CS 410 – Automata Theory and Formal Languages

Fall 2022

Project II Report

Murat Can Altun

1. Introduction

For the conversion from CFGs to Chomsky Normal Form, we must follow the following steps:

1. Add a new start variable.
2. Remove all epsilon rules.
3. Remove all unit rules.
4. Convert the remaining rules to the proper form.

To replicate this solution digitally, mapping the rules to non-terminals in Java makes sense since the data structure operations in Java is easier than C++ to deal with. In the next sections, the data structures and algorithms used will be explained in detail.

2. Data Structures Used

ArrayLists: Used for storing and using the non-terminals and the terminals. The specific definitions for the ArrayLists are the following:

- ArrayList<String> nonTerminal
- ArrayList<String> terminal

HashMap: Used for storing and using the rules, uppercase replacements, and lowercase replacements. The keys in the rules HashMap are non-terminals, while the values are mapped to them as a String in the format <KEY, VALUE> for the rule KEY → VALUE. The replacements HashMaps maps the replacements for the last step, where every rule must be of either the form A → BC or the form A → a. So, the uppercase replacements HashMap would hold X → BC for the rule A → BCC, converting it to A → XC. For the lowercase replacements HashMap, it would hold a → Y for the rule A → aC, converting it to A → YC. And the HashMaps are defined like this:

- HashMap<String, String> rules
- HashMap<Character, String> lowercaseReplacements
- HashMap<String, String> uppercaseReplacements

3. The Algorithm

a) File Reading

In the beginning, the program has only one Scanner Object for scanning the file the user wants scanned. The program will start by reading G1.txt, and if there is no “G1.txt” within the same folder as the program, it will output "Invalid path, please try again." to the console and quit. If the program can find and read “G1.txt”, then it starts to fill the ArrayLists one by one. The file to read must be named as “G1.txt” without the quotes, any other file names and formats are not accepted by the program.

b) ArrayList Filling

The program fills the ArrayLists in a generalized fill() function. This function takes the file reader Scanner object, starting string, stopping string, and the ArrayList the user wants filled as parameters. Then the function starts to read from the starting string all the way until the stopping string. For example, if the user wanted to fetch the terminals from the file, then they'd have to make to function call fill(fileReader, "NON-TERMINAL", "TERMINAL", nonTerminal).

c) HashMap Initialization and Filling

After filling the ArrayLists, then the program will then proceed to initialize the rules HashMap. The initialization process just initializes the key-value pairs for the HashMap using the non-terminals ArrayList, and the values inside the HashMaps are all empty strings that will later be modified by the filler function.

When the initialization is finished, the program starts to fill the empty ArrayLists by concatenating the empty strings with actual values. (If there are multiple rules for a non-terminal, then a " | " would be concatenated before the new rule.) This process continues until reaching "START" while reading the file. At the end of all this, the HashMap can be visualized for the following table:

Non-Terminals	Rules
S	00S | 11F
F	00F | e

After the Scanner reaches "START", it reads the next line and stops reading. Then it assigns what it just read to a String called start and proceeds continue with the next steps.

d) Adding a New Start Variable

To initialize a new start variable, the program first picks a random capital letter from the English alphabet. If the letter picked already exists in the non-terminals ArrayList, then it picks other random capital letters until it finds a letter that does not exist in non-terminals ArrayList. After finding such a letter, then the program adds it to the beginning of the non-terminals ArrayList and the program also maps it to the former start variable in the rules HashMap in the form [NEW START VARIABLE] → [OLD START VARIABLE].

e) Removing Epsilons

When the program finishes its work with the start variable, then it goes through the keys of the rules HashMap in a for loop. If any of the items contain "e" as substrings, then the non-terminal of that rule is marked by assigning it to a new String "toChange". After that's done, the program simply removes the substring "| e" from the marked rule by replacing it with "". After that's done, the program looks for all the rules containing the substring toChange and it concatenates those rules with the versions without toChange. For example, S → 00S | 11F becomes S → 00S | 11F | 11. This process continues in a while loop until there are no epsilons.

f) Removing Unit Rules

Just like removing epsilons, the program again goes through all rules to find if there are any unit rules. If there's a unit rule, then it is again marked with the String toChange and after it is marked, the program then treats toChange as a non-terminal (because toChange actually is a non-terminal) and it takes its rules, and then concatenates the rules that contain toChange while also removing the unit rule toChange. This process also goes on to work in a while loop until there are no unit rules.

g) Cleaning Lowercase and Uppercase

When the unit rules are all deleted, the program first cleans the lowercase rules and then it cleans the uppercase rules. Both clean functions are very similar to each other so they will be explained as if they were the same. These clean functions begin by going through the rules HashMap and checking if there are rules of the form A → aB or the form A → BCC. If there are such rules, then the program would map them to the relevant HashMap (either lowercaseReplacements or uppercaseReplacements) with either the form X → a or the form Y → BC. (The Xs and Ys are chosen randomly from the English alphabet, using the same piece of code from section d.) After the mapping is done, then the program checks if there are any rules which need replacements. If they need replacements, then the program modifies the rule String using the replacements HashMaps, so for example S → aB | BCC would become S → XB | YC.

Before this function is used, two historical HashMaps are initialized to make sure that all uppercase and lowercase rules are cleaned. In a while loop, the program first makes the function call cleanLowercase(rules, lowercaseReplacements, nonTerminal), then the function call cleanUppercase(rules, uppercaseReplacements, nonTerminal). After the functions are done, the second historical HashMap is reinitialized with the values of the first HashMap while the first historical HashMap is reinitialized with the values of the rules HashMap. The while loop continues to work until all three HashMaps are equal.

h) Printing

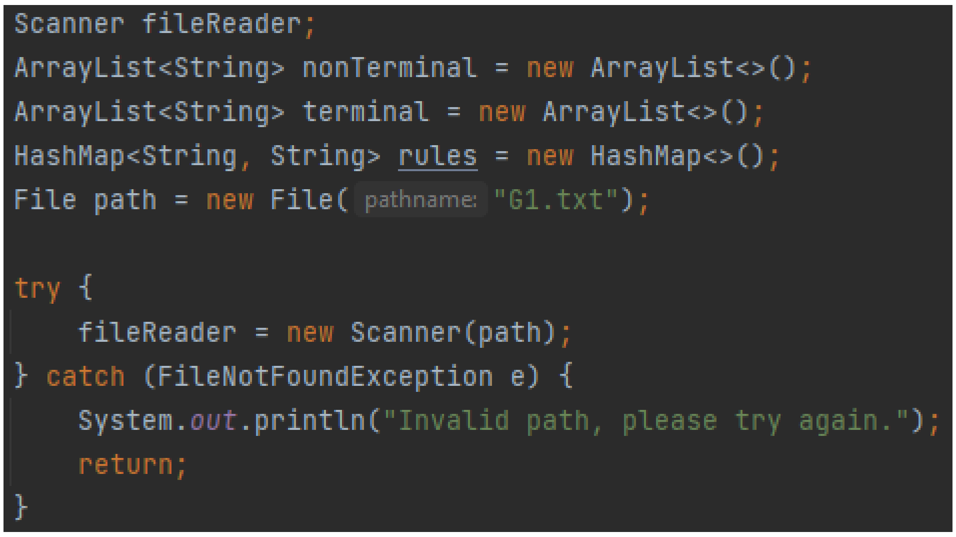
After everything is done, the program outputs the results to the console in the same format as the input like the following:

```
NON-TERMINAL
K
S
F
Y
D
I
L
TERMINAL
0
1
RULES
S:IS
S:LF
S:DD
D:1
F:IF
F:YY
Y:0
I:YY
K:IS
K:LF
K:DD
L:DD
START
K
```

4. Implementation

a) File Reading

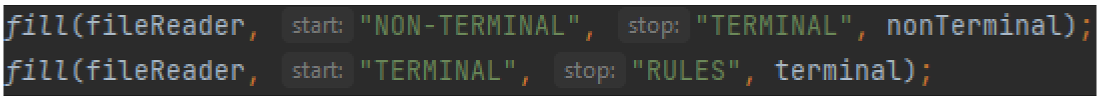
The program begins by defining the empty Scanner, ArrayList, and HashMap Objects. There are also a global Strings called last and start, which hold the last keyword accessed and the start variable. After that’s done, a File Object is defined with the path “G1.txt” since it is assumed that the file name and format is always “G1.txt” and that the input file is always in the same folder as the program. After the File Object is also initialized, then the program tries to initialize the empty Scanner Object with the File, if it catches FileNotFoundException then it must mean that “G1.txt” is not present in the folder and the program quits after outputting "Invalid path, please try again.". Below is the code for the process described:



```
Scanner fileReader;
ArrayList<String> nonTerminal = new ArrayList<>();
ArrayList<String> terminal = new ArrayList<>();
HashMap<String, String> rules = new HashMap<>();
File path = new File( pathname: "G1.txt");

try {
    fileReader = new Scanner(path);
} catch (FileNotFoundException e) {
    System.out.println("Invalid path, please try again.");
    return;
}
```

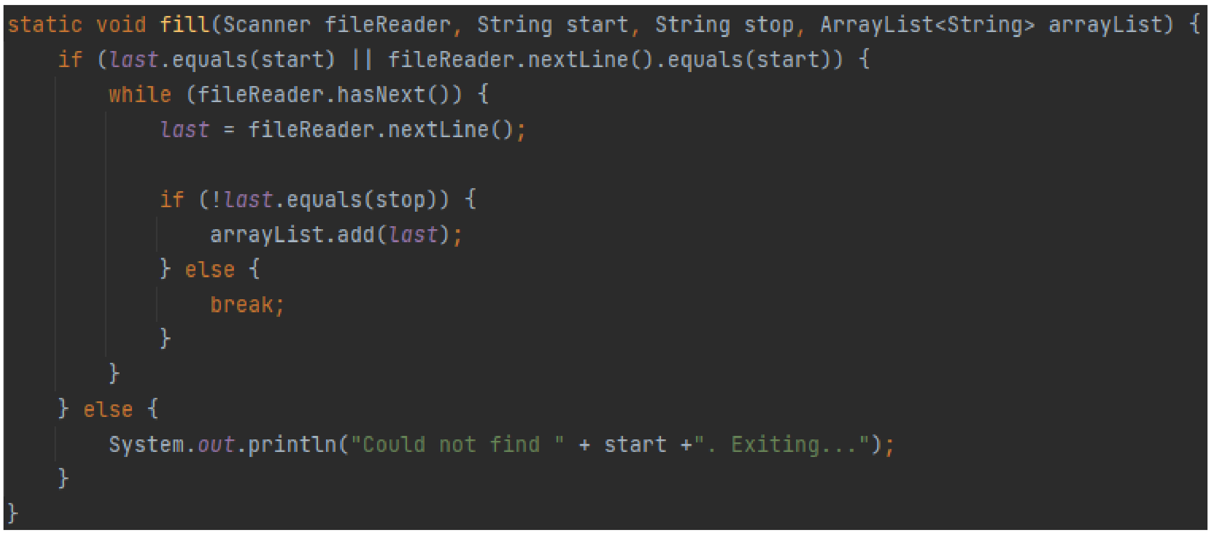
b) ArrayList Filling

After the fileReader Scanner is successfully initialized, then the program starts to use the fill() function to fill the two ArrayLists by reading “G1.txt”. The function calls are the following:



```
fill(fileReader,  start: "NON-TERMINAL",  stop: "TERMINAL", nonTerminal);
fill(fileReader,  start: "TERMINAL",  stop: "RULES", terminal);
```

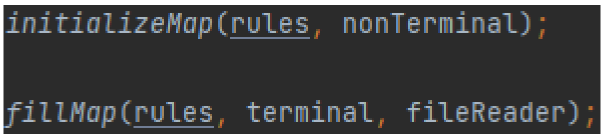
And the fill() function itself starts by first checking if the current or the next line (to avoid fencepost errors) in Scanner fileReader is on is the same as String start. If the result is false, then it outputs an error message in the console and quits. If the result is true, then it starts a while loop where it fetches all the results from the File to the ArrayList specified in the parameters, until it reaches the stop String. After reaching the stop String the loop is broken and the function returns. Below is the code for the process described:



```java
static void fill(Scanner fileReader, String start, String stop, ArrayList<String> arrayList) {
    if (last.equals(start) || fileReader.nextLine().equals(start)) {
        while (fileReader.hasNext()) {
            last = fileReader.nextLine();

            if (!last.equals(stop)) {
                arrayList.add(last);
            } else {
                break;
            }
        }
    } else {
        System.out.println("Could not find " + start +". Exiting...");
    }
}
```

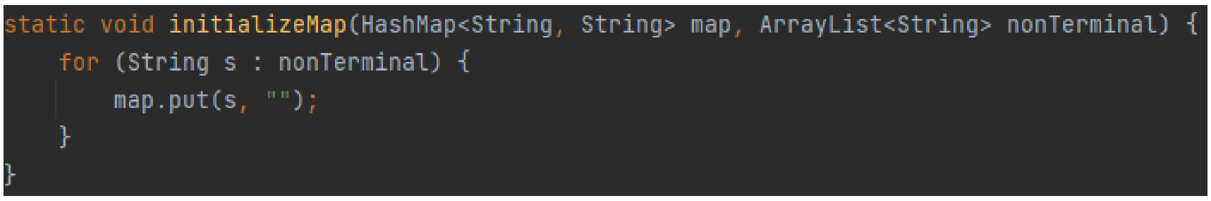
c) HashMap Initialization and Filling

After all ArrayLists are filled, the program makes the following function calls initializeMap() and fillMap(). Which are the following:



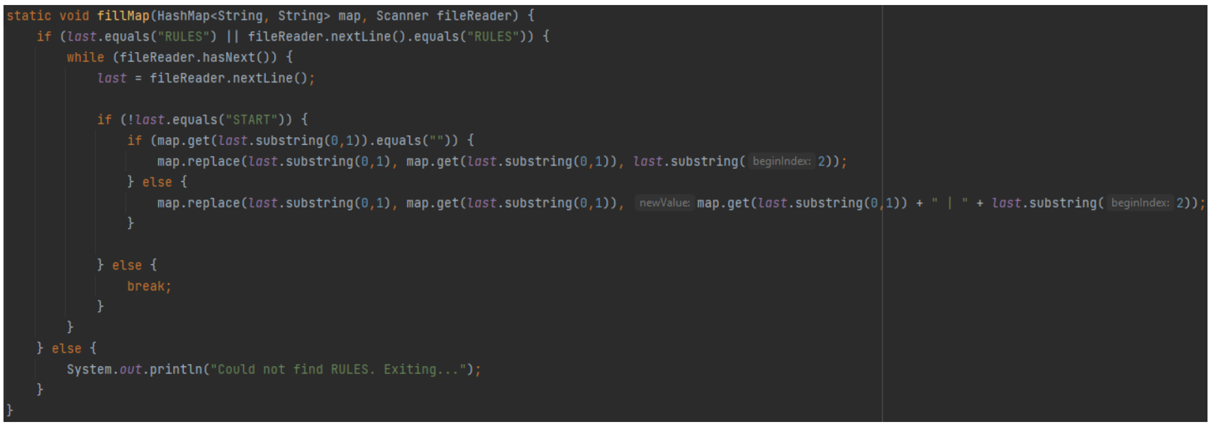
```java
initializeMap(rules, nonTerminal);

fillMap(rules, terminal, fileReader);
```

For all the non-terminals given to initializeMap() as a parameter, the program simply hashes non-terminal variables with empty rule Strings. The function ends when the for-loop ends. Below is the code for the process described:



```
static void initializeMap(HashMap<String, String> map, ArrayList<String> nonTerminal) {
    for (String s : nonTerminal) {
        map.put(s, "");
    }
}
```

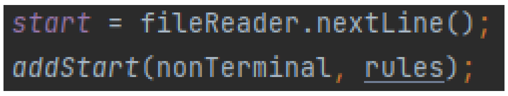
After the HashMap is initialized, fillMap() first checks if fileReader is on the keyword “RULES”, if it isn’t, then it quits with an error message. If the current keyword is indeed “RULES”, then it starts a while loop where it fetches rule from each line until “START” and concatenates the corresponding String inside the rules stored in the HashMap. Below is the code for the process described:



```java
static void fillMap(HashMap<String, String> map, Scanner fileReader) {
    if (last.equals("RULES") || fileReader.nextLine().equals("RULES")) {
        while (fileReader.hasNext()) {
            last = fileReader.nextLine();

            if (!last.equals("START")) {
                if (map.get(last.substring(0,1)).equals("")) {
                    map.replace(last.substring(0,1), map.get(last.substring(0,1)), last.substring( beginIndex: 2));
                } else {
                    map.replace(last.substring(0,1), map.get(last.substring(0,1)),  newValue: map.get(last.substring(0,1)) + " | " + last.substring( beginIndex: 2));
                }

            } else {
                break;
            }
        }
    } else {
        System.out.println("Could not find RULES. Exiting...");
    }
}
```

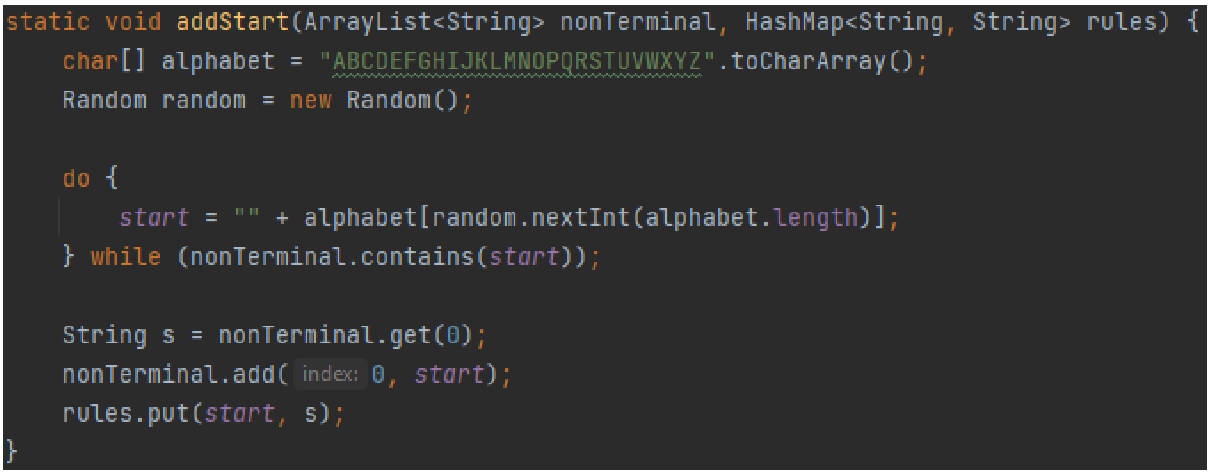
d) Adding a New Start Variable

After the rules HashMap is filled, the program takes the next line from the Scanner and it is then assigned to the String start. After that is done, the program calls the function to add a new start variable. Below is the code for the process described:



```
start = fileReader.nextLine();
addStart(nonTerminal, rules);
```

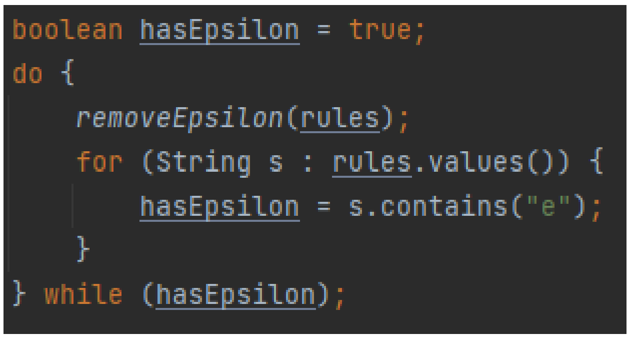
The function addStart() first initializes a char array which keeps all the capital letters in the English alphabet. After that, the program keeps trying to find a new letter for the new start variable which is not in the non-terminal ArrayList. When it finds such a letter, then it first adds it to the beginning of the non-terminals ArrayList and puts it inside the rules HashMap, with the rule [NEW START VARIABLE] → [OLD START VARIABLE]. Below is the code for the process described:



```java
static void addStart(ArrayList<String> nonTerminal, HashMap<String, String> rules) {
    char[] alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();
    Random random = new Random();

    do {
        start = "" + alphabet[random.nextInt(alphabet.length)];
    } while (nonTerminal.contains(start));

    String s = nonTerminal.get(0);
    nonTerminal.add( index: 0, start);
    rules.put(start, s);
}
```

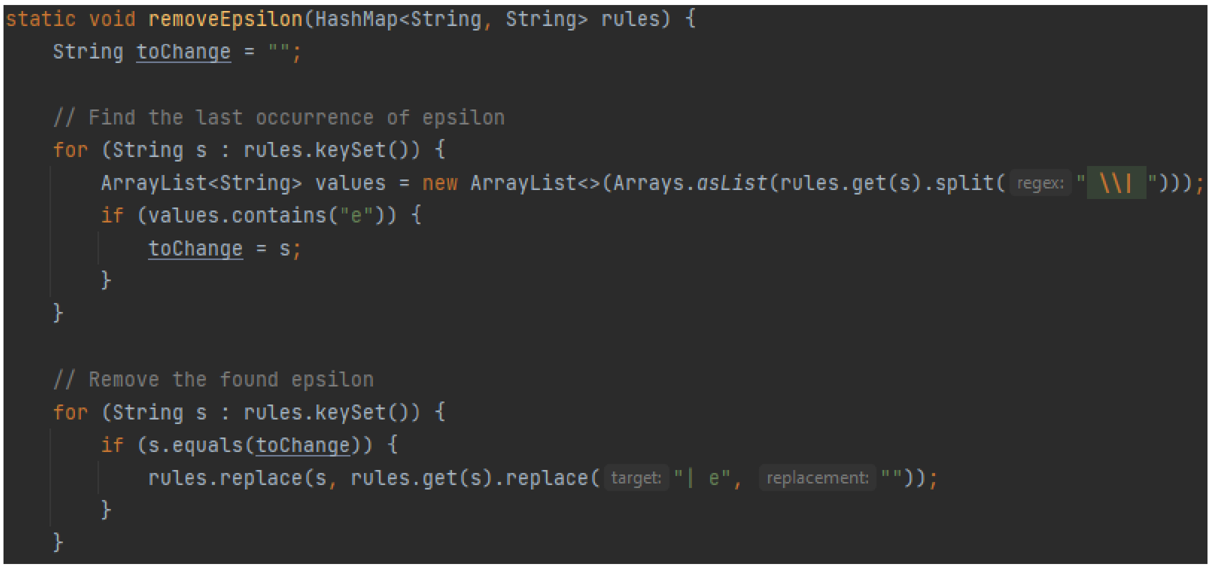
e) Removing Epsilons

After all the transitions of the new start state is added, the program calls the function removeEpsilon() in a while loop. Below is the code for the process described:



```java
boolean hasEpsilon = true;
do {
    removeEpsilon(rules);
    for (String s : rules.values()) {
        hasEpsilon = s.contains("e");
    }
} while (hasEpsilon);
```

The function first goes through the values of the rules HashMap to see if there are any rules that contain the substring "e" in a for loop. If so, then the function marks the non-terminal containing the empty rule and later removes it in the second for loop which is used for removing the empty rule. Below is the code for the process described:

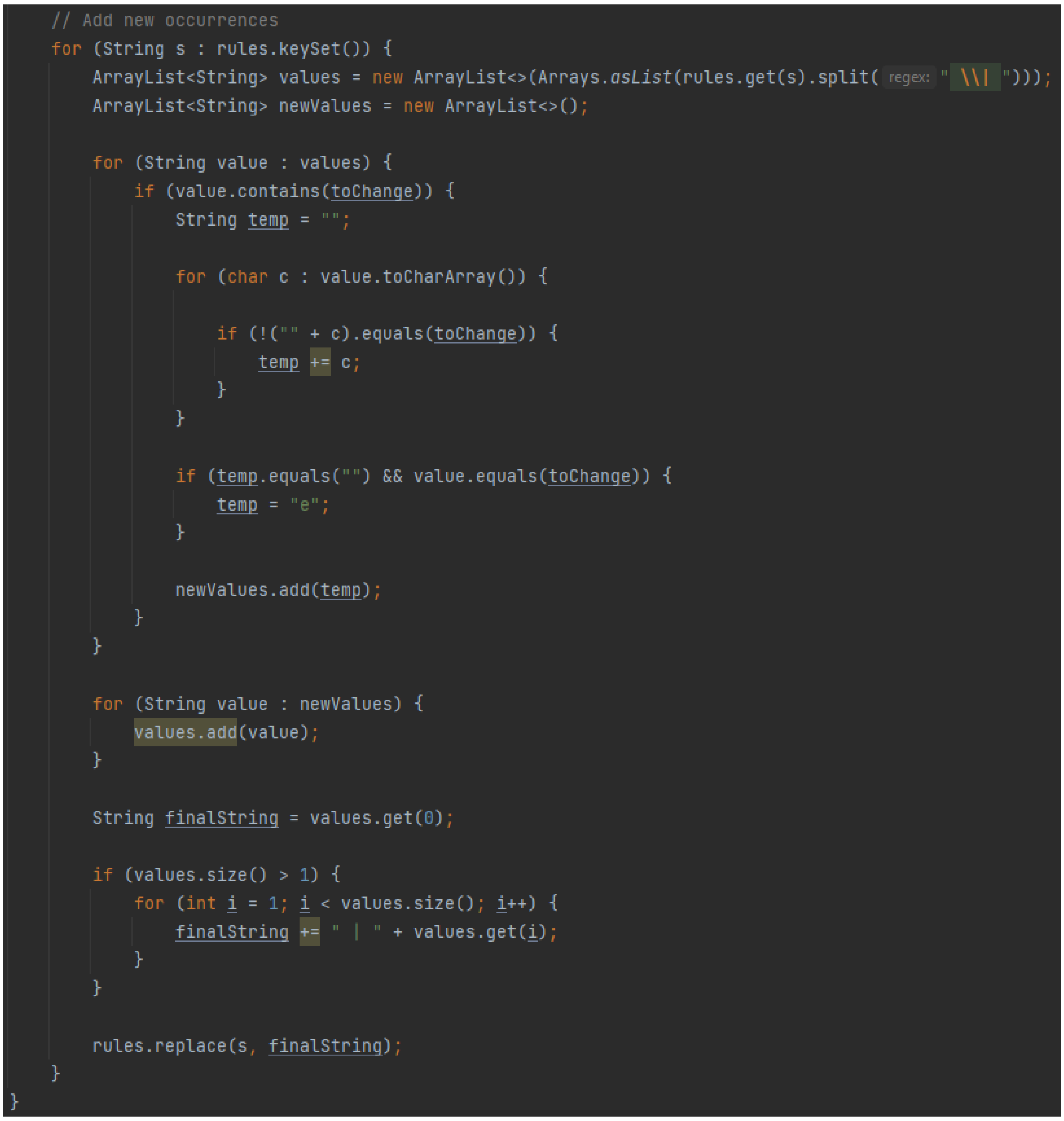


```java
static void removeEpsilon(HashMap<String, String> rules) {
    String toChange = "";

    // Find the last occurrence of epsilon
    for (String s : rules.keySet()) {
        ArrayList<String> values = new ArrayList<>(Arrays.asList(rules.get(s).split( regex: "\\|")));
        if (values.contains("e")) {
            toChange = s;
        }
    }

    // Remove the found epsilon
    for (String s : rules.keySet()) {
        if (s.equals(toChange)) {
            rules.replace(s, rules.get(s).replace( target: "| e", replacement: ""));
        }
    }
```

After removing the epsilon, the program goes through the values of the rules HashMap to see if any of them contain the marked String toChange, if so then it goes through all the letters in the rule and adds them one by one to a temporary String temp, except for the marked letter. If nothing is added into the temporary String and the part of the rule we're looking for is equal to the marked letter, then the program makes assigns “e” to the temporary String.

After the program is done working with temp, then it is added to the ArrayList newValues. And after the program is done going through all the values, then it starts adding the values from the newValues ArrayList to a temporary ArrayList which holds all the values of the rules of the current non-terminal called values.

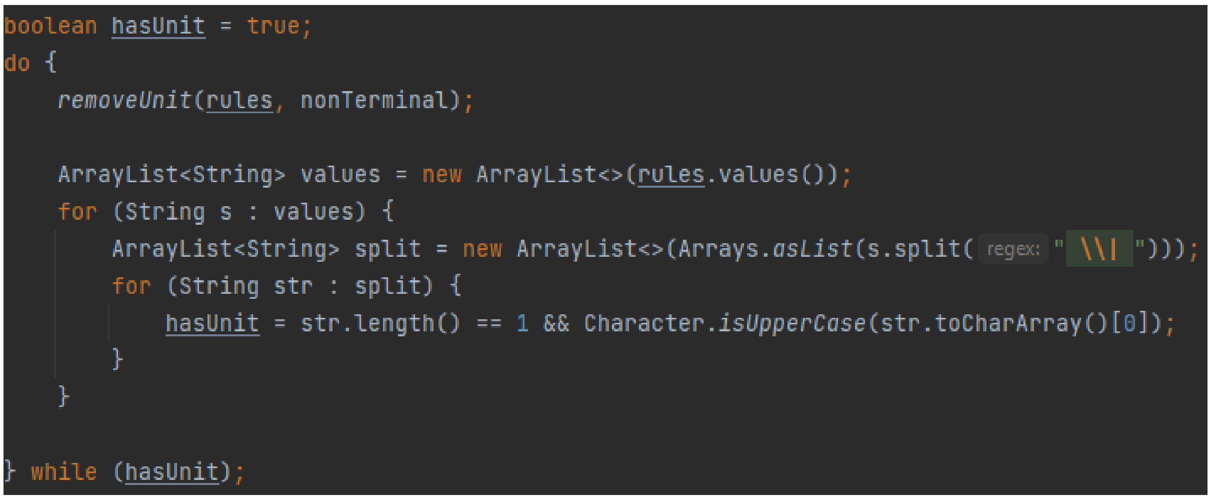
After that is done, the program turns the values ArrayList to a String using a for loop and puts it back into the rules HashMap. Below is the code for the process described:



```java
    // Add new occurrences
    for (String s : rules.keySet()) {
        ArrayList<String> values = new ArrayList<>(Arrays.asList(rules.get(s).split( regex: "\\|")));
        ArrayList<String> newValues = new ArrayList<>();

        for (String value : values) {
            if (value.contains(toChange)) {
                String temp = "";

                for (char c : value.toCharArray()) {

                    if (!("" + c).equals(toChange)) {
                        temp += c;
                    }
                }

                if (temp.equals("") && value.equals(toChange)) {
                    temp = "e";
                }

                newValues.add(temp);
            }
        }

        for (String value : newValues) {
            values.add(value);
        }

        String finalString = values.get(0);

        if (values.size() > 1) {
            for (int i = 1; i < values.size(); i++) {
                finalString += " | " + values.get(i);
            }
        }

        rules.replace(s, finalString);
    }
}
```

f) Removing Unit Rules

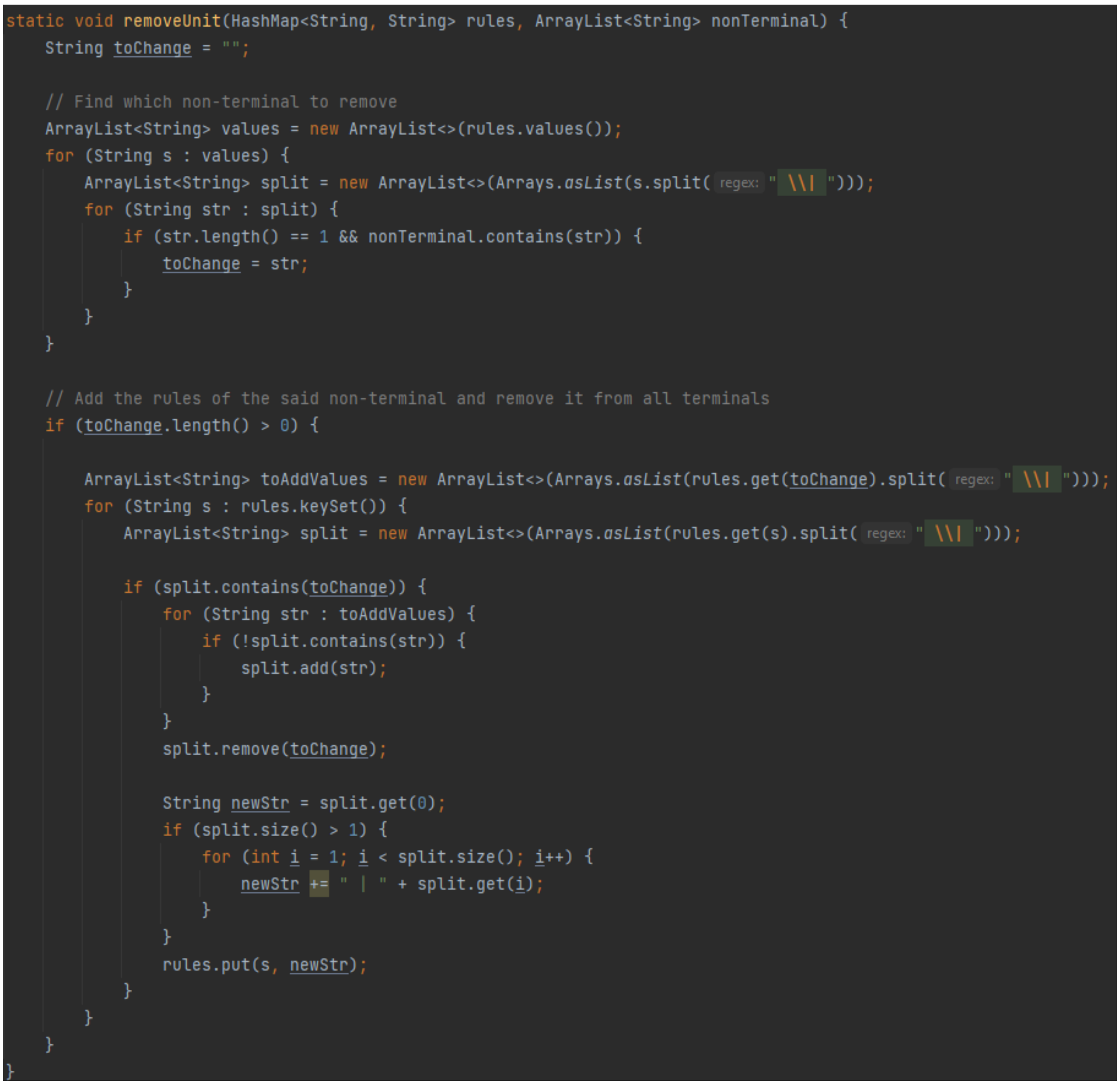
When all epsilons are removed, the program makes the function call removeUnit() and then checks if there are any unit rules by going through every rule and seeing if there are any rule String with length 1 and it is also uppercase. Below is the code for the process described:



```
boolean hasUnit = true;
do {
    removeUnit(rules, nonTerminal);

    ArrayList<String> values = new ArrayList<>(rules.values());
    for (String s : values) {
        ArrayList<String> split = new ArrayList<>(Arrays.asList(s.split( regex: "\\|")));
        for (String str : split) {
            hasUnit = str.length() == 1 && Character.isUpperCase(str.toCharArray()[0]);
        }
    }

} while (hasUnit);
```

The function removeUnit first starts by going through all the values of the rules HashMap, splits the values since they are usually of the form "AB | CD | A " to the ArrayList split with the form {AB, CD, A}, and checks if there are any non-terminal Strings in the values ArrayList with length 1. If there is such a String, then it is marked with toChange.

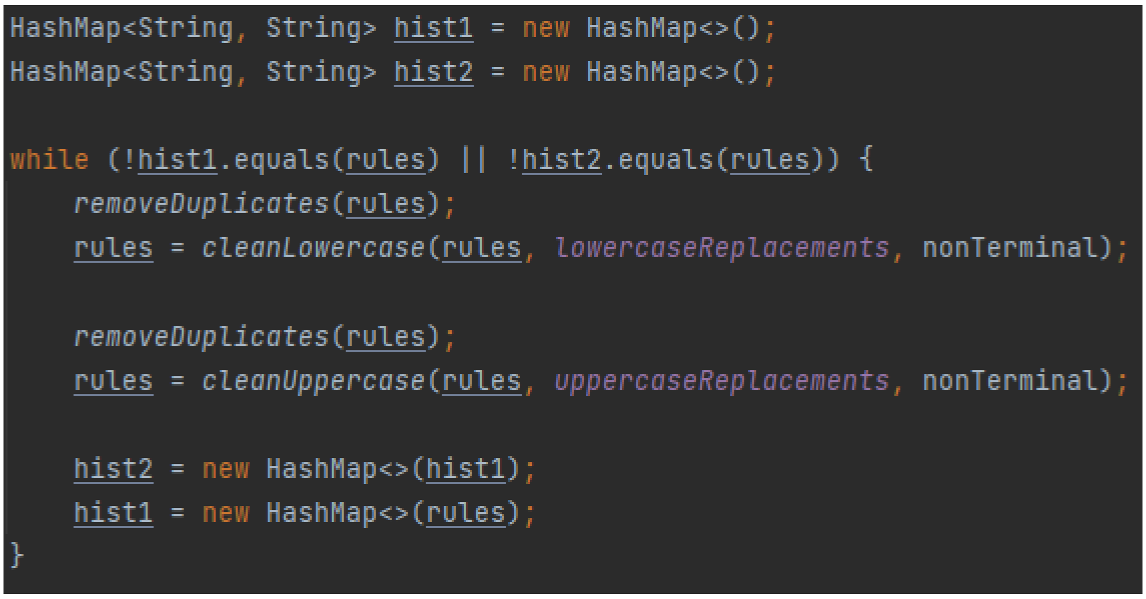
If toChange's length is bigger than 0, then the program first initializes a temporary ArrayList called toAddValues with the values of the toChange in the rules HashMap. Then, for each of the rules, the ones that don't exist in the ArrayList split are added to it. After the for loop is done, the variable toChange is removed from it since all the rules related to toChange are already added. When that is done, the split ArrayList is turned into a String newStr in a for loop and put back into the rules HashMap. Below is the code for the process described:



```java
static void removeUnit(HashMap<String, String> rules, ArrayList<String> nonTerminal) {
    String toChange = "";

    // Find which non-terminal to remove
    ArrayList<String> values = new ArrayList<>(rules.values());
    for (String s : values) {
        ArrayList<String> split = new ArrayList<>(Arrays.asList(s.split( regex: "\\|")));
        for (String str : split) {
            if (str.length() == 1 && nonTerminal.contains(str)) {
                toChange = str;
            }
        }
    }

    // Add the rules of the said non-terminal and remove it from all terminals
    if (toChange.length() > 0) {

        ArrayList<String> toAddValues = new ArrayList<>(Arrays.asList(rules.get(toChange).split( regex: "\\|")));
        for (String s : rules.keySet()) {
            ArrayList<String> split = new ArrayList<>(Arrays.asList(rules.get(s).split( regex: "\\|")));

            if (split.contains(toChange)) {
                for (String str : toAddValues) {
                    if (!split.contains(str)) {
                        split.add(str);
                    }
                }
                split.remove(toChange);

                String newStr = split.get(0);
                if (split.size() > 1) {
                    for (int i = 1; i < split.size(); i++) {
                        newStr += " | " + split.get(i);
                    }
                }
                rules.put(s, newStr);
            }
        }
    }
}
```

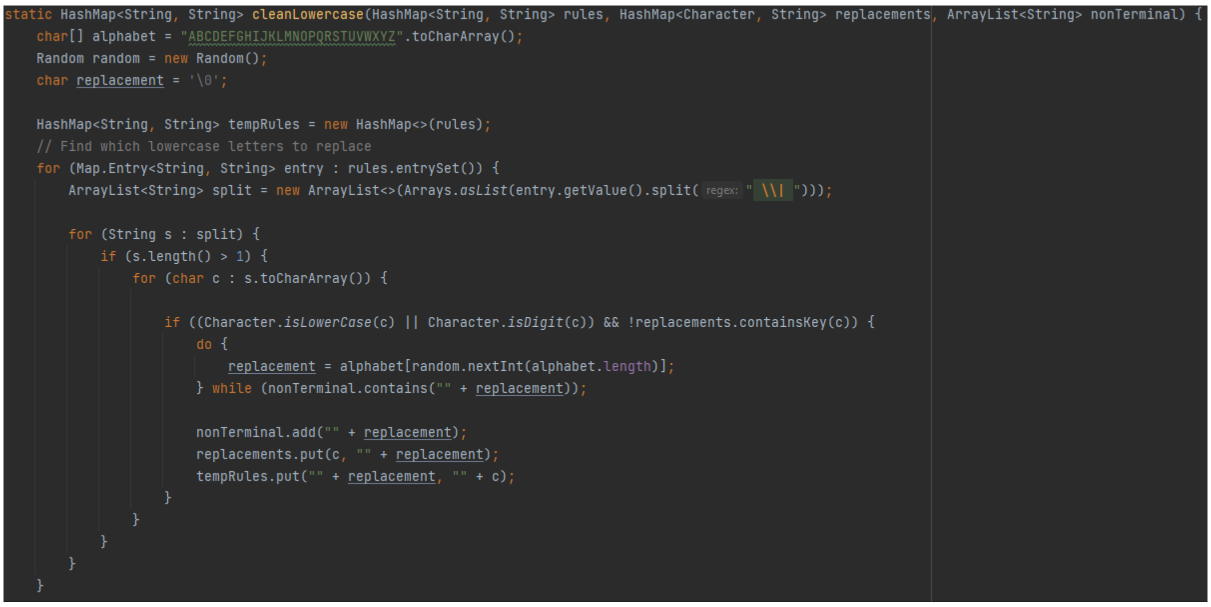
g) Cleaning Lowercase and Uppercase

When all unit rules are removed, two empty historical HashMaps (hist1 and hist2) are initialized for the rules HashMap. In a while loop, the program first calls the cleanLowercase() function and then the cleanUppercase() function. After the two functions are done, hist2 is reinitialized with the values of hist1 and hist1 is reinitialized with the values of rules. This while loop continues to work until all three HashMaps are equal. Below is the code for the process described:



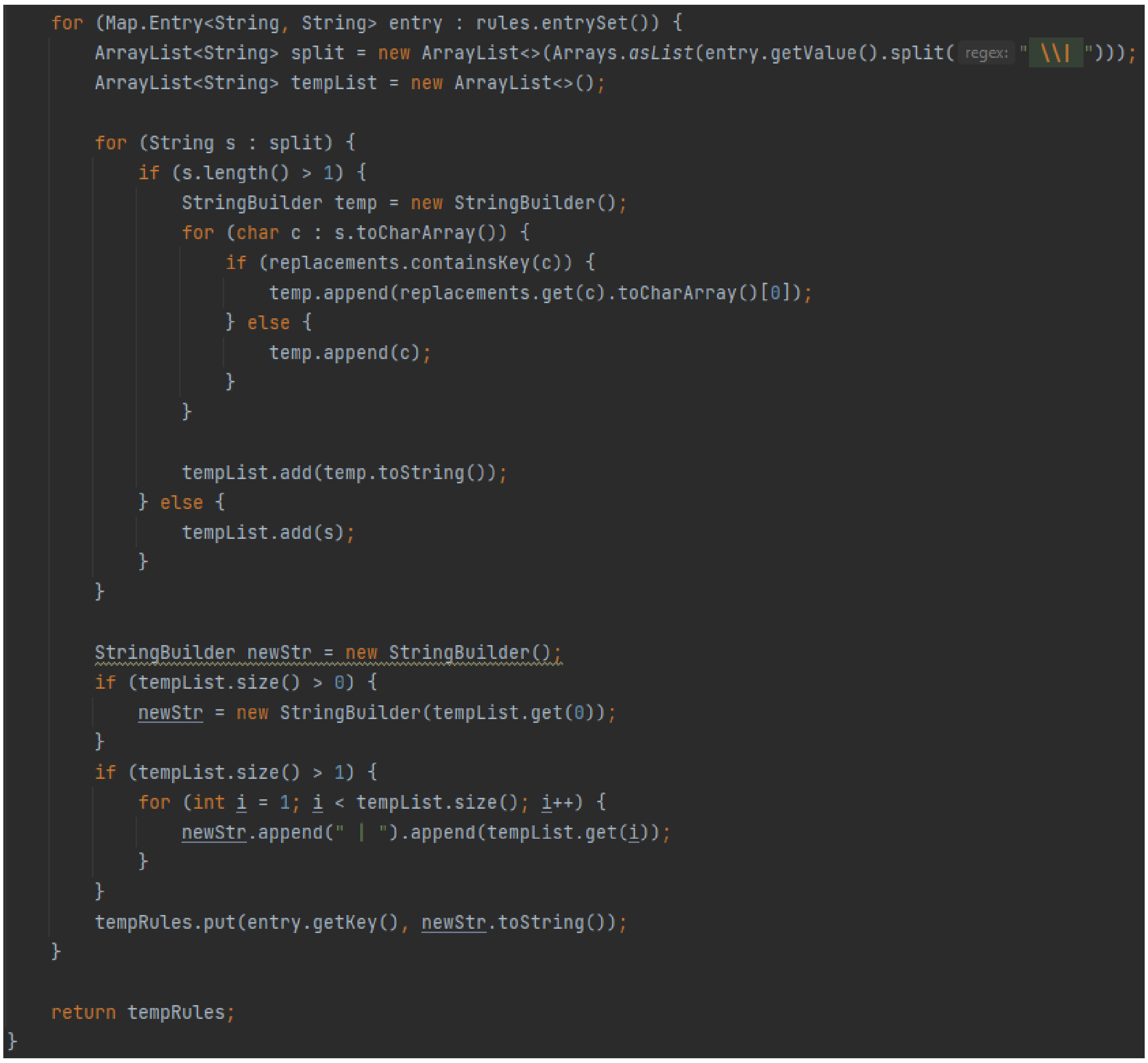
```java
HashMap<String, String> hist1 = new HashMap<>();
HashMap<String, String> hist2 = new HashMap<>();

while (!hist1.equals(rules) || !hist2.equals(rules)) {
    removeDuplicates(rules);
    rules = cleanLowercase(rules, lowercaseReplacements, nonTerminal);

    removeDuplicates(rules);
    rules = cleanUppercase(rules, uppercaseReplacements, nonTerminal);

    hist2 = new HashMap<>(hist1);
    hist1 = new HashMap<>(rules);
}
```

The function cleanLowercase first begins by defining the English alphabet and a Random Object, as done before. Then it also defines a new temporary HashMap with the values of the rules HashMap. For each of the rules, the program checks if there are any with length > 1. If there are such rules, then it generates a random letter that doesn't exist in the non-terminals ArrayList and then adds it to the non-terminals ArrayList, while also putting it into the lowercaseReplacements HashMap with the form a → X and putting it into the tempRules HashMap with the form X → a. Below is the code for the process described:



```java
static HashMap<String, String> cleanLowercase(HashMap<String, String> rules, HashMap<Character, String> replacements, ArrayList<String> nonTerminal) {
    char[] alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();
    Random random = new Random();
    char replacement = '\0';

    HashMap<String, String> tempRules = new HashMap<>(rules);
    // Find which lowercase letters to replace
    for (Map.Entry<String, String> entry : rules.entrySet()) {
        ArrayList<String> split = new ArrayList<>(Arrays.asList(entry.getValue().split( regex: "\\|")));

        for (String s : split) {
            if (s.length() > 1) {
                for (char c : s.toCharArray()) {

                    if ((Character.isLowerCase(c) || Character.isDigit(c)) && !replacements.containsKey(c)) {
                        do {
                            replacement = alphabet[random.nextInt(alphabet.length)];
                        } while (nonTerminal.contains("" + replacement));

                        nonTerminal.add("" + replacement);
                        replacements.put(c, "" + replacement);
                        tempRules.put("" + replacement, "" + c);
                    }
                }
            }
        }
    }
```

After determining all the replacements, the program goes through all the values again, but changing the values that need to be replaced using the lowercaseReplacements HashMap this time. All the values are added into the temporary tempList ArrayList, and tempList is turned into a String in a for loop and put into the tempRules HashMap. This function then returns tempRules. Below is the code for the process described:

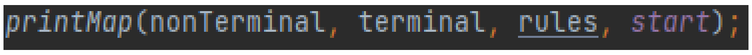


```
    for (Map.Entry<String, String> entry : rules.entrySet()) {
        ArrayList<String> split = new ArrayList<>(Arrays.asList(entry.getValue().split( regex: " \\| ")));
        ArrayList<String> tempList = new ArrayList<>();

        for (String s : split) {
            if (s.length() > 1) {
                StringBuilder temp = new StringBuilder();
                for (char c : s.toCharArray()) {
                    if (replacements.containsKey(c)) {
                        temp.append(replacements.get(c).toCharArray()[0]);
                    } else {
                        temp.append(c);
                    }
                }

                tempList.add(temp.toString());
            } else {
                tempList.add(s);
            }
        }

        StringBuilder newStr = new StringBuilder();
        if (tempList.size() > 0) {
            newStr = new StringBuilder(tempList.get(0));
        }
        if (tempList.size() > 1) {
            for (int i = 1; i < tempList.size(); i++) {
                newStr.append(" | ").append(tempList.get(i));
            }
        }
        tempRules.put(entry.getKey(), newStr.toString());
    }

    return tempRules;
}
```

The same process is also used for uppercase variables. The only difference is the usage of more String operations than the char operations since we deal with Strings bigger than length 1.

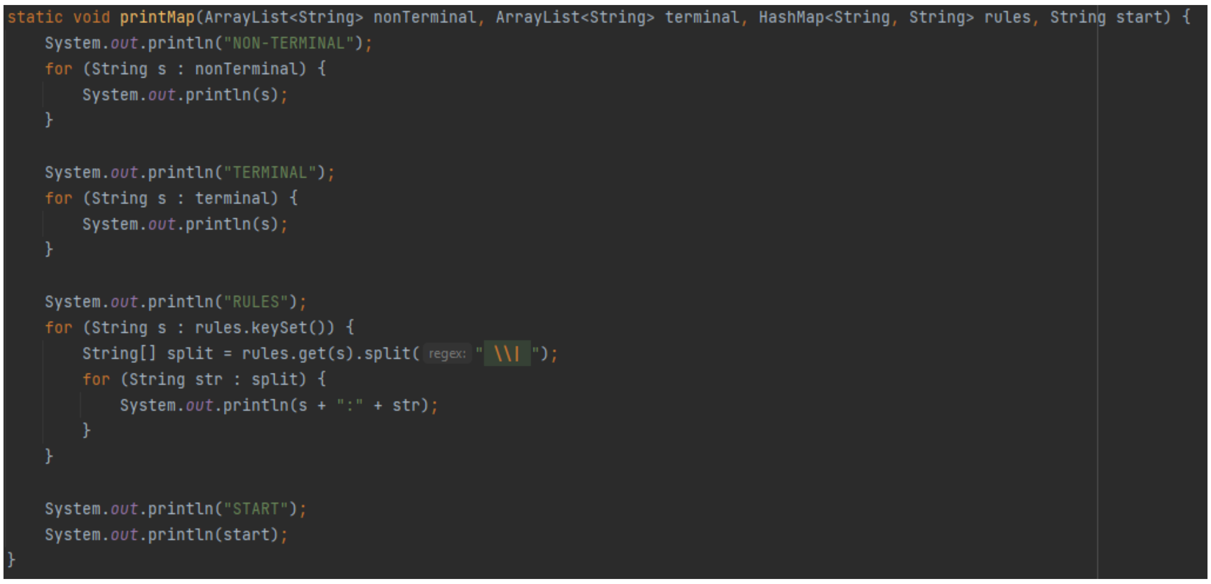
f) Printing

After every operation is done, the program makes the function call printMap() to print the results to the console. Below is the code for the process described:



```java
printMap(nonTerminal, terminal, rules, start);
```

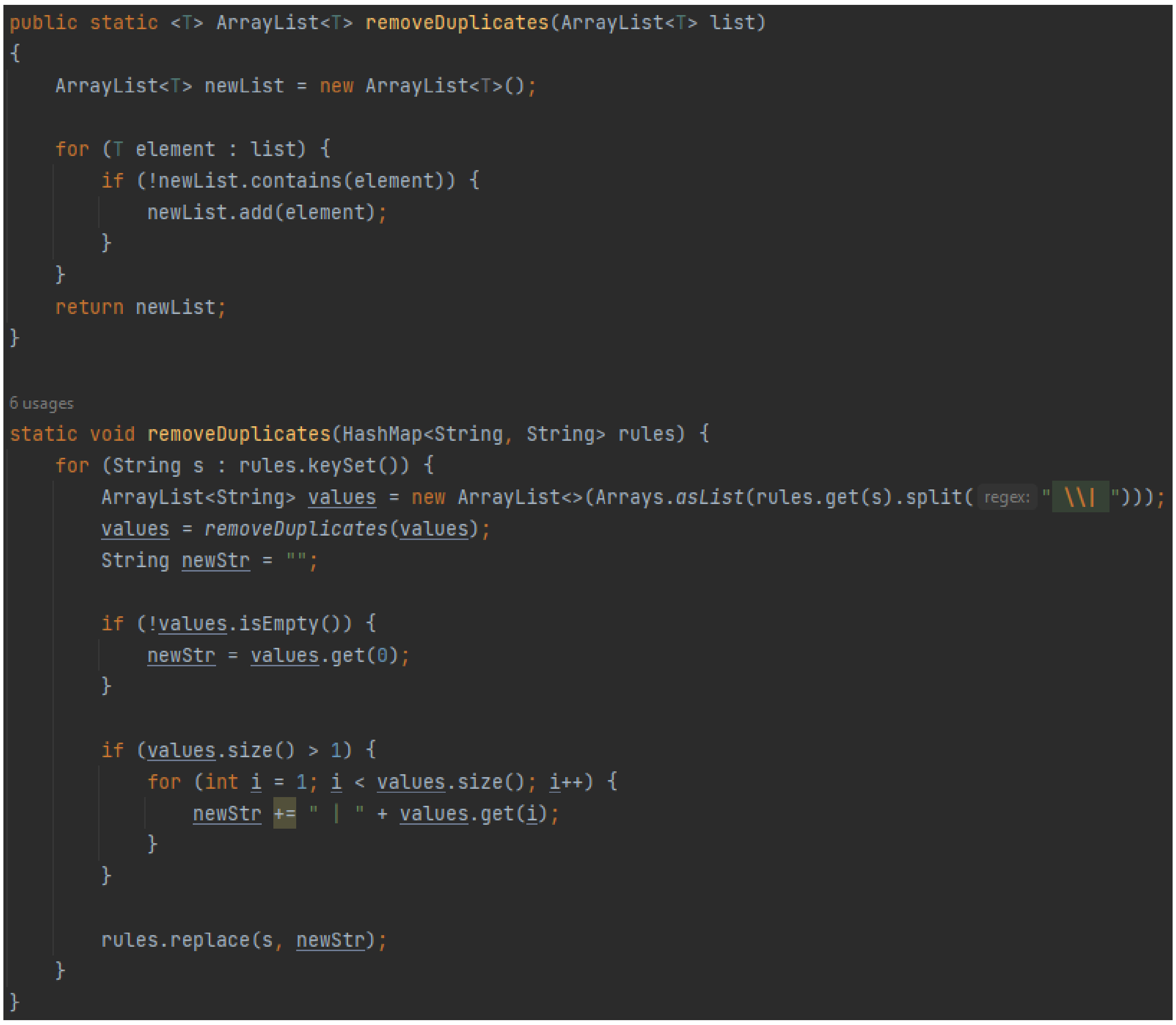
The function printMap() takes every ArrayList the program used so far along with the rules HashMap as parameters, after which it iterates over each of them and prints results using for loops with the following code:



```java
static void printMap(ArrayList<String> nonTerminal, ArrayList<String> terminal, HashMap<String, String> rules, String start) {
    System.out.println("NON-TERMINAL");
    for (String s : nonTerminal) {
        System.out.println(s);
    }

    System.out.println("TERMINAL");
    for (String s : terminal) {
        System.out.println(s);
    }

    System.out.println("RULES");
    for (String s : rules.keySet()) {
        String[] split = rules.get(s).split( regex: "\\|");
        for (String str : split) {
            System.out.println(s + ":" + str);
        }
    }

    System.out.println("START");
    System.out.println(start);
}
```

h) Utility Functions

The functions <T> removeDuplicates() (Which removes duplicate items in an ArrayList.) and void removeDuplicates() (Which removes duplicate items in a HashMap.) are simple functions that are used for making minor changes. Below is the code for the process described:



```java
public static <T> ArrayList<T> removeDuplicates(ArrayList<T> list)
{
    ArrayList<T> newList = new ArrayList<T>();

    for (T element : list) {
        if (!newList.contains(element)) {
            newList.add(element);
        }
    }
    return newList;
}

6 usages
static void removeDuplicates(HashMap<String, String> rules) {
    for (String s : rules.keySet()) {
        ArrayList<String> values = new ArrayList<>(Arrays.asList(rules.get(s).split( regex: "\\|")));
        values = removeDuplicates(values);
        String newStr = "";

        if (!values.isEmpty()) {
            newStr = values.get(0);
        }

        if (values.size() > 1) {
            for (int i = 1; i < values.size(); i++) {
                newStr += " | " + values.get(i);
            }
        }

        rules.replace(s, newStr);
    }
}
```

5. Results

The results the program outputs after it finishes running G1.txt and G2.txt (which was renamed to G1.txt for the program to run) are the following:

G1.txt	G2.txt
NON-TERMINAL D S F C H N L TERMINAL 0 1 RULES S:NS S:LF S:HH C:0 D:NS D:LF D:HH F:NF F:CC H:1 L:HH N:CC START D	NON-TERMINAL J S A B E Y TERMINAL a b RULES A:YB B:AE B:b B:a S:a S:EA S:AE S:b E:a Y:EB J:a J:EA J:AE J:b START J