# CS 410 – Automata Theory and Formal Languages

**Fall 2022**

## Project I Report

**Murat Can Altun**

## 1. Introduction

For the conversion from NFAs to DFAs, it is a common practice to use a transition table, like the one below for NFA1.txt:

| States/Symbols | 0 | 1 |
|---|---|---|
| A | A | AB |
| B | Ø | C |
| C | Ø | Ø |
| AB | A | ABC |
| ABC | A | ABC |

To replicate this solution digitally, mapping the transitions to states in Java is one of the most convenient and time efficient approaches. 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 alphabet, states, and start/final state data. An inner ArrayList is also nested inside the transitions HashMap to store the transitions from the key of the map (which is the actual state it's originating from). The size of the inner ArrayList is exactly as big as the alphabet ArrayList, since each element in the inner ArrayList corresponds to an alphabet symbol. The specific definitions for the ArrayLists are the following:

- ArrayList<String> alphabet
- ArrayList<String> states
- ArrayList<String> startStop

**HashMap**: Used for storing and using the transition functions. As explained above, each key holds an ArrayList that holds all the possible transitions from itself. For example, if we wanted to hash A from the table above, it would be hashed as "{A, {A, AB}}", or if we wanted to hash B, it would be hashed as "{B, {, C}}". And the HashMap is defined like this:

- HashMap<String, ArrayList<String>> transitions

## 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 NFA1.txt, and if there is no "NFA1.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 "NFA1.txt", then it starts to fill the ArrayLists one by one. The file to read must be named as "NFA1.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 states from the file, then they'd have to make to function call fill(fileReader, "STATES", "START", states).

### c) HashMap Initialization and Filling

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

When the initialization is finished, the program starts to fill the empty ArrayLists by concatenating the empty strings with actual values. This process continues until reaching "END" while reading the file. At the end of all this, the HashMap can be visualized for the following table: (Continuing from the previous example.)

| States/Symbols | 0 | 1 |
|---|---|---|
| A | A | AB |
| B | Ø | C |
| C | Ø | Ø |

However, inputting 1 into A yields AB and the program still needs to tell the user what happens when the transition to AB happens. Which leads to the extra initialization and filling part.

### d) Extra Initialization and Filling

Something like what was just done can be done in a while loop to make sure that the system continues to find the transitions and states until it keeps getting the same result. For this, the program creates two more completely empty historical data HashMaps, in a while loop it takes the states which it created by concatenating two states, inserts them into the states ArrayList and then does a similar initialization and filling operation similar to the one explained above. After each iteration of the while loop, historical data HashMap 2 is redefined with the values of historical data HashMap 1 and HashMap 1 is redefined as the transactions HashMap the system just acquired. This process goes on until all three of these HashMaps hold the same values. Which generates the following table:

| States/Symbols | 0 | 1 |
|---|---|---|
| A | A | AB |
| B | Ø | C |
| C | Ø | Ø |
| AB | A | ABC |
| ABC | A | ABC |

### e) Final States

When the program finishes its work with the state transitions, then it goes through the keys of the HashMap in a for loop. If any of the items contain the final states initially given as substrings, then they are also added as final states to the startStop ArrayList as well.

### f) Printing

After everything is done, the program outputs the results to the console like the following:

```
ALPHABET
0
1
STATES
A
B
C
AB
ABC
START
A
FINAL
C
ABC
TRANSITIONS
A 0 A
A 1 AB
AB 0 A
AB 1 ABC
B 1 C
ABC 0 A
ABC 1 ABC
END
```

## 4. Implementation

### a) File Reading

The program begins by defining the empty Scanner, ArrayList, and HashMap Objects. There is also a global String called last, which holds the last keyword accessed. After that's done, a File Object is defined with the path "NFA1.txt" since it is assumed that the file name and format is always "NFA1.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 "NFA1.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> alphabet = new ArrayList<>();
ArrayList<String> states = new ArrayList<>();
ArrayList<String> startStop = new ArrayList<>();
HashMap<String, ArrayList<String>> transitions = new HashMap<>();
File path = new File( pathname: "NFA1.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() to fill the three ArrayLists by reading "NFA1.txt". The function calls are the following:

```
// Fill the alphabet and states arraylists
fill(fileReader,  start: "ALPHABET",  stop: "STATES", alphabet);
fill(fileReader,  start: "STATES",  stop: "START", states);

// Assign the starting state to startStop member 0, assign the final state to startStop member 1 onwards
fill(fileReader,  start: "START",  stop: "FINAL", startStop);
fill(fileReader,  start: "FINAL",  stop: "TRANSITIONS", startStop);
```

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:

```
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:

```
// Initialize a map for the transitions
// The map works with pairs like the following:
// The state names are used as keys, and their values are lists that hold
// the transitions for ALL the symbols in the alphabet
initializeMap(transitions, states, alphabet.size());

// Fill the map from the transitions
fillMap(transitions, alphabet, fileReader);
```

For all the states given to initializeMap() as a parameter, the program first creates an empty ArrayList of size equal to the size of the alphabet and then fills it with empty Strings. After the entirety of the ArrayList is filled, it is then put into the HashMap as a value with the corresponding state as the key. The function ends when the for loop ends. Below is the code for the process described:

```
static void initializeMap(HashMap<String, ArrayList<String>> map, ArrayList<String> states, int alphabetLength) {
    for (String s : states) {
        ArrayList<String> keys = new ArrayList<>();

        for (int i = 0; i < alphabetLength; i++) {
            keys.add("");
        }

        map.put(s, keys);
    }
}
```

After the HashMap is initialized, a Scanner called stringScanner is created to read the lines fetched by the Scanner fileReader. The reasoning for creating such a Scanner is to make sure that no data is lost while reading state transitions and to make the input reading less confusing: fileReader reads a file line by line and stringScanner reads a String word by word. After stringScanner is defined, fillMap() first checks if fileReader is on the keyword “TRANSITIONS”, if it isn’t, then it quits with an error message. If the current keyword is indeed “TRANSITIONS”, then it starts a while loop where it fetches the key, symbol, and the value from each line until “END” and concatenates the corresponding value inside the inner ArrayList which is also stored in the HashMap. The extra ArrayLists tempList and newList are used due to Java’s type safety. Below is the code for the process described:

```
static void fillMap(HashMap<String, ArrayList<String>> map, ArrayList<String> alphabet, Scanner fileReader) {
    Scanner stringScanner;

    if (last.equals("TRANSITIONS") || fileReader.nextLine().equals("TRANSITIONS")) {
        while (fileReader.hasNext()) {
            last = fileReader.nextLine();

            if (!last.equals("END")) {
                stringScanner = new Scanner(last);

                String key = stringScanner.next();
                String symbol = stringScanner.next();
                String value = stringScanner.next();

                ArrayList<String> tempList = map.get(key);
                ArrayList<String> newList = new ArrayList<>(tempList);

                newList.set(alphabet.indexOf(symbol), newList.get(alphabet.indexOf(symbol)) + value);

                map.replace(key, tempList, newList);
            } else {
                break;
            }
        }
    } else {
        System.out.println("Could not find TRANSITIONS. Exiting...");
    }
}
```

**d) Extra Initialization and Filling**

After the HashMap is filled for the first time, the two historical HashMaps are initialized as empty maps. When the initialization is complete, then a while loop that continues to iterate until the all three HashMaps(transitions, transition_hist1 and transition_hist2) are equal. The body of this while loop makes the function calls initializeExtra() and fillExtra(), after which the transition_hist are modified. (transition_hist1 is always the last iteration, while transition_hist2 is always the one before transition_hist1.) Below is the code for the process described:

```
// Add the extra states for symbols like AB, BC, AC, ABC
HashMap<String, ArrayList<String>> transition_hist1 = new HashMap<>();
HashMap<String, ArrayList<String>> transition_hist2 = new HashMap<>();

while (!transition_hist1.equals(transitions) || !transition_hist2.equals(transitions)) {
    initializeExtra(transitions, states, alphabet.size());
    fillExtra(transitions, states);

    transition_hist2 = new HashMap<>(transition_hist1);
    transition_hist1 = new HashMap<>(transitions);
}
```

The function initalizeExtra() works very similar to the function initializeMap(), the only difference is that it also checks for duplicate keys (Otherwise the program keeps working and causes a stack overflow.). After initializeExtra() finishes its for loop that is used for searching for extra keys, it puts them inside the transitions HashMap with empty String values. Below is the code for the process described:

```
static void initializeExtra(HashMap<String, ArrayList<String>> map, ArrayList<String> states, int alphabetLength) {
    HashMap<String, ArrayList<String>> tempMap = new HashMap<>();
    for (Map.Entry<String, ArrayList<String>> set : map.entrySet()) {
        ArrayList<String> temp = new ArrayList<>(set.getValue());

        temp.removeAll(states);

        if (temp.size() > 0) {
            for (String s : temp) {
                if (s.length() > 1) {
                    ArrayList<String> keys = new ArrayList<>();

                    String trimmed = removeDuplicate(s.toCharArray(), s.length());

                    for (int i = 0; i < alphabetLength; i++) {
                        keys.add("");
                    }
                    tempMap.put(trimmed, keys);
                }
            }
        }
    }
    tempMap.forEach(map::putIfAbsent);
}
```

When the extra symbols are initialized, then the function fillExtra() (Again, which is also very similar to fillMap()) checks for duplicate values and puts the necessary values inside the HashMap transitions. The values inside transitions are also sorted for the sake of simplicity. Below is the code for the process described:

```
static void fillExtra(HashMap<String, ArrayList<String>> map, ArrayList<String> states) {
    HashMap<String, ArrayList<String>> tempMap = new HashMap<>(map);

    for (Map.Entry<String, ArrayList<String>> set : map.entrySet()) {
        if (!states.contains(set.getKey())) {
            ArrayList<String> temp = new ArrayList<>(set.getValue());
            int size = temp.size();

            for (String s : states) {
                if (set.getKey().contains(s)) {
                    temp.addAll(map.get(s));
                }
            }

            for (int i = 0; i < temp.size(); i++) {
                int modulo = i % size;
                if(!temp.get(modulo).contains(temp.get(i))) {
                    temp.set(modulo, temp.get(modulo) + temp.get(i));
                }
            }
            ArrayList<String> temp2 = new ArrayList<>();

            for (int i = 0; i < size; i++) {
                String deleted = removeDuplicate(temp.get(i).toCharArray(), temp.get(i).length());
                sortString(deleted);
                temp2.add(deleted);
            }
            temp2.set(0, sortString(temp2.get(0)));
            tempMap.put(set.getKey(), temp2);
            states.add(set.getKey());
        }
    }
    tempMap.forEach(map::put);
}
```

### e) Final States

After all three transition HashMaps are equal, then the program initializes a temporary ArrayList tmp. For all of the states in the HashMap transitions, if any of the stopping states are a substring then they are added into tmp. After that, all elements of tmp that are not members of the ArrayList startStop are added to startStop as well. Since there is only 1 start states and there can be many final states, it is always assumed that the first element of startStop is the start state while all the rest are final states. Below is the code for the process described:

```
ArrayList<String> tmp = new ArrayList<>();

for (String s : transitions.keySet()) {
    for (int i = 1; i < startStop.size(); i++) {
        if (s.contains(startStop.get(i))) {
            tmp.add(s);
        }
    }
}

for (String s : tmp) {
    if (!startStop.contains(s)) {
        startStop.add(s);
    }
}
```

### f) Printing

When the two for loops are done working, the program makes the final function call to print the output, which is:

```
printMap(alphabet, states, startStop, transitions);
```

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

```
static void printMap(ArrayList<String> alphabet, ArrayList<String> states, ArrayList<String> startStop, HashMap<String, ArrayList<String>> transitions) {
    System.out.println("ALPHABET");
    for (String s : alphabet) {
        System.out.println(s);
    }

    System.out.println("STATES");
    for (String s : states) {
        System.out.println(s);
    }

    System.out.println("START");
    System.out.println(startStop.get(0));

    System.out.println("FINAL");
    for (int i = 1; i < startStop.size(); i++) {
        System.out.println(startStop.get(i));
    }

    System.out.println("TRANSITIONS");
    for (Map.Entry<String, ArrayList<String>> set : transitions.entrySet()) {
        for (int i = 0; i < set.getValue().size(); i++) {
            if (!set.getValue().get(i).equals("")) {
                System.out.println(set.getKey() + " " + alphabet.get(i) + " " + set.getValue().get(i));
            }
        }
    }

    System.out.println("END");
}
```

### g) Utility Functions

The functions removeDuplicate() (Which removes duplicate characters in a String.) and sortString() (Which simply sorts the String alphabetically.) are simple functions that are used for making minor changes to Strings. removeDuplicate() is used inside initializeExtra() and fillExtra(), while sortString() is only used inside fillExtra().

```
static String removeDuplicate(char str[], int n)
{
    int index = 0;
    for (int i = 0; i < n; i++) {
        int j;
        for (j = 0; j < i; j++) {
            if (str[i] == str[j]) {
                break;
            }
        }
        if (j == i) {
            str[index++] = str[i];
        }
    }
    return String.valueOf(Arrays.copyOf(str, index));
}

2 usages
public static String sortString(String inputString)
{
    char tempArray[] = inputString.toCharArray();
    Arrays.sort(tempArray);
    return new String(tempArray);
}
```

## 5. Results

The results the program outputs after it finishes running NFA1.txt and NFA2.txt (which was renamed to NFA1.txt for the program to run) are the following:

NFA1.txt

```
ALPHABET
0
1
STATES
A
B
C
AB
ABC
START
A
FINAL
C
ABC
TRANSITIONS
A 0 A
A 1 AB
AB 0 A
AB 1 ABC
B 1 C
ABC 0 A
ABC 1 ABC
END
```

NFA2.txt

```
ALPHABET
0
1
STATES
A
B
C
BC
AB
ABC
START
A
FINAL
C
BC
ABC
TRANSITIONS
A 0 A
A 1 BC
BC 0 ABC
BC 1 B
AB 0 ABC
AB 1 BC
B 0 BC
ABC 0 ABC
ABC 1 BC
C 0 AB
C 1 B
END
```