CS 410 – Automata Theory and Formal Languages

Fall 2022

Project III Report

Murat Can Altun

1. Introduction

The Turing Machine we want to implement works like a very simple state machine with a read/write memory. Since this memory can read/write any part (Meaning it doesn't work exactly like a stack, we can modify the elements in the middle as well.) it makes sense to use and manipulate lists. Working with lists in Java is relatively easier than in C++, and so Java is chosen in the implementation of this project. 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 all the data. The alphabet, blank, states, start-accept-reject, transitions and detect ArrayLists are used for input reading while the route ArrayList is used for printing the route. The first element of start-accept-reject is the starting state, the second is the accept state and the third is the reject state. The tape ArrayList is for internal uses only and will be explained in the following sections. The specific definitions for the ArrayLists are the following:

- ArrayList<String> inputAlphabet
- ArrayList<String> tapeAlphabet
- ArrayList<String> blank
- ArrayList<String> states
- ArrayList<String> startAcceptReject
- ArrayList<String> transitions
- ArrayList<String> detect
- ArrayList<String> route
- ArrayList<Character> tape

3. Input File Structure

The input file structure is like the input file structures in project I and II. It has titles for each of the different types of input, and the inputs for the titles “BLANK”, “START”, “ACCEPT”, “REJECT” and “DETECT” are assumed to be of only one line, while the file ends with a title called “END”. The whole file is structured like the following:

```
INPUT
0
TAPE
0
X
BLANK
b
STATES
q1
q2
q3
q4
q5
qA
qR
START
q1
ACCEPT
qA
REJECT
qR
TRANSITIONS
q1 0 b R q2
q1 b b R qR
q1 X X R qR
q2 0 X R q3
q2 X X R q2
q2 b b R qA
q3 X X R q3
q3 0 0 R q4
q3 b b L q5
q4 X X R q4
q4 0 X R q3
q4 b b R qR
q5 0 0 L q5
q5 X X L q5
q5 b b R q2
DETECT
000
END
```

4. 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 Input_MURAT_CAN_ALTUN_S021707.txt, and if there is no “Input_MURAT_CAN_ALTUN_S021707.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 “Input_MURAT_CAN_ALTUN_S021707.txt”, then it starts to fill the ArrayLists one by one. The file to read must be named as “Input_MURAT_CAN_ALTUN_S021707.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) Tape Read/Write Preparation

Before starting to go over the transition functions in a loop, the program needs to prepare a few things so that the loop works properly and isn't infinite. The following steps are needed:

- Initializing a pointer (Not a real pointer, just a String which holds the same name.) for the current state.
- Taking the String to be detected and turning it into a char ArrayList, which will be treated as the tape.
- Adding a blank symbol to the end of the tape. (So that the program won't go beyond the bounds of the tape ArrayList during the transitions.)
- Adding the initial state to the route ArrayList.
- Initializing the stringScanner Scanner Object. (How it works will be explained later.)
- Initializing two historical tapes for the loop.
- Initializing a count integer to detect loops.

d) Tape Read/Write and Transitions

In a while loop which continues to work until either the ArrayLists tape, tape_hist1 and tape_hist2 are equal or count is larger than the size of the transitions ArrayList squared. The reasoning behind the count condition is to make sure that the program can detect loops, the square of the size of the transitions ArrayList is chosen because it is a high number that in theory should not be reached. Choosing a hard coded number like 100 or 1,000,000 is dangerous to use because the transitions alphabet might be large, which would result in the program stopping execution before it is done.

After the loop begins, the first thing it does is to increment the variable count. After incrementing it, it begins a for loop where it iterates over all the transitions. In this for loop, the object stringScanner is initialized to scan the whole line of transition. For example, it is set to scan the line "q4 0 X R q3". stringScanner scans the first String in the line which is "q4" in our example, and checks if it is equal to the current state pointer. If it is, then the stringScanner scans again to get the second String which is "0" in our example, then the program uses the static integer currentIndex to check if the index the memory is on is equal to what stringScanner just read. If the index and the second String are the same, then the Scanner reads the line three times at the same time. The first one is the new value for the index, the next one determines whether to shift left or right on the memory, and the final one determines the next state. Then it takes the values it read, changes the current index in the tape with the new value, shifts the pointer to either right or left depending on the input and it changes the current state pointer to the specified next state. After that, the new current

state is added to the route. When the for loop is done, tape_hist2 is reinitialized with the values of tape_hist1 while tape_hist1 is reinitialized with the values of tape. If the currentIndex pointer is lower than 0 then it means there is probably a loop so the for loop itself is broken, after which the while loop is also broken.

e) Printing

After everything is done, the program takes the route ArrayList and a StringBuilder Object, then appends the contents of the route ArrayList to the StringBuilder Object to print it later. Then it prints the route and checks if the last state it was in, after the check it prints whether it is accepted, rejected or it is a loop. The program outputs the results to the console like the following:

```
ROUTE: q1 q2 q3 q4 qR
RESULT: Rejected
```

5. Implementation

a) File Reading

The program begins by defining the empty Scanner, ArrayList, and HashMap Objects. There are also a global Strings called last, current and an integer called currentIndex. which holds the last keyword accessed, the pointer to the current state and the pointer current index. After that's done, a File Object is defined with the path "Input_MURAT_CAN_ALTUN_S021707.txt" since it is assumed that the file name and format is always "Input_MURAT_CAN_ALTUN_S021707.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 "Input_MURAT_CAN_ALTUN_S021707.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> inputAlphabet = new ArrayList<>();
ArrayList<String> tapeAlphabet = new ArrayList<>();
ArrayList<String> blank = new ArrayList<>();
ArrayList<String> states = new ArrayList<>();
ArrayList<String> startAcceptReject = new ArrayList<>();
ArrayList<String> transitions = new ArrayList<>();
ArrayList<String> detect = new ArrayList<>();
ArrayList<String> route = new ArrayList<>();
ArrayList<Character> tape = new ArrayList<>();

File path = new File( pathname: "Input_MURAT_CAN_ALTUN_S021707.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 all the ArrayLists by reading "Input_MURAT_CAN_ALTUN_S021707.txt". The function calls are the following:

```
fill(fileReader, start: "INPUT", stop: "TAPE", inputAlphabet);
fill(fileReader, start: "TAPE", stop: "BLANK", tapeAlphabet);
fill(fileReader, start: "BLANK", stop: "STATES", blank);
fill(fileReader, start: "STATES", stop: "START", states);
fill(fileReader, start: "START", stop: "ACCEPT", startAcceptReject);
fill(fileReader, start: "ACCEPT", stop: "REJECT", startAcceptReject);
fill(fileReader, start: "REJECT", stop: "TRANSITIONS", startAcceptReject);
fill(fileReader, start: "TRANSITIONS", stop: "DETECT", transitions);
fill(fileReader, start: "DETECT", stop: "END", detect);
```

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) Tape Read/Write Preparation

After all ArrayLists are filled, the program performs the following operations to prepare for the tape/read write to complete the steps explained in section 4.c:

```
current = startAcceptReject.get(0);

for (Character c : detect.get(0).toCharArray()) {
    tape.add(c);
}
tape.add(blank.get(0).toCharArray()[0]);

route.add(current);

Scanner stringScanner;

ArrayList<Character> tape_hist1 = new ArrayList<>();
ArrayList<Character> tape_hist2 = new ArrayList<>();
int count = 0;
```

d) Tape Read/Write and Transitions

After the two historical ArrayLists are initialized, the while loop starts working until all three tape ArrayLists are equal or the count variable is larger than the square of the transitions ArrayList's size. The inner for loop iterates over each transition function to check if their states match the current state the program is on, if so then it checks if the current indexes are the same in the function and the memory, if they are also the same then the old symbol is replaced with the new one and the index either shifts left or right. If the program shifts to an index lower than 0 then it breaks both loops. If not, the program goes the the next state and the new current state is added to the route ArrayList. At the end of each iteration of the while loop the program reinitializes tape_hist2 with the values of tape_hist1 and tape_hist1 with the values of tape. (tape_hist1 is always the last iteration, while tape_hist2 is always the one before tape_hist1.) Below is the code for the process described:

```
while (!tape_hist1.equals(tape) && !tape_hist2.equals(tape) || count < transitions.size() * transitions.size()) {
    count++;

    for (String s : transitions) {
        stringScanner = new Scanner(s);

        String state = stringScanner.next();

        if (state.equals(current)) {
            String change = stringScanner.next();

            if(tape.get(currentIndex).equals(change.toCharArray()[0])) {
                String newValue = stringScanner.next();
                String shift = stringScanner.next();
                String nextState = stringScanner.next();

                tape.set(currentIndex, newValue.toCharArray()[0]);

                if (shift.equals("R")) {
                    currentIndex++;
                }
                else if (shift.equals("L")) {
                    currentIndex--;
                }

                if (currentIndex < 0) {
                    break;
                }

                current = nextState;
                route.add(current);
            }
        }
    }
```

```
    if (currentIndex < 0) {
        break;
    }

    tape_hist2 = new ArrayList<>(tape_hist1);
    tape_hist1 = new ArrayList<>(tape);
}
```

e) Printing

When the while loop is done working, the program first initializes a StringBuilder called travelled, and in a for loop it traverses the route ArrayList to append its values to the StringBuilder. After that is done, it prints the String inside the StringBuilder. After that the program checks if the global static String it used to hold the current state. If it is equal to the accept state, then it outputs “RESULT: Accepted”, if it is equal to the reject state then it outputs “RESULT: Rejected”, if neither, then there must be a loop so it outputs “RESULT: Loop” to the console. Below is the code for the process described:

```
StringBuilder travelled = new StringBuilder("ROUTE: ");

for (String s : route) {
    travelled.append(s).append(" ");
}

System.out.println(travelled);

if (current.equals(startAcceptReject.get(1))) {
    System.out.println("RESULT: Accepted");
}
else if (current.equals(startAcceptReject.get(2))) {
    System.out.println("RESULT: Rejected");
}
else  {
    System.out.println("RESULT: Loop");
}
```

6. Results

The results the program outputs after it finishes running the sample input from section 3 are the following:

```
ROUTE: q1 q2 q3 q4 qR
RESULT: Rejected
```