** n = G.getNonTerminals().size();  
 Map<String,Set<GrammarString>> newP = **new** HashMap<>();  
 Set<String> NN = **new** HashSet<>();  
 List<String> N = ColUtils.*fromSet*(G.**N**);  
 N.sort((x,y) -> {  
 **int** x\_i = Integer.*parseInt*(x.split(**"\_"**)[1]);  
 **int** y\_i = Integer.*parseInt*(y.split(**"\_"**)[1]);  
 **return** Integer.*compare*(x\_i, y\_i);  
 });  
 *//System.out.println(N);* **for**(**int** i = 1; i <= n; i++){  
 String Ai = N.get(i - 1);*//S\_1.* **for**(**int** j = 1; j < i; j++){  
 String Aj = N.get(j - 1);  
 Set<GrammarString> A\_j = G.**P**.get(Aj);  
 Set<GrammarString> A\_i = *replaceAlternativesWithFirstN*(G.**P**.get(Ai),A\_j,Aj);*//replace A\_i -> bA\_jb to A\_i -> bal1b|bal2b|bal3b where A\_j -> al1|al2|al3  
 //System.out.println(A\_i);* G.**P**.put(Ai,A\_i);*//add replaced rule A\_i -> bA\_jb.* }  
 Set<GrammarString> A\_i = G.**P**.get(Ai);  
 *//System.out.println(A\_i);* Set<GrammarString> rb1 = **new** HashSet<>();  
 Set<GrammarString> lb1 = **new** HashSet<>();  
 **boolean** flag = **false**;  
 **if**(!*hasImmediateLeftRecursion*(A\_i,Ai)){*//each alternative has no immediate left-recursion.* newP.put(Ai,A\_i);  
 NN.add(Ai);  
 **continue**;  
 }  
 **for**(GrammarString str : A\_i){  
 **int** k = 0;  
 List<GrammarSymbol> symbols = str.getSymbols();  
 **if**(symbols.size() > 1 && symbols.get(0).getVal().equals(Ai)){*//immediate recursion.* GrammarString g = **new** GrammarString(**new** ArrayList<>(str.getSymbols().subList(1,symbols.size())));*//do not add first symbol.* g.addSymbol(**new** GrammarSymbol(**'n'**,Ai+**"\'"**));*//add new symbol.* lb1.add(g);*//add new sequence alpha A' for A -> A alpha. => A -> alpha A'* lb1.add(**new** GrammarString(**new** ArrayList<>(str.getSymbols().subList(1,symbols.size()))));*//add alpha sequence => A -> alpha A' | alpha* flag = **true**;  
 }  
 **else** { *//there is a beta sequence.* GrammarString g = **new** GrammarString(**new** ArrayList<>(str.getSymbols()));  
 g.addSymbol(**new** GrammarSymbol(**'n'**,Ai+**"\'"**));  
 rb1.add(g);*//add new sequence beta A' for A -> beta WHERE beta doesn't begin with A.* rb1.add(str);*//add old sequence beta* }  
 }  
 newP.put(Ai,rb1);  
 NN.add(Ai);  
 newP.put(Ai+**"\'"**,lb1);  
 NN.add(Ai+**"\'"**);  
 }  
 **return new** Grammar(G.**T**,NN,newP,G.getStart(),G.**E**,G.getLexicalRules());  
}  
  
  
*//Replace for all alternatives in al1 first symbol with name N (non-term) with alternatives contain body of N.  
//If N also consists of several alternatives then add them as new alternatives.  
//Example A -> N, beta, N -> y1 | y2 ... | yn  
// => A -> y1, beta | y2, beta | ... | yn, beta.***private static** Set<GrammarString> replaceAlternativesWithFirstN(Set<GrammarString> al1, Set<GrammarString> al2,String N){  
 List<GrammarString> it = ColUtils.*fromSet*(al2);  
 Set<GrammarString> result = **new** HashSet<>();  
 **for**(GrammarString s :al1) {  
 GrammarString r = **new** GrammarString();  
 **if**(s.getSymbols().get(0).getVal().equals(N)){  
 List<GrammarSymbol> suffix = s.getSymbols().subList(1,s.getSymbols().size());  
 **int** k = 0;  
 **while**(k < it.size()){  
 r.getSymbols().addAll(it.get(k).getSymbols());  
 r.getSymbols().addAll(suffix);  
 result.add(**new** GrammarString(**new** ArrayList<>(r.getSymbols())));*//S\_1d -> a,b,A\_2a* r = **new** GrammarString();*//only copy! not reference!* k++;  
 }  
 }  
 **else**{  
 r.getSymbols().addAll(s.getSymbols());  
 result.add(r);  
 }  
 }  
 **return** result;  
}

**private static boolean** hasImmediateLeftRecursion(Set<GrammarString> s,String P){  
 **for**(GrammarString body : s){  
 **if**(body.getSymbols().get(0).getVal().equals(P))  
 **return true**;  
 }  
 **return false**;  
}

Листинг 24. Устранение левой рекурсии из грамматики.

Метод *deleteLeftFactor* производит левую факторизацию грамматики. Он реализует алгоритм 4.10 из [3]. Для каждого правила, он находит сначала множество альтернатив, не имеющих общих префиксов с помощью метода nonPreffix. Затем, полученное множество вычитается из множества всех альтернатив правила. Получив множество альтернатив, в котором есть хотя бы две альтернативы, имеющие общий префикс, процедура *commonPreffix* вычисляет его и вовзращает. Далее, если этот преффикс , то для каждой альтернативы правила A, строится множество новых альтернатив (*bodyOfnewTerm)* для нового правида A’, вычитая из предыдущих альтернатив преффикс . Если же альтернатива не имела этот преффикс, то она добавляется в множество альтернатив для старого правила A (*nonPref*), которое надо изменить следующим образом: . Новое же правило примет вид: .

Для нескольких пар альтернатив, имеющих разные общие преффиксы, например, , алгоритм вернёт первый возможный преффикс , затем он построит множество альтернатив нового правила , а в множество альтернатив старого правила добавит и . Поскольку в старом правиле была устранена лишь одна пара, имеющая общий преффикс, то правило необходимо снова преобразовать, но на этот раз он добавит правило , а правило А в итоге будет выглядеть следующим образом: . В случае, когда дано правило , то правило т.е. имеют одинаковые правые части. Тогда надо выбрать правило с именем наименьшей длины ( и заменить все вхождения на , удалив после этого правило из множества правил *P* и нетерминалов *N*. Для получения минимально возможной длины альтернативы правила используется процедура *commonMinLength*.

**public** Grammar deleteLeftFactor(){  
 Map<String,Set<GrammarString>> newP = **new** HashMap<>(**this**.**P**);  
 Set<String> NN = **new** HashSet<>();  
 LinkedStack<String> rules = **new** LinkedStack<>();  
 Map<String,Integer> newRidxs = **new** HashMap<>();  
 Set<String> keys = newP.keySet();  
 **for**(String k : keys) {  
 **if** (!k.equals(**this**.**S**))  
 rules.push(k);  
 newRidxs.put(k,1);  
 }  
 rules.push(**this**.**S**);  
 **boolean** hasEmpty = **false**;  
 *//int k = 0;* **while**(!rules.isEmpty()){  
 String ph = rules.top();  
 rules.pop();  
 Set<GrammarString> alts = **new** HashSet<>(newP.get(ph));  
 Set<GrammarString> nonPref = nonPreffix(alts);  
*// System.out.println("P: "+alts);* alts.removeAll(nonPref);  
*// System.out.println("Y: "+alts);* GrammarString preffix = **null**;  
 **if**(alts.size() > 0)  
 preffix = commonPreffix(alts);  
 **else**{  
 NN.add(ph);  
 **continue**;  
 }  
*// System.out.println("Preffix: "+preffix);* **if**(preffix.getSymbols().size() > 1 || (preffix.getSymbols().size() == 1 && !preffix.getSymbols().get(0).getVal().equals(**this**.**E**))) {  
 **int** pL = preffix.getSymbols().size();  
 Set<GrammarString> bodyOfnewTerm = **new** HashSet<>();  
 **for**(GrammarString r : alts){  
 List<GrammarSymbol> l = r.getSymbols();  
 **if**(l.subList(0,pL).equals(preffix.getSymbols())) {  
 **if** (pL == l.size()) {  
 GrammarString empty = **new** GrammarString();  
 empty.addSymbol(**new** GrammarSymbol(**'t'**, **this**.**E**));  
 bodyOfnewTerm.add(empty);  
 hasEmpty = **true**;  
 } **else** bodyOfnewTerm.add(**new** GrammarString(**new** ArrayList<>(l.subList(pL,l.size()))));  
 }  
 **else**{  
 nonPref.add(**new** GrammarString(**new** ArrayList<>(r.getSymbols())));  
 }  
 }  
 GrammarString prefRule = **new** GrammarString(**new** ArrayList<>(preffix.getSymbols()));  
 **int** id = newRidxs.get(ph);  
 String nName = ph+**"\'"**+id;  
 prefRule.addSymbol(**new** GrammarSymbol(**'n'**,nName));  
 nonPref.add(prefRule);  
  
*// System.out.println("nonPref: "+nonPref);  
// System.out.println("pref: "+bodyOfnewTerm);* newP.put(ph,nonPref);  
 newP.put(nName,bodyOfnewTerm);  
 id++;  
 newRidxs.put(ph,id);  
 newRidxs.put(nName,1);  
 rules.push(ph);  
 rules.push(nName);  
 NN.add(nName);  
 }  
 NN.add(ph);  
 }  
 **if**(hasEmpty){  
 HashSet<String> NT = **new** HashSet<>(**this**.**T**);  
 NT.add(**this**.**E**);  
 NN.addAll(**N**);  
 Grammar NG = **new** Grammar(NT,NN,newP,**this**.**S**,**this**.**E**,**lex\_rules**);  
 NG = NG.getGrammarWithoutEqualRules();  
 *//System.out.println(NG);* **return** NG.getNonCycledGrammar().deleteUselessSymbols();

}  
 NN.addAll(**N**);

Grammar NG = **new** Grammar(NT,NN,newP,**this**.**S**,**this**.**E**,**lex\_rules**);  
 NG = NG.getGrammarWithoutEqualRules();  
 *//System.out.println(NG);* **return** NG.getNonCycledGrammar().deleteUselessSymbols();

}  
  
 **private int** commonMinLength(Set<GrammarString> prod){  
 List<GrammarString> rules = ColUtils.*fromSet*(prod);  
 **int** minL = rules.get(0).getSymbols().size();  
 *//Find minimum length of the grammarStr.* **for**(**int** i = 1; i < rules.size();i++){  
 **if**(rules.get(i).getSymbols().size() < minL)  
 minL = rules.get(i).getSymbols().size();  
 }  
 **return** minL;  
 }  
  
  
 *//works only with set of alternatives which have at least pair with common preffix  
 //Example: [abc, acb, bac, bcd] -> commonPreffixes: a for [abc,acb], b for [bac,bcd].  
 //It stops when it finds the first commonPreffix. (in previous example it stops after find a)* **public** Set<GrammarString> nonPreffix(Set<GrammarString> prod){  
 Set<GrammarString> nonF = **new** HashSet<>();  
 List<GrammarString> rules = ColUtils.*fromSet*(prod);  
 **for**(**int** i = 0; i < rules.size(); i++){  
 **boolean** flag = **true**;  
 GrammarSymbol si = rules.get(i).getSymbols().get(0);  
 **for**(**int** j = 0; j < rules.size();j++){  
 **if**(j == i)  
 **continue**;  
 GrammarSymbol sj = rules.get(j).getSymbols().get(0);  
 **if**(si.getVal().equals(sj.getVal())) {  
 flag = **false**;  
 **break**;  
 }  
 }  
 **if**(flag){  
 nonF.add(**new** GrammarString(**new** ArrayList<>(rules.get(i).getSymbols())));  
 }  
 }  
 **return** nonF;  
 }  
  
 **public** GrammarString commonPreffix(Set<GrammarString> prod){  
 List<GrammarString> r = **new** ArrayList<>();  
 List<GrammarString> rules = ColUtils.*fromSet*(prod);  
 **int** minL = commonMinLength(prod);  
 rules.sort((x,y) -> {  
 **return** Integer.*compare*(x.getSymbols().size(), y.getSymbols().size());  
 });  
 **boolean** flag = **true**;  
 **int** ca = 1;  
 **int** i = 0;  
 GrammarString current = **null**;  
 GrammarString preffix = **new** GrammarString();  
 **while**(flag && i < minL){  
 ca = 1;  
 current = **new** GrammarString(rules.get(0).getSymbols().subList(0,i+1));  
 **for**(**int** j = 1; j < rules.size();j++){  
 GrammarString c = **new** GrammarString(rules.get(j).getSymbols().subList(0,i+1));  
 **if**(c.equals(current)){  
 ca++;  
 }  
 }  
 **if**(ca > 1)  
 preffix = current;  
 **else** flag = **false**;  
 i++;  
 }  
 **return** preffix;  
 }

Листинг 25. Левая факторизация.

Как уже было сказано, при наличии нескольких обших преффиксов (т.е. нескольких множеств альтернатив, имеющих общий преффикс, новые правила могут стать эквивалентны друг другу из-за наличия общего суффикса у двух или более альтернатив. Ниже дан код метода *getGrammarWithoutEqualRules*, вовзращающий новую грамматику, в которой удалены такие лишние, одинаковые правила.

*//Given production with name n.  
//Produce list of the productions names which are equal to production n.* **private** List<String> equalRules(String n){  
 List<String> altNames = **new** LinkedList<>();  
 Set<String> ps = **P**.keySet();  
 Set<GrammarString> r1 = **P**.get(n);  
 altNames.add(n);  
 **for**(String p : ps){  
 **if**(p.equals(n))  
 **continue**;  
 Set<GrammarString> r2 = **P**.get(p);  
 **if**(r1.equals(r2))  
 altNames.add(p);  
 }  
 **return** altNames;  
 }  
  
 **public** Grammar getGrammarWithoutEqualRules(){  
 Map<String,Set<GrammarString>> newP = **new** HashMap<>();  
 Set<String> NN = **new** HashSet<>();  
 Set<String> ps = **P**.keySet().stream().sorted((x,y) -> {  
 **return** Integer.*compare*(y.length(), x.length());}).collect(Collectors.*toSet*());  
 Map<String,String> names = **new** HashMap<>();  
 ArrayList<String> mapped = **new** ArrayList<>();  
 **for**(String p : ps) {  
 List<String> l = equalRules(p);  
 **if**(l.size() > 0){  
 String s = l.stream().sorted((x,y) -> {  
 **return** Integer.*compare*(x.length(), y.length());  
 }).collect(Collectors.*toList*()).get(0);  
*// System.out.println(p+": "+l+" "+s);* **if**(!mapped.contains(p)){  
 **for**(String alts : l){  
 names.put(alts,s);  
 mapped.add(alts);  
 }  
 names.put(p,s);  
 mapped.add(p);  
 }  
 }  
 **else** {  
 names.put(p, p);  
 mapped.add(p);  
 }  
 }  
 String start = **null**;  
 **for**(String k : names.keySet()){  
 NN.add(names.get(k));  
 **if**(k.equals(**this**.**S**))  
 start = names.get(k);  
 }  
 **for**(String prod : NN){  
 Set<GrammarString> alts = **P**.get(prod);  
 Set<GrammarString> nalts = **new** HashSet<>();  
 **for**(GrammarString b : alts){  
 GrammarString nb = **new** GrammarString();  
 **for**(GrammarSymbol s : b.getSymbols()){  
 GrammarSymbol ns = **null**;  
 **if**(s.getType() == **'t'**)  
 ns = **new** GrammarSymbol(s.getType(),s.getVal());  
 **else** ns = **new** GrammarSymbol(s.getType(),names.get(s.getVal()));  
 nb.addSymbol(ns);  
 }  
 nalts.add(nb);  
 }  
 newP.put(prod,nalts);  
 }  
 **return new** Grammar(**T**,NN,newP,start,**this**.**E**,**lex\_rules**).deleteUselessSymbols();  
 }

Листинг 26. Устранение одинаковых правил, вызванные наличием общего суффикса у двух или более альтернатив при выполнении процедуры левой факторизации.

**4. Результаты работы программы.**

Алгоритм левой рекурсии порождает новые нетерминалы, которые получают имена оканчивающимися на одиночный апостроф *’*. Для левой факторизации имена новых нетерминалов заканчиваются апострофом, за которым следует цифра, например *’*1.

В качестве тестов на устранение -правил были выбраны грамматики из примера 2.23 и упражнения 2.4.11 [2].

Для грамматики из примера 2.23 множество правил *P*

преобразуется в *P’*

Новым стартовым символом станет *S’, а N’ = .*

Для упражнения 2.4.11, данное множество правил:

преобразуется в:

.

Новым стартовым символом станет *S’, а N’ = .* Так как нетерминалы *A,B,C* могут порождать пустые строки, то и строка ABC – может быть пустой, следовательно, продукция может быть заменена на пустую строку, т.е. .

Если для примера 2.23 сначала устранить длинные правила из грамматики, то получим след. грамматику G’ = (T = {a,b,e}, N’ = {S,S1,S2,S3,S4}, P ={

S -> bS3|e|aS1

S3 -> SS4

S4 -> aS

S1 -> SS2

S2 -> bS

},S);

После устранение из неё eps-правил, получим

S' -> S|e

S3 -> S4|SS4

S4 -> a|aS

S -> bS3|aS1

S1 -> SS2|S2

S2 -> bS|b,

Где S’ – новый стартовый символ грамматики, N’ = {S,S1,S2,S3,S4,S’}. В старом стартовом символе грамматики было eps-правило, поэтому новое правило S’ было добавлено. Так как в каждом правиле (кроме правил, с заголовком S) содержится ровно один нетерминал из N\_e (т.е. такой, который порождает пустые строки), то для таких правил множество новых альтернатив, будет содержать ровно один новый элемент, а именно: альтернативу без этого нетерминала. В итоге, к каждому правилу (кроме S), добавится по одной новой альтернативе.

Для грамматики с правилами:

S -> ABCd

A -> a|e

B -> AC

C -> c|e,

после удаления eps-правил получим:

S -> Cd|Bd|ACd|BCd|ABd|Ad|d|ABCd

A -> a

B -> A|AC|C

C -> c,

так, как S не порождает пустых строк, то стартовый символ не меняется (N\_e = {A,B,C}.) Если её сначала сократить (устранить длинные правила), а потом удалить eps-правила, то получим:

S -> AS1,

S1 -> BS2

S2 -> Cd

A -> a|e

B -> AC

C -> c|e,

=>

S -> S1|AS1

S1 -> BS2|S2

S2 -> Cd|d

A -> a

B -> A|AC|C

C -> c.

После сокращения получили два новых нетерминала S1, S2, и все правила, кроме A,B,C, стали содержать ровно по одному нетерминалу из множества N\_e = {A,B,C} (B – состоит ровно из двух нетерминалов A,C).

Грамматика из упражнения 2.4.11 [2] содержит ровно одно длинное правило: , устранив которое, мы получим два правила: и один новый нетерминал S1. После удаления eps-правил из полученной грамматики с короткими правилами её множество P’ примет вид:

S' -> S|e

A -> BB|B

B -> CC|a|C

S -> A|S1|AS1

C -> AA|A|b

S1 -> BC|B|C

Результаты тестов, определённых в классе TestNonEmpty удовлетворяют всему вышесказанному:

**package** ru.osipov.labs.lab2;  
  
**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.TestDFA;  
**import** ru.osipov.labs.lab2.grammars.Grammar;  
**import** ru.osipov.labs.lab2.jsonParser.SimpleJsonParser;  
**import** ru.osipov.labs.lab2.jsonParser.jsElements.JsonObject;  
  
@ExtendWith(SpringExtension.**class**)  
@RunWith(SpringJUnit4ClassRunner.**class**)  
@ContextConfiguration(classes= TestNonEmpty.**class**)  
**public class** TestNonEmpty {  
 @Test  
 **public void** test1(){  
 String p = System.*getProperty*(**"user.dir"**);  
 System.***out***.println(p);  
 p = p+**"\\src\\test\\java\\ru\\osipov\\labs\\lab2\\"**;  
 p = p+**"grammarJson\\G\_2\_23.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);  
 **int** N = G.getNonTerminals().size();  
 G = G.getNonEmptyWordsGrammar();  
 System.***out***.println(**"Target"**);  
 System.***out***.println(G);  
 **assert** G != **null** && N == G.getNonTerminals().size() - 1;  
 **assert** G.getProductions().get(**"S"**).size() == 8;  
 **assert** G.getProductions().containsKey(**"S\'"**);  
 }  
  
 *//G\_2\_23.json :  
 //Source: 3 rules (including alternatives) 3 terms, 1 non-term.  
 //Source -> after deleting eps -> 10 rules (2 non-terms) -> after shorting 18 rules (10 non-terms)  
 //Source -> after shorting 7 rules (5 non-terms) -> after deleting eps 12 rules (6 non-terms)  
 //N\_e : [S] after deleting eps -> [S']* @Test  
 **public void** test2(){  
 String p = System.*getProperty*(**"user.dir"**);  
 System.***out***.println(p);  
 p = p+**"\\src\\test\\java\\ru\\osipov\\labs\\lab2\\"**;  
 p = p+**"grammarJson\\G\_2\_23.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);  
 **int** N = G.getNonTerminals().size();  
 G = G.getShortenedGrammar().getNonEmptyWordsGrammar();  
 System.***out***.println(**"Target"**);  
 System.***out***.println(G);  
 **assert** G != **null** && N == G.getNonTerminals().size() - 5;  
 **assert** G.getProductions().get(**"S"**).size() == 2;  
 **assert** G.getProductions().containsKey(**"S\'"**);  
 }  
  
 @Test  
 **public void** test3(){  
 String p = System.*getProperty*(**"user.dir"**);  
 System.***out***.println(p);  
 p = p+**"\\src\\test\\java\\ru\\osipov\\labs\\lab2\\"**;  
 p = p+**"grammarJson\\G\_W\_E1.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);  
 **int** N = G.getNonTerminals().size();  
 G = G.getNonEmptyWordsGrammar();  
 System.***out***.println(**"Target"**);  
 System.***out***.println(G);  
 **assert** G != **null** && N == G.getNonTerminals().size();  
 **assert** G != **null** && G.getProductions().get(**"S"**).size() == 8;  
 }  
  
 *//G\_W\_E1.json :  
 //Source: 6 rules. (including alternatives) 4 terms, 4 non-terms.  
 //rules 6, after shortening -> 8 (6 non-terms), after deleting eps -> 11 rules. (6 non-terms)  
 //rules 6, after deleting eps -> 13 (4 non-terms), after shortening -> 18 rules. (9 non-terms)  
 //N\_e: [A,B,C] after deleting eps -> [].* @Test  
 **public void** test4(){  
 String p = System.*getProperty*(**"user.dir"**);  
 System.***out***.println(p);  
 p = p+**"\\src\\test\\java\\ru\\osipov\\labs\\lab2\\"**;  
 p = p+**"grammarJson\\G\_W\_E1.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);  
 **int** N = G.getNonTerminals().size();  
 G = G.getShortenedGrammar();  
 System.***out***.println(G);  
 G = G.getNonEmptyWordsGrammar();  
 System.***out***.println(**"Target"**);  
 System.***out***.println(G);  
 }  
  
 @Test  
 **public void** test5(){  
 String p = System.*getProperty*(**"user.dir"**);  
 System.***out***.println(p);  
 p = p+**"\\src\\test\\java\\ru\\osipov\\labs\\lab2\\"**;  
 p = p+**"grammarJson\\G\_2\_4\_11.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);  
 **int** N = G.getNonTerminals().size();  
 G = G.getNonEmptyWordsGrammar();  
 System.***out***.println(**"Target"**);  
 System.***out***.println(G);  
 **assert** G != **null** && G.getNonTerminals().size() - 1 == N;  
 **assert** G.getProductions().get(**"S"**).size() == 7;  
 **assert** G.getProductions().containsKey(**"S\'"**);  
 }  
  
 *//G\_2\_4\_11.json :  
 //Source : 7 rules (including alternatives) 3 terms, 4 non-terms.  
 //Source -> after deleting eps -> 17 rules (5 non-terms), -> after shorting 18 rules (6 non-terms)  
 //Source -> after shorting 8 rules (5 non-terms) -> after deleting eps 16 rules (6 non-terms).  
 //N\_e: [A,B,S,C] after deleting eps -> [S']* @Test  
 **public void** test6(){  
 String p = System.*getProperty*(**"user.dir"**);  
 System.***out***.println(p);  
 p = p+**"\\src\\test\\java\\ru\\osipov\\labs\\lab2\\"**;  
 p = p+**"grammarJson\\G\_2\_4\_11.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);  
 **int** N = G.getNonTerminals().size();  
 G = G.getShortenedGrammar();  
 System.***out***.println(G);  
 G = G.getNonEmptyWordsGrammar();  
 System.***out***.println(**"Target"**);  
 System.***out***.println(G);  
 **assert** G != **null** && N == G.getNonTerminals().size() - 2;  
 **assert** G.getProductions().get(**"S"**).size() == 3;  
 **assert** G.getProductions().containsKey(**"S\'"**);  
 }  
}

Листинг 27. Тесты для реализации алгоритма 2.10. (устранение eps-правил).

Для устранения левой рекурсии используется грамматика из примера 2.27 [2]. Так как это грамматика не содержит eps-правил и циклов, поэтому, не удаляя цепных правил, получим из множества P = {

}

P’ равное :

N’ = {E\_1, E\_1’, T\_2, T\_2’, F\_3}, S = E\_1, T = {+,\*,(,**id**,)} *G\* = (T,N’,P’,S);*

Здесь, правила упорядочены относительно их появления в выводимой цепочке, начиная со стартового символа. Как видно, у четырёх продукций имеется общий префикс у двух альтернатив. (T\_2, F\_2, +T\_2, \*F\_3 для продукций E\_1, E\_1’, T\_2, T\_2’).

Как видно, если есть цепные и eps-правила в грамматике, то в ней возможно присутствие циклов, но они не обязательно будут присутствовать в грамматике, как показано на примере 2.27 из [2].

Если удалить из грамматики eps-правила и цепные правила, то в ней гарантированно не будет циклов. Например, для грамматики из примера 2.27, грамматика без eps-правил и цепных правил будет иметь следующее множество продукций *P’* =

E -> T\*F|**id**|(E)|E+T

T -> T\*F|id|(E)

F -> **id**|(E).

Применив алгоритм устранения левой рекурсий к данной грамматике, получим:

E\_1 -> **id**E\_1'|(E\_1)E\_1'|T\_2\*F\_3|**id** |T\_2\*F\_3E\_1'|(E\_1)

E\_1' -> +T\_2E\_1'|+T\_2

T\_2 -> (E\_1)T\_2'|**id** |**id**T\_2'|(E\_1)

T\_2' -> \*F\_3|\*F\_3T\_2'

F\_3 -> id|(E\_1)

Как видно, после удаления цепных правил, множество цепочек , которые не начинаются с левой части продукции (т.е. с заголовка продукции, к которой они относятся) увеличивается.

Рассмотрим работу алгоритма левой факторизации на примере полученной выше нелеворекурсивной грамматики *G\**.

На первом шаге, у нетерминала E\_1 будет обнаружен преффикс у двух альтернатив, равный T\_2. Поскольку других альтернатив у E\_1 нет, то получим новые правила: E\_1 -> T\_2E\_1’1, где E\_1’1 -> E\_1’ | eps. (Здесь, цифра после апострофа указывает на текущий преффикс, для двух или более альтернатив, в том смысле, что могут быть альтернативы, которые могут иметь несколько общих преффиксов, т.е. несколько подмножеств из множества альтернатив, имеющих свой общий преффикс). Далее у нетерминала E\_1’ также обнаружится общий преффикс: +T\_2. От него получим правила: E\_1’ -> +T\_2E\_1”1, E\_1”1 -> E\_1’ | eps. В итоге, нетерминалы E\_1’1 и E\_1”1 порождают одни и те же продукции, следовательно из них можно выбрать нетерминал с более коротким именем E\_1’1. Далее, для нетерминалов T\_2 и T\_2’ ситуация повторяется.

T\_2 -> F\_3T\_2’1,

T\_2’1 -> T\_2’ | eps,

T\_2’ -> \*F\_3T\_2”1,

T\_2”1 -> T\_2’ | eps.

Здесь, у нетерминалов T\_2’1 и T\_2”1 общие продукции, поэтому можно выбрать нетерминал T\_2’1. Заменив E\_2”1 на E\_2’1 и T\_2”1 на T\_2’1 во всех правых частях продукци, получим

E\_1 -> T\_2E\_1’1,

E\_1’1 -> E\_1’ | eps,

E\_1’ -> +T\_2E\_1’1,

T\_2 -> F\_3T\_2’1,

T\_2’1 -> T\_2’ | eps,

T\_2’ -> \*F\_3T\_2’1.

Заметим, что здесь есть цепные правила: T\_2’1 -> T\_2’ и E\_1’1 -> E\_1’. Устранив их, мы получим недостижимые символы E\_1’ и T\_1’, и в итоге нетерминалы E\_1’1 и T\_1’1 будут содержать продукции из E\_1’ и T\_1’ т.е. они как бы наследуются от них. Что касается нетерминала F\_3, то его альтернативы не имеют общих преффиксов, поэтому он добавляется в множество новых правил без изменений. Удалив бесполезные символы мы окончательно получим:

E\_1 -> T\_2E\_1’1,

E\_1’1 -> +T\_2E\_1’1 | eps,

T\_2 -> F\_3T\_2’1,

T\_2’1 -> \*F\_3T\_2’1 | eps,

F\_3 -> (E\_1) | **id**.

Что касается грамматики из примера 2.27 [2], в которой сначала были удалены цепные правила, а затем была удалена левая рекурсия, то применив левую факторизацию получим следующее множество правил новой грамматики :

E\_1 -> (E\_1)E\_1'3|T\_2\*F\_3E\_1'2|idE\_1'1

E\_1' -> +T\_2E\_1''1

T\_2''1 -> e|T\_2'

E\_1''1 -> E\_1'|e

T\_2 -> (E\_1)T\_2'2|idT\_2'1

T\_2' -> \*F\_3T\_2''1

E\_1'1 -> E\_1'|e

T\_2'2 -> e|T\_2'

E\_1'2 -> E\_1'|e

T\_2'1 -> e|T\_2'

F\_3 -> id|(E\_1)

E\_1'3 -> E\_1'|e.

Если же сначала удалить левую рекурсию (т.е. получить грамматику *G\**), а затем устранить цепные правила, то получим грамматику *G\*\*,* с множеством правил:

E\_1 -> id|F\_3T\_2'|T\_2E\_1'|(E\_1)

E\_1' -> +T\_2E\_1'|+T\_2

T\_2 -> id|F\_3T\_2'|(E\_1)

T\_2' -> \*F\_3|\*F\_3T\_2'

F\_3 -> id|(E\_1).

Применив к ней левую факторизацию, множество правил преобразуется в:

E\_1 -> id|F\_3T\_2'|T\_2E\_1'|(E\_1)

E\_1' -> +T\_2E\_1''1

T\_2''1 -> e|T\_2'

E\_1''1 -> E\_1'|e

T\_2 -> id|F\_3T\_2'|(E\_1)

T\_2' -> \*F\_3T\_2''1

F\_3 -> id|(E\_1).

Заметим, что полученное множество имеет меньший размер, чем у того множества, грамматика которого сначала была преобразованна к *приведённой форме,* перед устранением левой рекурсии и левой факторизации.

Метод, реализующий алгоритм 4.10 из [3], сначала выполняет левую факторизацию, затем с помощью процедуры *getNonCycledGrammar* устраняет цепные правила, после этого, вызвав *getGrammarWithoutEqualRules* у грамматики без цепных правил, он получает грамматику без нетерминалов, которые содержали бы одно и то же множество продукций (альтернатив), и без бесполезных символов.

Для грамматики *G\*\** эти процедуры возвращают множество правил *P\_nc’ =*

E\_1 -> id|F\_3T\_2'|T\_2E\_1'|(E\_1)

E\_1' -> +T\_2E\_1''1

T\_2''1 -> \*F\_3T\_2''1|e

E\_1''1 -> e|+T\_2E\_1''1

T\_2 -> id|F\_3T\_2'|(E\_1)

T\_2' -> \*F\_3T\_2''1

F\_3 -> id|(E\_1) .

Для грамматики будет получено:

E\_1 -> T\_2\*F\_3E\_1'1|(E\_1)E\_1'1|idE\_1'1

T\_2 -> (E\_1)T\_2'2|idT\_2'2

E\_1'1 -> e|+T\_2E\_1'1

T\_2'2 -> e|\*F\_3T\_2'2

F\_3 -> id|(E\_1)

Для грамматики G\* будет:

E\_1 -> T\_2E\_1'1

T\_2 -> F\_3T\_2'1

E\_1'1 -> e|+T\_2E\_1'1

T\_2'1 -> e|\*F\_3T\_2'1

F\_3 -> id|(E\_1).

Очевидно, что для грамматики без цепных правил, результирующее множество правил может быть большим, чем для грамматики без eps-правил и циклов, но содержащее цепные правила (и не являющаяся *приведённой* по определению). Поэтому, перед тем, как удалять цепные правила из грамматики, необходимо выяснить, есть ли в ней циклы (т.е. возвожно отношение вида Если циклов нет, то нет необходимости устранять цепные правила перед левой рекурсией и левой факторизацией. Для этого необходимо вычислить множество цепных правил *N\_c*. Затем для каждого нетерминала из данного множества проверить, не имеет ли место правило вида С этой целью, определим два метода *getAllChainedNonTerms,* возвращающий множество нетерминалов, которые являются заголовками цепных правил, и *hasCycles,* вычисляющий не порождает ли нетерминал из множества цепных правил сам себя. Ниже дан код этих методов:

**public** Set<String> getAllChainedNonTerms(){  
 Set<String> r = **new** HashSet<>();  
 Set<String> ps = **P**.keySet();  
 **for**(String p : ps){  
 Set<GrammarString> alts = **P**.get(p);  
 **for**(GrammarString rule : alts){  
 **if**(rule.getSymbols().size() == 1 && rule.getSymbols().get(0).getType() != **'t'**)  
 r.add(p);  
 }  
 }  
 **return** r;  
}  
  
**public boolean** hasCycles(){  
 Set<String> C = getAllChainedNonTerms();  
 **for**(String r : C){  
 **if**(compute\_allGeneratedN(r,**true**).contains(r))  
 **return true**;  
 }  
 **return false**;  
}

Листинг 28. Проверка того, что грамматика не имеет циклов.

Метод *compute\_allGeneratedN* модифицирован следующим образом.

В нём объявлен второй аргумент *isCicled,* который принимает значение *true*, указывающее выполнить первый шаг перед вычислением множества , а именно выполнить шаг 1.б из алгоритма 2.11 положив где *N* – первый аргумент метода *compute\_allGeneratedN,* т.е. нетерминал *A*, из шага 1.а алгоритма 2.11.

Во всех остальных случаях, кроме вызова внутри метода *hasCycles,* параметр *isCicled* принимает значение false, т.е. работает согласно алгоритму 2.11 с допущением, что грамматика уже не содержит eps-правил.

**public** Set<String> compute\_allGeneratedN(String N,**boolean** isCicled){

…

**if**(isCicled){  
 Set<GrammarString> pr1 = **P**.get(N);  
 **for** (GrammarString alpha : pr1) {*//for each alternative* **if** (alpha.getSymbols().size() == 1 && alpha.getSymbols().get(0).getType() != **'t'**)  
 V\_0.add(alpha.getSymbols().get(0).getVal());  
 }  
 }  
 **else** {  
 V\_0.add(N);  
 }

…

}

Листинг 29. Модификация метода *compute\_allGeneratedN.*

Теперь, в методе *deleteLeftRecursion* можно заменить строчку *G = G.getNonCycledGrammar(),* строчкой: *if(G.hasCycles()){G = G.getNonCycledGrammar();}*

Результаты тестов, удовлетворяют требованию минимизации правил грамматики.

**package** ru.osipov.labs.lab2;  
**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.TestDFA;  
**import** ru.osipov.labs.lab2.grammars.Grammar;  
**import** ru.osipov.labs.lab2.jsonParser.SimpleJsonParser;  
**import** ru.osipov.labs.lab2.jsonParser.jsElements.JsonObject;  
  
@ExtendWith(SpringExtension.**class**)  
@RunWith(SpringJUnit4ClassRunner.**class**)  
@ContextConfiguration(classes= TestLeftRecursion.**class**)  
**public class** TestLeftRecursion {  
 @Test  
 **public void** testRec1() {  
 String p = System.*getProperty*(**"user.dir"**);  
 System.***out***.println(p);  
 p = p+**"\\src\\test\\java\\ru\\osipov\\labs\\lab2\\"**;  
 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);  
 **int** N1 = G.getNonTerminals().size();  
 G = Grammar.*deleteLeftRecursion*(G);  
 System.***out***.println(**"Target"**);*//Step 1: A -> aAA'1, | aBc, A'1 -> B | c, Step 2: A -> aA'2, A'2 -> AA'1 | Bc.* System.***out***.println(G);  
 **assert** G.getN\_e().size() == 0 && G.getNonTerminals().size() == N1 + 2;  
 }  
  
 @Test  
 **public void** testRec2(){  
 String p = System.*getProperty*(**"user.dir"**);  
 System.***out***.println(p);  
 p = p+**"\\src\\test\\java\\ru\\osipov\\labs\\lab2\\"**;  
 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);  
 **int** N1 = G.getNonTerminals().size();  
 G = G.getNonCycledGrammar();  
 **int** N2 = G.getNonTerminals().size();  
 System.***out***.println(**"G without chained rules"**);  
 System.***out***.println(G);  
 System.***out***.println(**"Target"**);  
 G = Grammar.*deleteLeftRecursion*(G);  
 System.***out***.println(G);  
 **int** N3 = G.getNonTerminals().size();  
 G = G.deleteLeftFactor();  
 System.***out***.println(**"Delete preffix"**);  
 System.***out***.println(G);  
 **int** N4 = G.getNonTerminals().size();  
 System.***out***.println(N1+**" : "**+N2+**" : "**+N3+**" : "**+N4);  
 }  
}

Листинг 30. Класс тестов устранения левой рекурсии.

Как видно, сама левая факторизация может порождать цепные правила и eps-правила в грамматике из исходной грамматики. Рассмотрим несколько тестовых примеров. Для них будем выполнять только левую факторизацию, т.е. не вызывать методы *getNonCycledGrammar ,getGrammarWithoutEqualRules,* в целях тестирования самого алгоритма.

Пример 4.1

Для грамматики G = (T = {a,b,eps}, N = {A,B} *P* = {

A -> aAc|aAB|aBc }, A), получим *P*’ равное

A -> aA'2

A'1 -> B|c

A'2 -> Bc|AA'1.

На первом шаге для альтернатив aAc|aAB|aBc найдётся общий преффикс aA для двух альтернатив: aAc | aAB. В итоге, получим

A -> aAA’1 | aBc.

A’1 -> B | c.

На втором шаге, нетерминал A’1 не имеет общих преффиков, но у нетерминала A остался один общий преффикс между двумя альтернативами – a. Преобразовав снова правило А, получим:

A -> aA’2,

A’2 -> AA’1 | Bc,

A’1 -> B | c.

В итоге все правые части не содержат общие преффиксы и алгоритм завершает работу.

Пример 4.2.

Для грамматики с правилами:

S -> a|iEtSeS|iEtS

E -> b,

получим:

S -> a|iEtSS'1

E -> b

S'1 -> eps|eS.

Здесь, у правила S имеется общий преффикс iEtS у двух альтернатив, причём одна из них, целиком и полностью из него состоит. Следовательно, эта альтернатива без преффикса равна eps. Поэтому получим: S -> a | iEtSS’1, S’1 -> eS | eps. Так как больше нет общих преффиксов во всех правых частях, то алгоритм завершает работу.

Пример 4.3

Для грамматики с правилами:

S -> a|ab|abc|abcd,

получим:

S -> aS'1

S'1'1 -> cS'1'1'1|eps

S'1'1'1 -> d|eps

S'1 -> eps|bS'1'1

На первом шаге а является общим префиксом для всех альтернатив правила S, поэтому получим: S -> aS’1, S’1 -> b | bc | bcd | eps.

На втором шаге, у S’1 имеется общий преффикс b для трёх альтернатив, следовательно, получим: S’1 -> bS’1’1 | eps, S’1’1 -> c | cd | eps.

На третьем шаге у S’1’1 есть общий преффикс c у двух альтернатив. Получим: S’1’1 -> cS’1’1’1 | eps, S’1’1’1 -> d | eps. После третьего шага, у всех правых частей продукции нет общих преффиксов, поэтому алгоритм завершает работу.

Пример 4.4.

Для грамматики с правилами:

S -> aac|aab|bad|d|bac

получим:

S -> d|aaS'1|baS'2

S'1 -> b|c

S'2 -> c|d.

На первом шаге у S имеется общий преффикс aa у двух альтернатив (aac, aab). Получим правила: S -> aaS’1 | bac | bad | d, S’1 -> c | b.

На втором шаге у S имеется общий преффикс ba у двух других альтернатив (bac, bad). Получим правила: S -> aaS’1 | baS’2 | d,

S’2 -> c | d.

После двух шагов, все продукции в правых частях не содержат общих преффиксов, поэтому алгоритм завершает работу.

Результаты тестов, определённых в классе TestLeftFactor, дают всё выше определённое.

**package** ru.osipov.labs.lab2;  
  
**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.TestDFA;  
**import** ru.osipov.labs.lab2.grammars.Grammar;  
**import** ru.osipov.labs.lab2.jsonParser.SimpleJsonParser;  
**import** ru.osipov.labs.lab2.jsonParser.jsElements.JsonObject;  
  
@ExtendWith(SpringExtension.**class**)  
@RunWith(SpringJUnit4ClassRunner.**class**)  
@ContextConfiguration(classes= TestLeftFactor.**class**)  
**public class** TestLeftFactor {  
 @Test  
 **public void** testFactor1() {  
 String p = System.*getProperty*(**"user.dir"**);  
 System.***out***.println(p);  
 p = p+**"\\src\\test\\java\\ru\\osipov\\labs\\lab2\\"**;  
 p = p+**"grammarJson\\testFactor\_1.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);  
 **int** N1 = G.getNonTerminals().size();  
 G = G.deleteLeftFactor();  
 **assert** G != **null**;  
 System.***out***.println(**"Target"**);*//Step 1: A -> aAA'1, | aBc, A'1 -> B | c, Step 2: A -> aA'2, A'2 -> AA'1 | Bc.* System.***out***.println(G);  
 **assert** G.getN\_e().size() == 0 && G.getNonTerminals().size() == N1 + 2;  
 }  
  
 @Test  
 **public void** testFactor\_2(){  
 String p = System.*getProperty*(**"user.dir"**);  
 System.***out***.println(p);  
 p = p+**"\\src\\test\\java\\ru\\osipov\\labs\\lab2\\"**;  
 p = p+**"grammarJson\\testFactor\_2.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);  
 **int** N1 = G.getNonTerminals().size();  
 G = G.deleteLeftFactor();  
 **assert** G != **null**;  
 System.***out***.println(**"Target"**);  
 System.***out***.println(G);  
 **assert** G.getN\_e().size() != 0 && G.getNonTerminals().size() == N1 + 1;  
 }  
  
 @Test  
 **public void** testFactor\_3(){  
 String p = System.*getProperty*(**"user.dir"**);  
 System.***out***.println(p);  
 p = p+**"\\src\\test\\java\\ru\\osipov\\labs\\lab2\\"**;  
 p = p+**"grammarJson\\testFactor\_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);  
 **int** N1 = G.getNonTerminals().size();  
 G = G.deleteLeftFactor();  
 **assert** G != **null**;  
 System.***out***.println(**"Target"**);  
 System.***out***.println(G);  
 **assert** G.getN\_e().size() != 0 && G.getNonTerminals().size() == N1 + 3;  
 }  
  
 @Test  
 **public void** testFactor\_4(){  
 String p = System.*getProperty*(**"user.dir"**);  
 System.***out***.println(p);  
 p = p+**"\\src\\test\\java\\ru\\osipov\\labs\\lab2\\"**;  
 p = p+**"grammarJson\\testFactor\_4.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);  
 **int** N1 = G.getNonTerminals().size();  
 G = G.deleteLeftFactor();  
 **assert** G != **null**;  
 System.***out***.println(**"Target"**);  
 System.***out***.println(G);  
 **assert** G.getN\_e().size() == 0 && G.getNonTerminals().size() == N1 + 2;  
 }  
  
}

Листинг 31. Тестирование алгоритма левой факторизации.

**5. Анализ результатов и краткие выводы.**

Полученное в ходе разработки приложение собрано в виде единого арихвного файла в формате jar. Главная точка входа в приложение расположена в типе Main, определённый следующим образом:

**package** ru.osipov.labs.lab2;  
  
**import** org.springframework.boot.CommandLineRunner;  
**import** org.springframework.boot.SpringApplication;  
**import** org.springframework.boot.SpringBootConfiguration;  
**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.lab2.jsonParser.SimpleJsonParser;  
**import** ru.osipov.labs.lab2.jsonParser.jsElements.JsonObject;  
**import** ru.osipov.labs.lab2.grammars.json.InvalidJsonGrammarException;  
**import** java.util.\*;  
  
@SpringBootConfiguration  
**public class** Main **implements** CommandLineRunner {  
 **public static void** main(String[] args){  
 SpringApplication.*run*(Main.**class**,args);  
 }  
  
 @Override  
 **public void** run(String... args) **throws** Exception {  
 **if**(args.**length** < 2) {  
 System.***out***.println(**"Error. Specify filename (.json) to be processed"**);  
 System.*exit*(-1);  
 }  
 String fn = args[1];

SimpleJsonParser parser = **new** SimpleJsonParser();  
 JsonObject ob = parser.parse(fn);

**if** (ob != **null**) {  
 System.***out***.println(**"Json was read successfull."**);

Grammar G = **null**;  
 **try** {  
 G = **new** Grammar(ob);  
 }  
 **catch** (InvalidJsonGrammarException e){  
 System.***out***.println(e.getMessage());  
 System.*exit*(-2);  
 }

System.***out***.println(G);

G = Grammar.*deleteLeftRecursion*(G);  
 System.***out***.println(**"deleted recursion"**);  
 System.***out***.println(G);  
 G = G.deleteLeftFactor();  
 System.***out***.println(**"deleted preffix"**);  
 System.***out***.println(G);

}

**else**{  
 System.***out***.println(**"Invalid json. Json document is required!"**);  
 System.*exit*(-3);  
 }

}

}

Листинг 32. Главный класс приложения. (Точка входа).

Программа удовлетворяет требованиям задачи и правильно проводит устранение eps-правил из грамматики, левой рекурсии, а также корректно для любой последовательности альтернатив производит левую факторизацию, т.е. устранение общих преффиксов у двух или более альтернатив.

Оценим теперь сложность реализованных алгоритмов в худшем случае. Как уже было сказано выше, для устранения eps-правил из грамматики, алгоритм имеет экспоненциальную сложность для длинных правил, равную

А для коротких правил, он имеет сложность

правил грамматики. Что касается алгоритма устранения левой рекурсии, то оценим сначала сложность выполнения процедуры *replaceAlternativesWithFirstN.* Для множеств альтернатив правил Ai и Aj, а также нетерминала Aj, она возвращает новое множество альтернатив для Ai, в котором, каждый первый символ Aj, заменён на его множество продукции, т.е. В её теле на первом шаге происходит преобразование множества альтернатив Aj в список затем в данный список. Преобразование занимает максимальное число альтернатив, которое может содержать правило в грамматике. Далее процедура перебирает этот список, и там, где первым символов в альтернативе служит нетерминал Aj, то происходит его замена, путём формирования нового множества альтернатив, полученного из исходного, заменой Aj новыми альтернативами. Это замена обходится в максимальная длина строки грамматических символов, т.е. альтернативы.

В итоге, сложность данной процедуры будет равна Пусть далее число нетерминалов для грамматики *G*. Внутренний цикл в процедуре deleteLeftRecursion обходится в . Внешний цикл выполняется за Получение множества упорядоченных нетерминалов занимает . Аналогично занимает устранение цепных правил из грамматики, и удаление бесполезных символов. В итоге, сложность алгоритма устранения левой рекурсии будет равна Оценим теперь сложность алгоритма левой факторизации. Заметим, что вычисление множества альтернатив, не содержащих общий преффикс занимает , а вычисление общего преффикса у множества альтернив для одного правила также . Пусть k = максимально возможное число преффиксов для двух альтернатив. Так как на каждом шаге необходимо заново проверять все правила грамматики на наличие оставшихся преффиксов, то сложность цикла *while* составит . Что касается устранения лишних правил с помощью процедур *getGrammarWithoutEqualRules, getNonCycledGrammar, deleteUselessSymbols,* то все они имеют сложность . Поскольку нетерминалов может быть гораздо больше чем максимальное число альтернатив для каждого правила грамматики, а число , то сложность алгоритма

**Список литературы и использованных источников.**

1. ECMA-404, The JSON Data Interchange Syntax 2nd edition.

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

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

4. Eliminating left-recursion: three steps. URL: http://www.d.umn.edu/~hudson/5641/l11m.pdf