ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

Кафедра вычислительных систем

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовой работе по дисциплине

«Теория языков программирования и методы трансляции»

Выполнил:

студент гр. ИП-613

Чистов Д.А.

Проверила:

Доцент кафедры ПМиК

Бах О.А.

Новосибирск, 2019

Содержание

1. Текст задания……………………………………………………………………………3
2. Описание алгоритма решения задачи………………………………………………….3
3. Описание основных блоков программы……………………………………………….4
4. Листинг кода программы………………………………………………………………..5

# 1. Текст задания

Преобразования КС-грамматик и виды разбора. Вариант 16 - Написать программу, которая будет принимать на вход контекстно-свободную грамматику в каноническом виде (проверить корректность задания и при отрицательном результате выдать соответствующее сообщение) и приведёт её к нормальной форме Хомского. Программа должна проверить построенную грамматику (БНФ) на эквивалентность исходной: по обеим грамматикам сгенерировать множества всех цепочек в заданном пользователем диапазоне длин и проверить их на идентичность. Для подтверждения корректности выполняемых действий 4 предусмотреть возможность корректировки любого из построенных множеств пользователем (изменение цепочки, добавление, удаление…). При обнаружении несовпадения должна выдаваться диагностика различий – где именно несовпадения и в чём они состоят. Построить дерево вывода для любой выбранной цепочки из числа сгенерированных.

# 2. Описание алгоритма решения задачи

Задачу можно разделить на 3 основные части, каждую из которых необходимо выполнить:

1. Нужно научить программу определять, что КС-гармматика задана нам в каноническом виде, для этого используем правила определения является ли грамматика в каноническом виде и в случае когда, все верно, приводим её к нормальной форме Хомского с помощью алгоритма данного в пункте 3.3.1

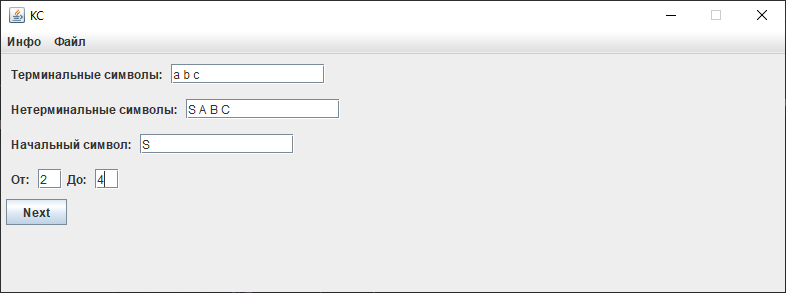


Рис. 1 Ввод начальных данных



Рис. 2 Ввод правил

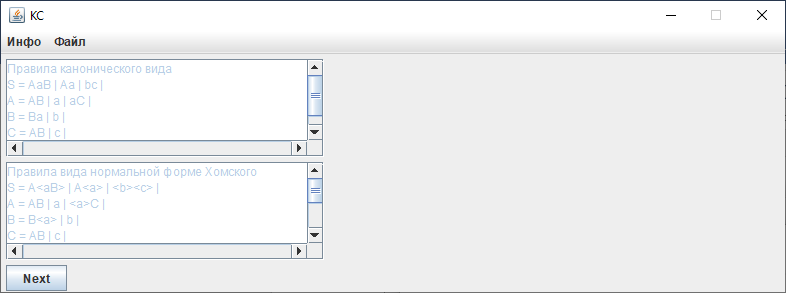


Рис. 3 Вывод правил в двух видах

1. Затем мы генерируем цепочки определенного диапазона проходя по всем вариантам их построения(это нужно для запоминания пути построения конечных цепочек , по которому мы в конце будем строить дерево).И проверяем каждый ряд конечных цепочек разного вида на эквивалентность друг к другу. В случае ошибки программа незамедлительно сообщит об этом.

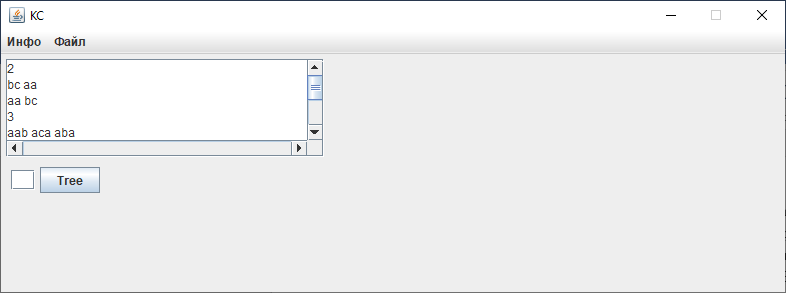


Рис 4. Демонстрация цепочек

1. Далее необходимо построить дерево вывода, для этого мы используем уже заранее построенный массив путей, в котором содержится все пути конечных цепочек.

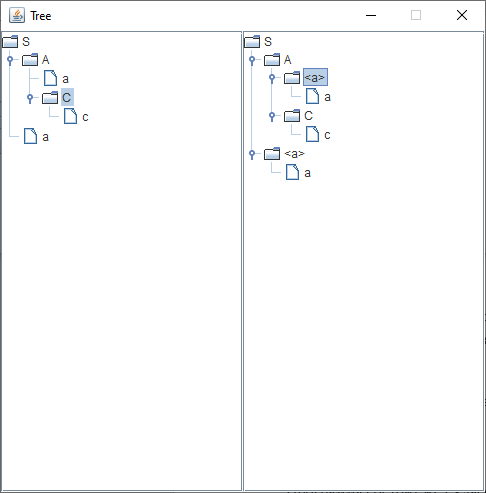


Рис.5 Дерево Вывода

Перечисленные выше пункты позволяют генерировать цепочки, вида указанном в задании.

# 3. Описание основных блоков программы

Программа состоит из 1 классов.

**public class** Main Класс в котором происходят основные метода и храняться данные

**public static void** rules() Метод позволяющий ввести правила для дальнейшей их проверки

**public static boolean** check\_Kanon() Метод проверяющий, на каноничность вида

**public static void** homski() Метод строящий правило вида нормальной формы Хомского, и позволяющее построить дерево в случае верной проверки на эквивалентность

**public static void** generation(HashMap <String, ArrayList<String>> map, ArrayList <String> list,HashMap <String, ArrayList<String>> ans, HashMap<Integer, HashMap<String, String>> tree) Метод генерирующий цепочки конечного вида и их пути.

# 4. Листинг кода программы

\*В листинге кода опущены строки форм для диалога с пользователем и методы для работы со строкой

**public class** Main {  
 **private static** String *jt\_str\_term*, *jt\_str\_noterm*, *jt\_str\_begin*, *jt\_str\_numb*, *jt\_str\_nume*;  
 **private static int** *num\_begin*, *num\_end*;  
 **private static int** *index* = 0;  
 **private static** JFrame *myWindow*;  
 **private static** JPanel *jp\_main*;  
 **private static** JPanel *jp\_rule*;  
 **private static** JPanel *jp\_gramm*;  
 **private static** JPanel *jp\_chain*;  
 **private static** ArrayList <String> *str\_temp\_term* = **new** ArrayList<>(); *//терминальные символы* **private static** ArrayList <String> *str\_temp\_noterm* = **new** ArrayList<>(); *// нетерминальные символы канон вида* **private static** HashMap <String, ArrayList<String>> *rule* = **new** HashMap<>(); *//Содержит перечень правил канон вида* **private static** HashMap <String, ArrayList<String>> *answer\_bas* = **new** HashMap<>(); *//Содержит перечень ответов простого вида* **private static** HashMap <String, ArrayList<String>> *answer\_Hom* = **new** HashMap<>(); *//Содержит перечень ответов вида Хомского* **private static** HashMap <String, ArrayList<String>> *rule\_Hom* = **new** HashMap<>(); *//Содержит перечень правил вида Хомского* **private static** ArrayList <String> *str\_temp\_noter\_Hom* = **new** ArrayList<>(); *// нетерминальные символы вида Хомского* **private static** HashMap<Integer, HashMap<String, String>> *tree\_number* = **new** HashMap<>(); *// Содержит номера правил* **private static** HashMap<Integer, HashMap<String, String>> *tree\_number\_Hom* = **new** HashMap<>(); *// Содержит номера правил для Хомского* **private static** JMenuItem *avtor*;  
  
 **public static void** main(String[] args) {  
 *myWindow* = **new** JFrame(**"KC"**);  
 *myWindow*.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);  
 JMenuBar menubar = **new** JMenuBar();  
 *// создаем меню* menubar.add(*info*(*myWindow*));  
 menubar.add(*file*(*myWindow*));  
 *// добавляем панель меню в окно  
 myWindow*.setJMenuBar(menubar);  
 *// панель для внесения данных  
 jp\_main* = **new** JPanel(**new** VerticalLayout());  
 *jp\_rule* = **new** JPanel(**new** VerticalLayout());  
 *jp\_gramm* = **new** JPanel(**new** VerticalLayout());  
 *jp\_chain* = **new** JPanel(**new** VerticalLayout());  
 *//Панель для терминальных* JPanel panel1 = **new** JPanel();  
 JLabel lb\_term = **new** JLabel(**"Терминальные символы: "**);  
 JTextField jt\_term = **new** JTextField(15);  
 panel1.add(lb\_term);  
 panel1.add(jt\_term);  
 *jp\_main*.add(panel1);  
 *//Панель для не терминальных* panel1 = **new** JPanel();  
 JLabel lb\_noterm = **new** JLabel(**"Нетерминальные символы: "**);  
 JTextField jt\_noterm = **new** JTextField(15);  
 panel1.add(lb\_noterm);  
 panel1.add(jt\_noterm);  
 *jp\_main*.add(panel1);  
 *//Начальный символ* panel1 = **new** JPanel();  
 JLabel jl\_begin = **new** JLabel(**"Начальный символ: "**);  
 JTextField jt\_begin = **new** JTextField(15);  
 panel1.add(jl\_begin);  
 panel1.add(jt\_begin);  
 *jp\_main*.add(panel1);  
 *//Диапазон* panel1 = **new** JPanel();  
 JLabel jl\_diap\_beg = **new** JLabel(**"От: "**);  
 JTextField jt\_diap\_beg = **new** JTextField(2);  
 panel1.add(jl\_diap\_beg);  
 panel1.add(jt\_diap\_beg);  
 JLabel jl\_diap\_end = **new** JLabel(**"До: "**);  
 JTextField jt\_diap\_end = **new** JTextField(2);  
 panel1.add(jl\_diap\_end);  
 panel1.add(jt\_diap\_end);  
 *jp\_main*.add(panel1);  
 *myWindow*.add(*jp\_main*);  
 *//Добавление кнопки для проверки данных* JButton check = **new** JButton(**"Next"**);  
 check.addActionListener(**new** ActionListener() {  
  
 @Override  
 **public void** actionPerformed(ActionEvent e) {  
  
 *jt\_str\_term* = jt\_term.getText();  
 *jt\_str\_noterm* = jt\_noterm.getText();  
 *jt\_str\_begin* = jt\_begin.getText();  
 *jt\_str\_numb* = jt\_diap\_beg.getText();  
 *jt\_str\_nume* = jt\_diap\_end.getText();  
 **for** (String std:*jt\_str\_term*.split(**" "**)) {  
 *str\_temp\_term*.add(std);  
 }  
 **for** (String std:*jt\_str\_noterm*.split(**" "**)) {  
 *str\_temp\_noterm*.add(std);  
 *str\_temp\_noter\_Hom*.add(std);  
 }  
 String regex = **"-?\\d+(\\.\\d+)?"**;  
 **if** (!(*jt\_str\_numb*.matches(regex)) || !(*jt\_str\_nume*.matches(regex))) {  
 *str\_temp\_noterm*.clear();  
 *str\_temp\_term*.clear();  
 *str\_temp\_noter\_Hom*.clear();  
 JOptionPane.*showMessageDialog*(*myWindow*, **"ОШИБКА, ДИАПАЗОН ЗАДАЕТСЯ ТОЛЬКО ЧИСЛАМИ"**);  
 **return**;  
 } **else** {  
 *num\_begin* = Integer.*parseInt*(*jt\_str\_numb*);  
 *num\_end* = Integer.*parseInt*(*jt\_str\_nume*);  
 **if** (*num\_begin* == 0 || *num\_end* == 0 || *num\_end* - *num\_begin* <= 0) {  
 *str\_temp\_noterm*.clear();  
 *str\_temp\_term*.clear();  
 *str\_temp\_noter\_Hom*.clear();  
 JOptionPane.*showMessageDialog*(*myWindow*, **"ОШИБКА, ДИАПАЗОН ЗАДАН НЕ ВЕРНО"**);  
 **return**;  
 }  
 }  
  
 **if** (*str\_temp\_noterm*.isEmpty()) {  
 *str\_temp\_noterm*.clear();  
 *str\_temp\_term*.clear();  
 *str\_temp\_noter\_Hom*.clear();  
 JOptionPane.*showMessageDialog*(*myWindow*, **"ОШИБКА, ПУСТОЕ ПОЛЕ"**);  
 **return**;  
 }  
 **if** (*str\_temp\_term*.isEmpty()) {  
 *str\_temp\_noterm*.clear();  
 *str\_temp\_term*.clear();  
 *str\_temp\_noter\_Hom*.clear();  
 JOptionPane.*showMessageDialog*(*myWindow*, **"ОШИБКА, ПУСТОЕ ПОЛЕ"**);  
 **return**;  
 }  
 *//Проверка на вшивость* **boolean** flag = **true**;  
 **for** (String std: *str\_temp\_noterm*) {  
 **if** (std.equals(*jt\_str\_begin*)) {  
 flag = **false**;  
 }  
 }  
 **if** (flag) {  
 *str\_temp\_noterm*.clear();  
 *str\_temp\_term*.clear();  
 *str\_temp\_noter\_Hom*.clear();  
 JOptionPane.*showMessageDialog*(*myWindow*, **"ОШИБКА, НАЧАЛЬНЫЙ СИМВОЛ НЕ ПРИНАДЛЕЖИТ НЕТЕРМИНАЛЬНЫМ"**);  
 **return**;  
 }  
  
 flag = **false**;  
 **for** (String std: *str\_temp\_noterm*) {  
 **for** (String std1 : *str\_temp\_term*) {  
 **if** (std.equals(std1)) {  
 flag = **true**;  
 }  
 }  
 }  
  
 **if** (flag) {  
 *str\_temp\_noterm*.clear();  
 *str\_temp\_term*.clear();  
 *str\_temp\_noter\_Hom*.clear();  
 JOptionPane.*showMessageDialog*(*myWindow*, **"ОШИБКА, ТЕРМИНАЛЬНЫЕ СИВМОЛЫ ПРИНАДЛЕЖАТ НЕТЕРМИНАЛЬНЫМ"**);  
 **return**;  
 }  
 *jp\_main*.setVisible(**false**);  
 *rules*();  
  
 }  
 });  
 *jp\_main*.add(check);  
 *myWindow*.setVisible(**true**);  
 *myWindow*.setSize(800, 300);  
 *myWindow*.setResizable(**false**);  
 }  
  
 **public static void** rules() {  
 JLabel pt = **new** JLabel();  
 pt.setText(*str\_temp\_noterm*.get(*index*));  
 JTextField pp = **new** JTextField(14);  
 JButton ne = **new** JButton(**"Next"**);  
 ne.addActionListener(**new** ActionListener() {  
  
 @Override  
 **public void** actionPerformed(ActionEvent e) {  
 **if** (*str\_temp\_noterm*.size() == *index* + 1) {  
 **if**(*check\_Rules*(pp.getText())) {  
 **if** (*check\_Kanon*()) {  
 *jp\_rule*.setVisible(**false**);  
 *homski*();  
 } **else** {  
 *rule*.clear();  
 *index* = 0;  
 *jp\_rule*.removeAll();  
 *jp\_rule*.updateUI();  
 *rules*();  
 }  
 } **else** {  
 *rule*.clear();  
 *index* = 0;  
 *jp\_rule*.removeAll();  
 *jp\_rule*.updateUI();  
 *rules*();  
 }  
 } **else** {  
 **if**(*check\_Rules*(pp.getText())) {  
 *index*++;  
 } **else** {  
 *rule*.clear();  
 *index* = 0;  
 }  
 *jp\_rule*.removeAll();  
 *jp\_rule*.updateUI();  
 *rules*();  
 }  
 }  
 });  
 *jp\_rule*.add(pt);  
 *jp\_rule*.add(pp);  
 *jp\_rule*.add(ne);  
 *myWindow*.add(*jp\_rule*);  
 }  
  
 **public static boolean** check\_Rules(String pp) {  
 **if** (pp.equals(**""**)) {  
 JOptionPane.*showMessageDialog*(*myWindow*, **"ВВИД ГРАММАТИКИ НЕ В КАНОНИЧЕСКОМ ВИДЕ"**);  
 **return false**;  
 }  
 ArrayList<String> templ = **new** ArrayList<>();  
 **for** (String std: pp.split(**" "**)) {  
 templ.add(std);  
 }  
  
 **for** (String std: templ) {  
 **for** (String s:*jt\_str\_term*.split(**" "**)) {  
 std = std.replaceAll(s, **""**);  
 }  
 **for** (String s:*jt\_str\_noterm*.split(**" "**)) {  
 std = std.replaceAll(s, **""**);  
 }  
 **if** (!(std.equals(**""**))) {  
 JOptionPane.*showMessageDialog*(*myWindow*, **"В ПРАВИЛЕ ЕСТЬ СИМВОЛЫ КОТОРЫХ НЕТУ В ГРАММАТИКЕ"**);  
 **return false**;  
 }  
 }  
 *rule*.put(*str\_temp\_noterm*.get(*index*), templ);  
 **return true**;  
 }  
  
 **public static boolean** check\_Kanon() {  
 *// цепные правила* **for** (**int** i = 0; i < *str\_temp\_noterm*.size(); i++) {  
 **for** (String str : *rule*.get(*str\_temp\_noterm*.get(i))) {  
 **for** (String st: *str\_temp\_noterm*) {  
 **if** (str.equals(st)) {  
 JOptionPane.*showMessageDialog*(*myWindow*, **"ВВИД ГРАММАТИКИ НЕ В КАНОНИЧЕСКОМ ВИДЕ 4 "** + *str\_temp\_noterm*.get(i) );  
 **return false**;  
 }  
 }  
 }  
 }  
 *// все недостижимые символы* String ttt = *jt\_str\_begin*;  
 **for** (**int** i = 0; i < *str\_temp\_noterm*.size(); i++) {  
 **for** (String str : *rule*.get(*str\_temp\_noterm*.get(i))) {  
 **for** (String st: *str\_temp\_noterm*) {  
 **if** (str.contains(st) && !ttt.contains(st)) {  
 ttt += st;  
 }  
 }  
 }  
 }  
 **if** (!(ttt.length() == *str\_temp\_noterm*.size())) {  
 JOptionPane.*showMessageDialog*(*myWindow*, **"ВВИД ГРАММАТИКИ НЕ В КАНОНИЧЕСКОМ ВИДЕ 2"**);  
 **return false**;  
 }  
  
 *//все бесплодные символы* String strp = **""**;  
 **int** counter = 0;  
 **int** intz = 0, intf;  
 **do** {  
 intf = intz;  
 **for** (**int** i = 0; i < *str\_temp\_noterm*.size(); i++) {  
 **for** (String str : *rule*.get(*str\_temp\_noterm*.get(i))) {  
 **for** (String st : *str\_temp\_noterm*) {  
 **if** (!str.contains(st) || strp.contains(st)) {  
 counter++;  
 }  
 }  
 **if** (counter == *str\_temp\_noterm*.size()) {  
 **if** (!strp.contains(*str\_temp\_noterm*.get(i))) {  
 strp += *str\_temp\_noterm*.get(i);  
 intz++;  
 }  
 }  
 counter = 0;  
 }  
 }  
 } **while** (intz - intf > 0);  
 **if** (intz != *str\_temp\_noterm*.size()) {  
 JOptionPane.*showMessageDialog*(*myWindow*, **"ВВИД ГРАММАТИКИ НЕ В КАНОНИЧЕСКОМ ВИДЕ 1"**);  
 **return false**;  
 }  
  
 **return true**;  
 }  
  
 **public static void** homski() {  
 **int**[] term;  
 **int**[] noterm;  
 **for** (**int** i = 0; i < *str\_temp\_noterm*.size(); i++) {  
 ArrayList<String> templ = **new** ArrayList<>();  
 ArrayList<String> pp = **new** ArrayList<>();  
 **for** (String str : *rule*.get(*str\_temp\_noterm*.get(i))) {  
 noterm = **new int**[]{0, 0};  
 term = **new int**[]{0, 0};  
 *// 5 правило* **boolean** flag = **true**;  
 **if** (str.length() > 2) {  
 String s = str.substring(0, 1);  
 **for** (String nstr: *str\_temp\_term*) {  
 **if** (str.substring(0, 1).equals(nstr)) {  
 s = **"<"** + str.substring(0, 1) + **">"**;  
 **for** (String sss: *str\_temp\_noter\_Hom*) {  
 **if** (sss.equals(s)) {  
 flag = **false**;  
 }  
 }  
 **if** (flag) {  
 *str\_temp\_noter\_Hom*.add(s);  
 flag = **true**;  
 }  
 pp.add(str.substring(0, 1));  
 *rule\_Hom*.put(s, pp);  
 pp = **new** ArrayList<>();  
 }  
 }  
 templ.add(s + **"<"** + str.substring(1) + **">"**);  
 **int** p = str.length() - 2;  
 str = str.substring(1);  
 **for** (**int** j = 0; j < p; j++) {  
 s = str.substring(0, 1);  
 **for** (String nstr: *str\_temp\_term*) {  
 **if** (str.substring(0, 1).equals(nstr)) {  
 s = **"<"** + str.substring(0, 1) + **">"**;  
 **for** (String sss: *str\_temp\_noter\_Hom*) {  
 **if** (sss.equals(s)) {  
 flag = **false**;  
 }  
 }  
 **if** (flag) {  
 *str\_temp\_noter\_Hom*.add(s);  
 flag = **true**;  
 }  
 pp.add(str.substring(0, 1));  
 *rule\_Hom*.put(s, pp);  
 pp = **new** ArrayList<>();  
 }  
 }  
 pp.add(s + str.substring(1));  
 **for** (String sss: *str\_temp\_noter\_Hom*) {  
 **if** (sss.equals(**"<"** + str + **">"**)) {  
 flag = **false**;  
 }  
 }  
 **if** (flag) {  
 *str\_temp\_noter\_Hom*.add(**"<"** + str + **">"**);  
 flag = **true**;  
 }  
 *rule\_Hom*.put(**"<"** + str + **">"**, pp);  
  
 pp = **new** ArrayList<>();  
 str = str.substring(1);  
 }  
  
 } **else** {  
 *// 1 правило* **for** (String nstr: *str\_temp\_term*) {  
 *countChar*(str, nstr.charAt(0), term);  
 }  
 **for** (String nstr: *str\_temp\_noterm*) {  
 *countChar*(str, nstr.charAt(0), noterm);  
 }  
 **if** ((noterm[0] == 1 && noterm[1] == 1 && term[0] == 0 && term[1] == 0) ||  
 (noterm[0] == 0 && noterm[1] == 0 && term[0] == 1 && term[1] == 0)) {  
 templ.add(str);  
  
 }  
 *// 2 правило* flag = **true**;  
 **if** (noterm[0] == 0 && noterm[1] == 1 && term[0] == 1 && term[1] == 0) {  
 String s = **"<"** + str.substring(0, 1) + **">"**;  
 **for** (String sss: *str\_temp\_noter\_Hom*) {  
 **if** (sss.equals(s)) {  
 flag = **false**;  
 }  
 }  
 **if** (flag) {  
 *str\_temp\_noter\_Hom*.add(s);  
 }  
 templ.add(s + str.substring(1));  
 pp.add(str.substring(0, 1));  
 *rule\_Hom*.put(s, pp);  
 pp = **new** ArrayList<>();  
 }  
 *// 3 правило* flag = **true**;  
 **if** (noterm[0] == 1 && noterm[1] == 0 && term[0] == 0 && term[1] == 1) {  
 String s = **"<"** + str.substring(1) + **">"**;  
 **for** (String sss: *str\_temp\_noter\_Hom*) {  
 **if** (sss.equals(s)) {  
 flag = **false**;  
 }  
 }  
 **if** (flag) {  
 *str\_temp\_noter\_Hom*.add(s);  
 }  
 templ.add(str.substring(0, 1) + s);  
 pp.add(str.substring(1));  
 *rule\_Hom*.put(s, pp);  
 pp = **new** ArrayList<>();  
 }  
 *//4 правило* flag = **true**;  
 **if** (noterm[0] == 0 && noterm[1] == 0 && term[0] == 1 && term[1] == 1) {  
 String s = **"<"** + str.substring(0, 1) + **">"**;  
 **for** (String sss: *str\_temp\_noter\_Hom*) {  
 **if** (sss.equals(s)) {  
 flag = **false**;  
 }  
 }  
 **if** (flag) {  
 *str\_temp\_noter\_Hom*.add(s);  
 }  
 flag = **true**;  
 String t = **"<"** + str.substring(1) + **">"**;  
 **for** (String sss: *str\_temp\_noter\_Hom*) {  
 **if** (sss.equals(t)) {  
 flag = **false**;  
 }  
 }  
 **if** (flag) {  
 *str\_temp\_noter\_Hom*.add(t);  
 }  
 templ.add(s + t);  
 pp.add(str.substring(0, 1));  
 *rule\_Hom*.put(s, pp);  
 pp = **new** ArrayList<>();  
 pp.add(str.substring(1));  
 *rule\_Hom*.put(t, pp);  
 pp = **new** ArrayList<>();  
 }  
 }  
  
  
 }  
 *rule\_Hom*.put(*str\_temp\_noterm*.get(i), templ);  
 }  
 String s = **new** String();  
 JTextArea textArea2 = **new** JTextArea(5, 30);  
 s += **"Правила канонического вида \n"**;  
 **for** (**int** i = 0; i < *rule*.size(); i++) {  
 s += *str\_temp\_noterm*.get(i) + **" = "**;  
 **for** (String std: *rule*.get(*str\_temp\_noterm*.get(i))) {  
 s += std + **" | "**;  
 }  
 s += **"\n"**;  
 }  
 textArea2.append(s);  
 textArea2.setEnabled(**false**);  
 JScrollPane scroll = **new** JScrollPane(textArea2,  
 JScrollPane.***VERTICAL\_SCROLLBAR\_ALWAYS***,  
 JScrollPane.***HORIZONTAL\_SCROLLBAR\_ALWAYS***);  
 *jp\_gramm*.add(scroll);  
 s = **new** String();  
 JTextArea textArea1 = **new** JTextArea(5, 30);  
 s += **"Правила вида нормальной форме Хомского \n"**;  
 **for** (**int** i = 0; i < *rule\_Hom*.size(); i++) {  
 s += *str\_temp\_noter\_Hom*.get(i) + **" = "**;  
 **for** (String std: *rule\_Hom*.get(*str\_temp\_noter\_Hom*.get(i))) {  
 s += std + **" | "**;  
 }  
 s += **"\n"**;  
 }  
 textArea1.append(s);  
 textArea1.setEnabled(**false**);  
 JScrollPane scroll1 = **new** JScrollPane(textArea1,  
 JScrollPane.***VERTICAL\_SCROLLBAR\_ALWAYS***,  
 JScrollPane.***HORIZONTAL\_SCROLLBAR\_ALWAYS***);  
 *jp\_gramm*.add(scroll1);  
 JButton bat = **new** JButton(**"Next"**);  
 bat.addActionListener(**new** ActionListener() {  
  
 @Override  
 **public void** actionPerformed(ActionEvent e) {  
 *jp\_gramm*.setVisible(**false**);  
 *generation*(*rule*, *str\_temp\_noterm*, *answer\_bas*, *tree\_number*);  
 *generation*(*rule\_Hom*, *str\_temp\_noter\_Hom*, *answer\_Hom*, *tree\_number\_Hom*);  
 JTextArea textArea = **new** JTextArea(5, 30);  
 **int** ind = 0;  
 **for** (**int** i = *num\_begin*; i <= *num\_end*; i++) {  
 textArea.append(i + **"\n"**);  
 **for** (String str: *answer\_bas*.get(Integer.*toString*(i))) {  
 **for** (String str1: *answer\_Hom*.get(Integer.*toString*(i))) {  
 **if** (str.equals(str1)) {  
 ind++;  
 }  
 }  
 }  
 **if** (ind == *answer\_bas*.get(Integer.*toString*(i)).size() && ind == *answer\_bas*.get(Integer.*toString*(i)).size()) {  
 **for** (String str: *answer\_bas*.get(Integer.*toString*(i))) {  
 textArea.append(str + **" "**);  
 }  
 textArea.append(**"\n"**);  
 **for** (String str: *answer\_Hom*.get(Integer.*toString*(i))) {  
 textArea.append(str + **" "**);  
 }  
 } **else** {  
 JOptionPane.*showMessageDialog*(*myWindow*, **"НЕСОВПАДЕНИЕ ЦЕПОЧЕК "** + ind);  
 **return**;  
 }  
 ind = 0;  
 textArea.append(**" \n"**);  
 }  
 JButton batton = **new** JButton(**"Tree"**);  
 JPanel panel1 = **new** JPanel();  
 JTextField text\_tree = **new** JTextField(2);  
 batton.addActionListener(**new** ActionListener() {  
  
 @Override  
 **public void** actionPerformed(ActionEvent e) {  
  
 *jp\_chain*.setEnabled(**false**);  
 JFrame myWindow = **new** JFrame(**"Tree"**);  
 myWindow.addWindowListener(**new** WindowAdapter() {  
 **public void** windowClosing(WindowEvent e) {  
 *jp\_chain*.setEnabled(**true**);  
 }  
 });  
 myWindow.setVisible(**true**);  
 myWindow.setSize(500, 500);  
 myWindow.setResizable(**false**);  
 String s = text\_tree.getText();  
 **int** lenght = s.length();  
 **int** counter = 2;  
 String str = **null**;  
 **try** {  
 str = *tree\_number*.get(lenght).get(s);  
 } **catch** (Exception ef) {  
 JOptionPane.*showMessageDialog*(myWindow, **"ЭТО МНОЖЕСТВО НЕ ПРЕНАДЛЕЖИТ К КОНЕЧНЫМ ЦЕПОЧКАМ"**);  
 **return**;  
 }  
 String ROOT = **null**;  
 **for** ( String key : *tree\_number*.get(lenght).keySet() ) {  
 **if** (*tree\_number*.get(lenght).get(key).equals(str.substring(0, 1))) {  
 ROOT = key;  
 }  
 }  
 String num\_rul = str.substring(0, 1);  
 String time\_root = **null**;  
 **boolean** flag = **false**;  
 **boolean** flag\_term = **true**;  
 DefaultMutableTreeNode root = **new** DefaultMutableTreeNode(ROOT);  
 **while**(counter <= str.length()) {  
 Enumeration en = root.depthFirstEnumeration();  
 **while** (en.hasMoreElements()) {  
 DefaultMutableTreeNode node = (DefaultMutableTreeNode) en.nextElement();  
 **if** (node.isLeaf()) {  
 flag = **false**;  
 String s\_non\_term = **null**;  
 **for** (String str\_1: *str\_temp\_noterm*) {  
 **if** (str\_1.equals(node.toString())) {  
 flag = **true**;  
 s\_non\_term = str\_1;  
 }  
 }  
 **if** (flag) {  
 String temp\_one = **null**;  
 String temo\_two = **null**;  
 **for** ( String key : *tree\_number*.get(lenght).keySet() ) {  
 **if** (*tree\_number*.get(lenght).get(key).equals(num\_rul)) {  
 temp\_one = key;  
 }  
 }  
 **for** ( String key : *tree\_number*.get(lenght).keySet() ) {  
 **if** (*tree\_number*.get(lenght).get(key).equals(str.substring(0, counter))) {  
 temo\_two = key;  
 }  
 }  
 flag\_term = **true**;  
 **if** (temp\_one.length() == temo\_two.length()) {  
 **for** (String sss: *str\_temp\_term*) {  
 **if** (temo\_two.substring(temp\_one.indexOf(s\_non\_term) , temp\_one.indexOf(s\_non\_term) + 1).equals(sss)) {  
 node.add(**new** DefaultMutableTreeNode(temo\_two.substring(temp\_one.indexOf(s\_non\_term) , temp\_one.indexOf(s\_non\_term) + 1), **false**));  
 flag\_term = **false**;  
 }  
 }  
 **if** (flag\_term) {  
 node.add(**new** DefaultMutableTreeNode(temo\_two.substring(temp\_one.indexOf(s\_non\_term), temp\_one.indexOf(s\_non\_term) + 1)));  
 }  
 } **else if** (temp\_one.indexOf(s\_non\_term) == 0){  
 **for**(**int** i = 0;i < temo\_two.length() - temp\_one.length() + 1; i++) {  
 flag\_term = **true**;  
 **for** (String sss: *str\_temp\_term*) {  
 **if** (temo\_two.substring(i , i + 1).equals(sss)) {  
 node.add(**new** DefaultMutableTreeNode(temo\_two.substring(i, i + 1), **false**));  
 flag\_term = **false**;  
 }  
 }  
 **if** (flag\_term) {  
 node.add(**new** DefaultMutableTreeNode(temo\_two.substring(i, i + 1), **true**));  
 }  
 }  
 } **else if** (temp\_one.indexOf(s\_non\_term) == temp\_one.length() - 1) {  
 flag\_term = **true**;  
 **for**(**int** i = temp\_one.length() - 1;i < temo\_two.length() ; i++) {  
 **for** (String sss: *str\_temp\_term*) {  
 **if** (temo\_two.substring(i , i + 1).equals(sss)) {  
 node.add(**new** DefaultMutableTreeNode(temo\_two.substring(i, i + 1), **false**));  
 flag\_term = **false**;  
 }  
 }  
 **if** (flag\_term) {  
 node.add(**new** DefaultMutableTreeNode(temo\_two.substring(i, i + 1), **true**));  
 }  
 }  
 } **else** {  
 **for**(**int** i = temp\_one.indexOf(s\_non\_term); i < temo\_two.length() - temp\_one.length() + 2 ; i++) {  
 flag\_term = **true**;  
 **for** (String sss: *str\_temp\_term*) {  
 **if** (temo\_two.substring(i , i + 1).equals(sss)) {  
 node.add(**new** DefaultMutableTreeNode(temo\_two.substring(i , i + 1), **false**));  
 flag\_term = **false**;  
 }  
 }  
 **if** (flag\_term) {  
 node.add(**new** DefaultMutableTreeNode(temo\_two.substring(i, i + 1), **true**));  
 }  
 }  
 }  
 **break**;  
 }  
 }  
 }  
 num\_rul = str.substring(0, counter);  
 counter++;  
 }  
  
  
 DefaultTreeModel treeModel1 = **new** DefaultTreeModel(root, **true**);  
 JTree tree1 = **new** JTree(treeModel1);  
 JPanel contents = **new** JPanel(**new** GridLayout(1, 2));  
  
  
 **try** {  
 str = *tree\_number\_Hom*.get(lenght).get(s);  
 } **catch** (Exception ef) {  
 JOptionPane.*showMessageDialog*(myWindow, **"ЭТО МНОЖЕСТВО НЕ ПРЕНАДЛЕЖИТ К КОНЕЧНЫМ ЦЕПОЧКАМ"**);  
 **return**;  
 }  
 String ROOT1 = **null**;  
 **for** ( String key : *tree\_number\_Hom*.get(lenght).keySet() ) {  
 **if** (*tree\_number\_Hom*.get(lenght).get(key).equals(str.substring(0, 1))) {  
 ROOT1 = key;  
 }  
 }  
 num\_rul = str.substring(0, 1);  
 counter = 2;  
 DefaultMutableTreeNode root1 = **new** DefaultMutableTreeNode(ROOT1);  
 **boolean** flag\_end = **false**;  
 **boolean** flag\_meddle = **false**;  
 **while**(counter <= str.length()) {  
 Enumeration en = root1.depthFirstEnumeration();  
 **while** (en.hasMoreElements()) {  
 DefaultMutableTreeNode node = (DefaultMutableTreeNode) en.nextElement();  
 **if** (node.isLeaf()) {  
 flag = **false**;  
 String s\_non\_term = **null**;  
 **for** (String str\_1: *str\_temp\_noter\_Hom*) {  
 **if** (str\_1.equals(node.toString())) {  
 flag = **true**;  
 s\_non\_term = str\_1;  
 }  
 }  
 **if** (flag) {  
 String temp\_one = **null**;  
 String temo\_two = **null**;  
 **for** ( String key : *tree\_number\_Hom*.get(lenght).keySet() ) {  
 **if** (*tree\_number\_Hom*.get(lenght).get(key).equals(num\_rul)) {  
 temp\_one = key;  
 }  
 }  
 **for** ( String key : *tree\_number\_Hom*.get(lenght).keySet() ) {  
 **if** (*tree\_number\_Hom*.get(lenght).get(key).equals(str.substring(0, counter))) {  
 temo\_two = key;  
 }  
 }  
 flag\_term = **true**;  
 **if** (temp\_one.substring(temp\_one.length() - 1).equals(**">"**)) {  
 System.***out***.println(temp\_one);  
 **for**(**int** i = temp\_one.length() - 1; i > 0; i-- ) {  
 **if** (temp\_one.substring(i, i + 1).equals(**"<"**)) {  
 **if** (temp\_one.indexOf(s\_non\_term) == i) {  
 flag\_end = **true**;  
 **break**;  
 }  
 }  
 }  
 }  
  
 **if** (*countCharall*(temp\_one, *str\_temp\_noter\_Hom*) == *countCharall*(temo\_two, *str\_temp\_noter\_Hom*)) {  
 System.***out***.println(temp\_one.indexOf(s\_non\_term) + **" "** + temp\_one + **" "** + s\_non\_term);  
 **for** (String sss : *str\_temp\_term*) {  
 **if** (temo\_two.substring(temp\_one.indexOf(s\_non\_term), temp\_one.indexOf(s\_non\_term) + 1).equals(sss)) {  
 node.add(**new** DefaultMutableTreeNode(temo\_two.substring(temp\_one.indexOf(s\_non\_term), temp\_one.indexOf(s\_non\_term) + 1), **false**));  
 flag\_term = **false**;  
 }  
 }  
 **if** (flag\_term) {  
 **if** (temo\_two.substring(temp\_one.indexOf(s\_non\_term), temp\_one.indexOf(s\_non\_term) + 1).equals(**"<"**)) {  
 String p = **""**;  
 **int** i = temp\_one.indexOf(s\_non\_term);  
 **int** j = temp\_one.indexOf(s\_non\_term) + 1;  
 **while** (!temo\_two.substring(i , j).equals(**">"**)) {  
 i++;  
 j++;  
 }  
 p += temo\_two.substring(temp\_one.indexOf(s\_non\_term) , j);  
 node.add(**new** DefaultMutableTreeNode(p, **true**));  
 } **else** {  
 node.add(**new** DefaultMutableTreeNode(temo\_two.substring(temp\_one.indexOf(s\_non\_term), temp\_one.indexOf(s\_non\_term) + 1)));  
 }  
 }  
  
 } **else if** (temp\_one.indexOf(s\_non\_term) == 0 || temp\_one.indexOf(**"<"**) == 0){  
 **for**(**int** i = 0;i < temo\_two.length() - temp\_one.length() + 1; i++) {  
 **if** (temo\_two.substring(i, i + 1).equals(**"<"**)) {  
 String p = **""**;  
 **while** (!temo\_two.substring(i, i + 1).equals(**">"**)) {  
 p += temo\_two.substring(i, i + 1);  
 i++;  
 }  
 p += temo\_two.substring(i, i + 1);  
 node.add(**new** DefaultMutableTreeNode(p, **true**));  
 } **else** {  
 flag\_term = **true**;  
 **for** (String sss : *str\_temp\_term*) {  
 **if** (temo\_two.substring(i, i + 1).equals(sss)) {  
 node.add(**new** DefaultMutableTreeNode(temo\_two.substring(i, i + 1), **false**));  
 flag\_term = **false**;  
 }  
 }  
 **if** (flag\_term) {  
 **if** (temo\_two.substring(i, i + 1) == **"<"**) {  
 String p = **null**;  
 **while** (temo\_two.substring(i, i + 1) != **">"**) {  
 p += temo\_two.substring(i, i + 1);  
 i++;  
 }  
 node.add(**new** DefaultMutableTreeNode(p, **true**));  
 } **else** {  
 node.add(**new** DefaultMutableTreeNode(temo\_two.substring(i, i + 1), **true**));  
 }  
 }  
 }  
 }  
 } **else  
 if** (temp\_one.indexOf(s\_non\_term) == temp\_one.length() - 1 || flag\_end) {  
 **if** (flag\_end) {  
 **for** (**int** i = temp\_one.indexOf(s\_non\_term);i < temo\_two.length();i++) {  
 **if** (temo\_two.substring(i, i + 1).equals(**"<"**)) {  
 String p = **""**;  
 **while** (!temo\_two.substring(i, i + 1).equals(**">"**)) {  
 p += temo\_two.substring(i, i + 1);  
 i++;  
 }  
 p += temo\_two.substring(i, i + 1);  
 node.add(**new** DefaultMutableTreeNode(p, **true**));  
 } **else** {  
 flag\_term = **true**;  
 **for** (String sss : *str\_temp\_term*) {  
 **if** (temo\_two.substring(i, i + 1).equals(sss)) {  
 node.add(**new** DefaultMutableTreeNode(temo\_two.substring(i, i + 1), **false**));  
 flag\_term = **false**;  
 }  
 }  
 **if** (flag\_term) {  
 **if** (temo\_two.substring(i, i + 1) == **"<"**) {  
 String p = **null**;  
 **while** (temo\_two.substring(i, i + 1) != **">"**) {  
 p += temo\_two.substring(i, i + 1);  
 i++;  
 }  
 node.add(**new** DefaultMutableTreeNode(p, **true**));  
 } **else** {  
 node.add(**new** DefaultMutableTreeNode(temo\_two.substring(i, i + 1), **true**));  
 }  
 }  
 }  
 }  
 } **else** {  
 flag\_term = **true**;  
 **for** (**int** i = temp\_one.length() - 1; i < temo\_two.length(); i++) {  
 **if** (temo\_two.substring(i, i + 1).equals(**"<"**)) {  
 String p = **""**;  
 **while** (!temo\_two.substring(i, i + 1).equals(**">"**)) {  
 p += temo\_two.substring(i, i + 1);  
 i++;  
 }  
 p += temo\_two.substring(i, i + 1);  
 node.add(**new** DefaultMutableTreeNode(p, **true**));  
 } **else** {  
 flag\_term = **true**;  
 **for** (String sss : *str\_temp\_term*) {  
 **if** (temo\_two.substring(i, i + 1).equals(sss)) {  
 node.add(**new** DefaultMutableTreeNode(temo\_two.substring(i, i + 1), **false**));  
 flag\_term = **false**;  
 }  
 }  
 **if** (flag\_term) {  
 **if** (temo\_two.substring(i, i + 1) == **"<"**) {  
 String p = **null**;  
 **while** (temo\_two.substring(i, i + 1) != **">"**) {  
 p += temo\_two.substring(i, i + 1);  
 i++;  
 }  
 node.add(**new** DefaultMutableTreeNode(p, **true**));  
 } **else** {  
 node.add(**new** DefaultMutableTreeNode(temo\_two.substring(i, i + 1), **true**));  
 }  
 }  
 }  
 }  
 }  
 }  
 **else** {  
 **for**(**int** i = temp\_one.indexOf(s\_non\_term); i < temo\_two.length() - temp\_one.length() + 2 ; i++) {  
  
 flag\_term = **true**;  
 **for** (String sss: *str\_temp\_term*) {  
 **if** (temo\_two.substring(i , i + 1).equals(sss)) {  
 node.add(**new** DefaultMutableTreeNode(temo\_two.substring(i , i + 1), **false**));  
 flag\_term = **false**;  
 }  
 }  
 **if** (flag\_term) {  
 node.add(**new** DefaultMutableTreeNode(temo\_two.substring(i, i + 1), **true**));  
 }  
 }  
 }  
 **break**;  
 }  
 }  
 }  
 num\_rul = str.substring(0, counter);  
 counter++;  
 }  
  
  
 DefaultTreeModel treeModel2 = **new** DefaultTreeModel(root1, **true**);  
 JTree tree2 = **new** JTree(treeModel2);  
 contents.add(**new** JScrollPane(tree1));  
 contents.add(**new** JScrollPane(tree2));  
 myWindow.setContentPane(contents);  
  
  
 }  
 });  
 panel1.add(text\_tree);  
 panel1.add(batton);  
 JScrollPane scroll1 = **new** JScrollPane(textArea,  
 JScrollPane.***VERTICAL\_SCROLLBAR\_ALWAYS***,  
 JScrollPane.***HORIZONTAL\_SCROLLBAR\_ALWAYS***);  
 *jp\_chain*.add(scroll1);  
 *jp\_chain*.add(panel1);  
 *avtor*.setEnabled(**true**);  
 *myWindow*.add(*jp\_chain*);  
 }  
 });  
 *jp\_gramm*.add(bat);  
 *myWindow*.add(*jp\_gramm*);  
 }  
  
 **public static void** generation(HashMap <String, ArrayList<String>> map, ArrayList <String> list,  
 HashMap <String, ArrayList<String>> ans, HashMap<Integer, HashMap<String, String>> tree) {  
 **boolean** flag\_all = **true**;  
 **boolean** flag\_term = **true**;  
 **boolean** flag\_add = **true**;  
 **int** counter = 0;  
 **int** add\_tree = 1;  
 ArrayList<String> m1 = **new** ArrayList<>();  
 ArrayList<String> m2 = **new** ArrayList<>();  
 ArrayList<String> answer = **new** ArrayList<>();  
 String split1 = **""**, split2 = **""**;  
 String s = **null**;  
 String ss = **null**;  
 HashMap <String, String> tree\_tamp = **new** HashMap<>();  
 m1.add(*jt\_str\_begin*);  
 **for** (**int** f = *num\_begin*; f <= *num\_end*; f++) {  
 answer = **new** ArrayList<>();  
 flag\_all = **true**;  
 flag\_term = **true**;  
 flag\_add = **true**;  
 m1.clear();  
 m2.clear();  
 split1 = **""**;  
 split2 = **""**;  
 s = **null**;  
 ss = **null**;  
 m1.add(*jt\_str\_begin*);  
 counter = 0;  
 tree\_tamp = **new** HashMap<>();  
 tree\_tamp.put(**"S"**, **"1"**);  
 **while** (flag\_all) {  
 **for** (String str1 : m1) {  
 split1 = **""**;  
 split2 = **""**;  
 counter = 0;  
 flag\_term = **true**;  
 add\_tree = 1;  
 **while** (flag\_term) {  
 s = str1.substring(counter, counter + 1);  
 **if** (s.equals(**"<"**)) {  
 *//Хомский* **int** counter\_temp = counter;  
 counter++;  
 **while** (!(str1.substring(counter, counter + 1).equals(**">"**))) {  
 s += str1.substring(counter, counter + 1);  
 counter++;  
 }  
 s += str1.substring(counter, counter + 1);  
 **if** (counter\_temp == 0) {  
 split2 = str1.substring(counter + 1);  
 } **else** {  
 split2 = str1.substring(counter + 1);  
 split1 = str1.substring(0, counter\_temp);  
 }  
 flag\_term = **false**;  
 counter = 0;  
 } **else** {  
 **for** (String str2 : list) {  
 **if** (str2.equals(s)) {  
 flag\_term = **false**;  
 **if** (counter == 0) {  
 split2 = str1.substring(counter + 1);  
 } **else** {  
 split2 = str1.substring(counter + 1);  
 split1 = str1.substring(0, counter);  
 }  
 counter = 0;  
 }  
 }  
 }  
 counter++;  
 }  
  
 **for** (String str : map.get(s)) {  
 flag\_add = **true**;  
 ss = split1 + str + split2;  
 **if** (*countCharsimple*(ss) > f) {  
 **continue**;  
 }  
 **if** (*countCharall*(ss, list) > f) {  
 **continue**;  
 }  
  
 **for** (String str3 : list) {  
 **if** (ss.contains(str3)) {  
 flag\_add = **false**;  
 m2.add(ss);  
 tree\_tamp.put(ss, tree\_tamp.get(str1) + add\_tree);  
 add\_tree++;  
 **break**;  
 }  
 }  
  
 **boolean** flag = **true**;  
 **if** (flag\_add && ss.length() == f) {  
 tree\_tamp.put(ss, tree\_tamp.get(str1) + add\_tree);  
 add\_tree++;  
 **for** (String sss : answer) {  
 **if** (sss.equals(ss)) {  
 flag = **false**;  
 }  
 }  
 **if** (flag) {  
 answer.add(ss);  
 }  
 }  
 }  
 }  
  
 m1.clear();  
 **for** (String str : m2) {  
 m1.add(str);  
 }  
 m2.clear();  
 **if** (m1.isEmpty()) {  
 flag\_all = **false**;  
 }  
 }  
 ans.put(Integer.*toString*(f), answer);  
 tree.put(f, tree\_tamp);  
 }  
 }