Revision History

Date	Version	Description	Author
13.11.2023	1.0	Design of the program.	Elif Özmen

TABLE OF CONTENTS

1 Introduction

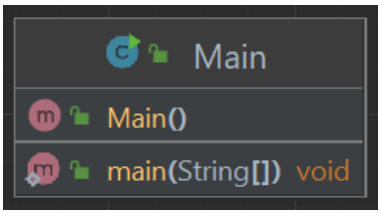
In this report I mainly tried to explain how I implement the assignment given in my CS410 class. Report mainly focuses on the introduction to the problem, design considerations, algorithms and data structures used in implementation, input/output format, error handling, testing, code structure and challenges faced during implementation.
The problem in this project was simulating a Push-Down Automaton (PDA) which works with any given alphabet, number of states, number of variables and transitions. The simulation has to work correctly as a PDA would do and it should read the information from the given input file then print the output information about the given string, states visited, string being accepted or rejected.

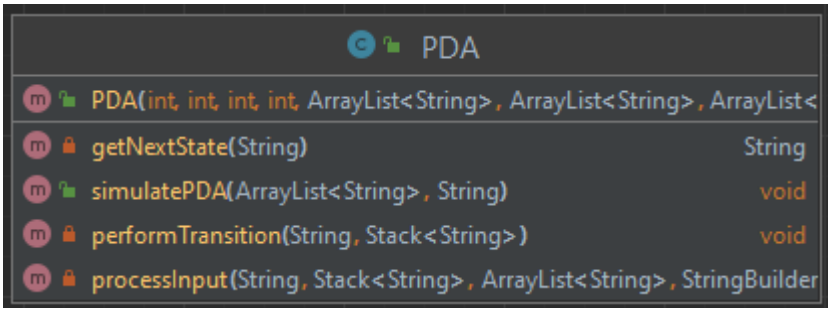
1.1 References

1.1.1 Project References

#	Document Identifier	Document Title
[SRS]	PDA-SRS-2	PDA Software Requirements Specifications

2 Software Architecture overview

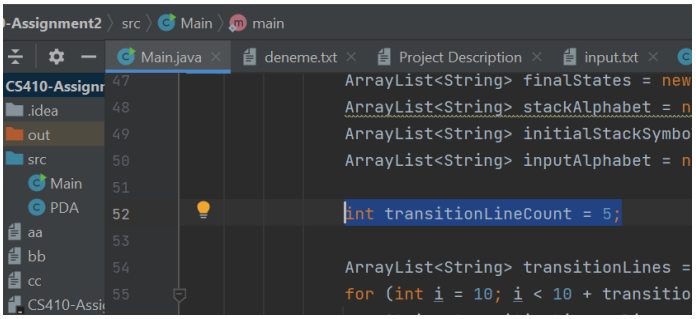




3 Software design description

3.1.1 Component interfaces

Input Data: Input is taken from the text file constructed with the same way with the example input file given. Input file's name is taken from the user in the console.
It is important to not to forget to change the "transitionLineCount" in the main according to how many lines you have in the input file.



Output Data: In the program the results are written in a output file and also to the console. Output file's name is taken from the user in the console.

3.1.2 Designing Description

Before starting to implement, I considered how to read the information from the input file. I stored each line in an ArrayList. In the given example input file it shows that the first line is the number of variables in the input alphabet, second line is the number of variables in the stack alphabet,, third line is the number of final states in PDA and the fourth line is the number of states. For the first four lines, since they are the count of the elements, I decided to parse them to integer and make each one integer parameters. For the fifth line of the input file, it is the information about the states names. I decided to keep them in a ArrayList of strings, each state name is one element in the list. As it is similar in the sixth, seventh, eighth, ninth and the tenth lines which are the names of final states and the symbols, I applied the same process to them. It is always precise, the number of the first 10 lines in the input file and what information they contain. However, it starts to move to transition lines on line 11, and this can change to any number depending on the number of states and variables. The transition lines do not include every element of the transition table's every row and column. But it may change according to the input file and I could not find a way to determine it with a generic code. So I assigned it to 5 as it was in the example input file. If there is an input file with more or less lines it needs to be changed before running. After the transition lines end, in the input file, there starts the string lines that will be taken to check if it is accepted or not in the PDA. So it starts with the (11 + transitionLineCount) line and continues until the input file ends. Again each string is an element of the ArrayList.

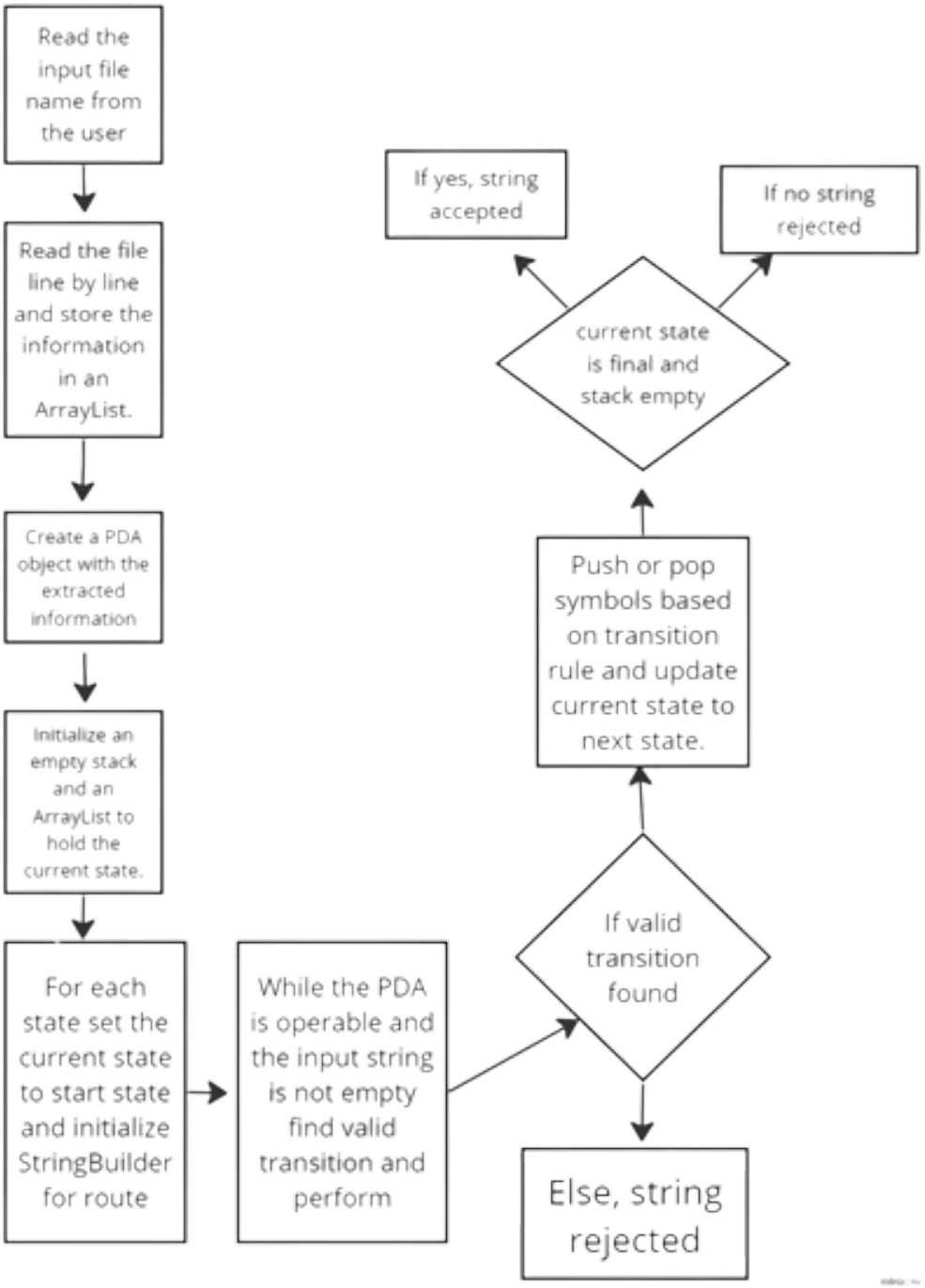
```
if (!lines.isEmpty()) {
    String firstLine = lines.get(0).trim();
    String secondLine = lines.get(1).trim();
    String thirdLine = lines.get(2).trim();
    String fourthLine = lines.get(3).trim();
    String fifthLine = lines.get(4);
    String sixthLine = lines.get(5);
    String seventhLine = lines.get(6);
    String eighthLine = lines.get(7);
    String ninethLine = lines.get(8);
    String tenthLine = lines.get(9);

    try {
        int numberOfInputAlphabet = Integer.parseInt(firstLine);
        int numberOfStackAlphabet = Integer.parseInt(secondLine);
        int numberOfGoalStates = Integer.parseInt(thirdLine);
        int numberOfStates = Integer.parseInt(fourthLine);

        ArrayList<String> statesList = new ArrayList<>(Arrays.asList(fifthLine.split( regex: " ")));
        ArrayList<String> startState = new ArrayList<>(Arrays.asList(sixthLine.split( regex: " ")));
        ArrayList<String> finalStates = new ArrayList<>(Arrays.asList(seventhLine.split( regex: " ")));
        ArrayList<String> stackAlphabet = new ArrayList<>(Arrays.asList(eighthLine.split( regex: " ")));
        ArrayList<String> initialStackSymbol = new ArrayList<>(Arrays.asList(ninethLine.split( regex: " ")));
        ArrayList<String> inputAlphabet = new ArrayList<>(Arrays.asList(tenthLine.split( regex: " ")));

        int transitionLineCount = 5;
```

After reading the input file and retrieving the information, I had to construct the PDA. So I created a class called PDA and I created a constructor for this class with the parameters I took from the input file. In PDA class there are four methods. “simulatePDA” method is for analyzing each input string and deciding if the PDA will accept it or reject it. simulatePDA method takes the inputLines, creates the stack and a StringBuilder to keep the route. Here, processInput method is called with these to make the PDA work. Also the information of whether the strings are accepted or not is printed. In the processInput method, it applies the transition algorithms while the operable variable is true, which I set for the stack operations if it is able to do them or not. It parts the transition lines as a String and for each input symbol it checks the transition lines which fits to make a transition. It takes account of three things, one where the currentState matches and the inputSymbol is epsilon which is “e”, other one is where the currentState and inputSymbol matches, and one is again with the epsilon transition but it is when all the transitions are done and to go to the final state. In each it calls the perfomTransition method for the stack operations. Again in this method it checks 3 cases to make the pop operation from the stack. processInput also calls the getNextState method to parse the next state from the transition lines. The processInput keeps doing the same things until the operable variable turns to the false, which means there happened a transition that is impossible so that the string will be rejected. After I implemented these I called the PDA constructor in the main method. I decided to print the results to both the console and to an output file. I decided to use Scanner because I wanted to take the input file’s name from the user to make it more flexible. Also I decided to take the output file’s name from the user, again with the same reason. So, this was my decision and implementation phase.

3.1.3 Workflows and algorithms

Key data structures used by the application include arrays for storing states, transitions, and other PDA components, and stacks for managing symbols. The method applies transition rules iteratively to input strings, adjusting the stack as necessary. The "processInput" method manages the stack, updates states, performs stack operations, and decides whether to accept or deny a request based on predefined rules. The "performTransition" method handles stack operations. The process illustrates how the fundamental logic of a PDA simulation is implemented using data structures like stacks and arrays.
The constructor of the PDA class initializes these data structures with the parameters retrieved from the input file. With this, it makes the code work with Object Oriented Programming principles.


Actor
Main
PDA
1:main
1.1:<<create>>
1.2:simulatePDA
1.2.1:processInput
1.2.1.1:performTransition
1.2.1.2:getNextState
1.2.1.3:performTransition
1.2.1.4:getNextState
1.2.1.5:performTransition
1.2.1.6:getNextState



4 COTS Identification

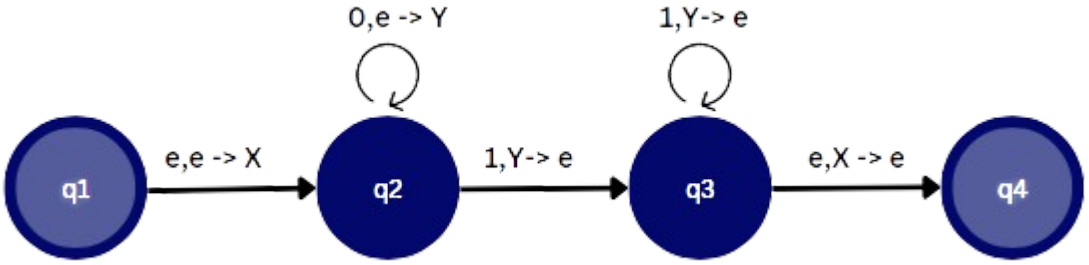
COTS (commercial of the shelf) libraries used in DFA are the following:

- java.io.IOException;
- java.util.ArrayList;
- java.util.Arrays;
- java.util.Scanner;

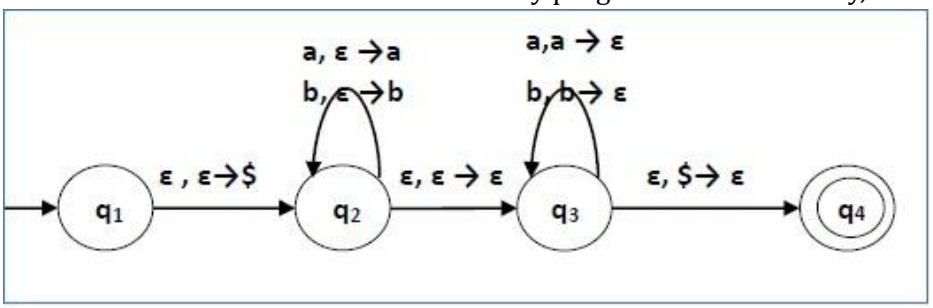
- java.io.BufferedReader;
- java.io.FileReader;
- java.io.BufferedWriter;
- java.io.FileWriter;
- java.util.Stack;

5 Testing and Error Handling

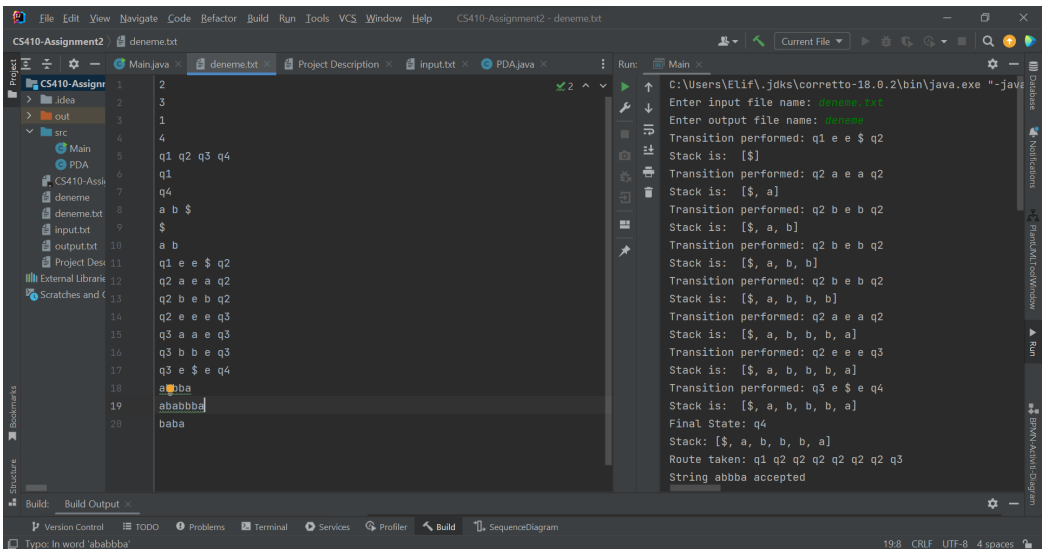
To test whether my program was working correctly, I used the PDA that given in the example input file.
Here is the PDA constructed:



For the given example PDA, it was working correct for every string.
After that I tried another PDA to check if my program works correctly, which was:

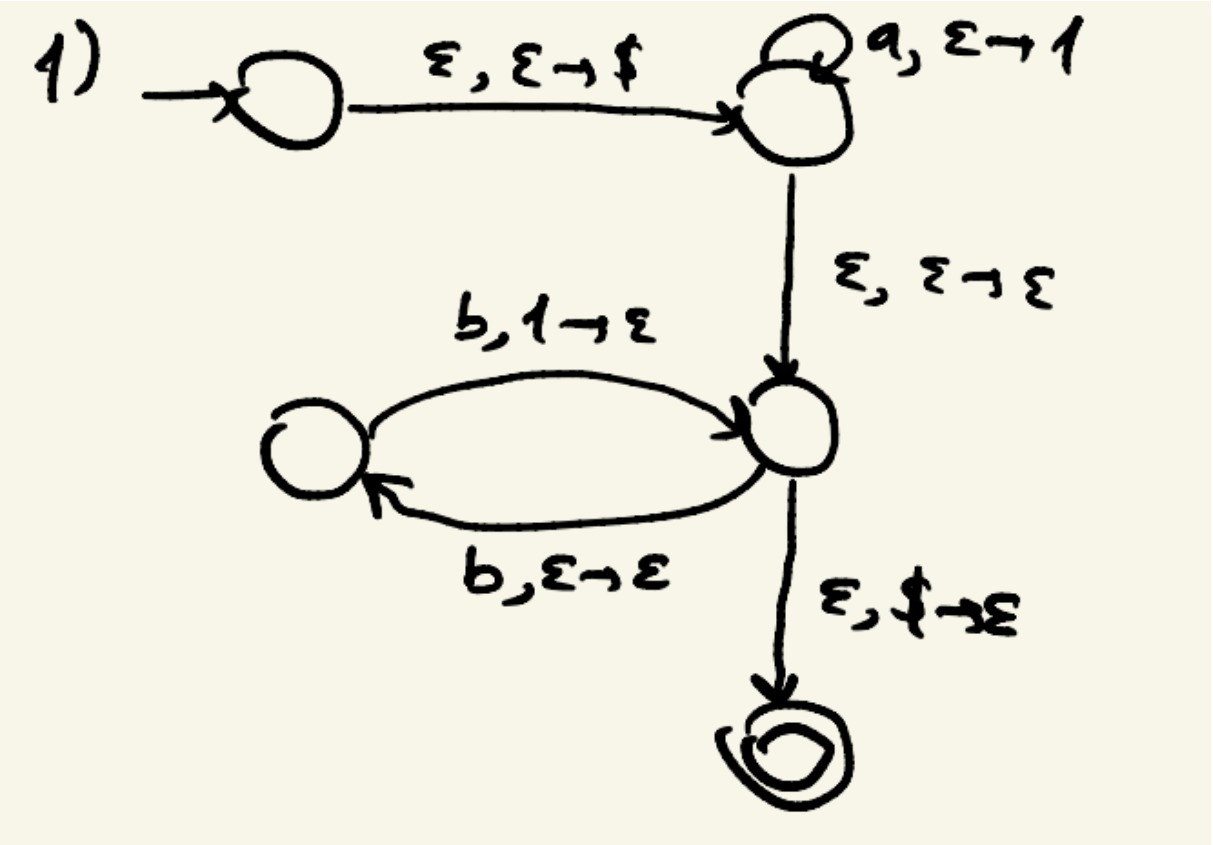


PDA for L= {ww^R | w = (a+b)*}



So I wrote the text file for that PDA and ran, it showed the correct route and transitions but it was accepting some strings that should not be accepted so I tried to understand what was

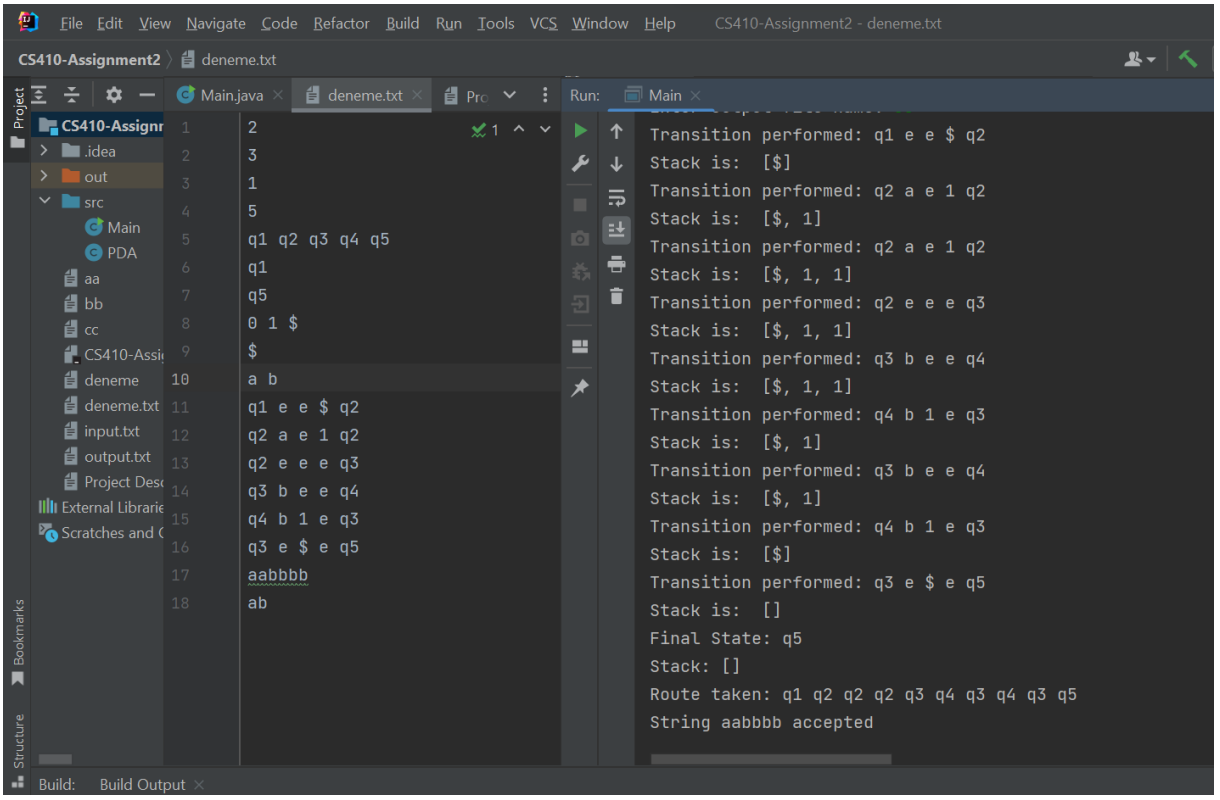
wrong with my algorithm and tried to fix. But I realized that it is a PDA that is not good for trying because the program can't understand when to move other state since there is nothing that limits that to not to stay in the q2 forever than move to the final state. So I found a better PDA example that we saw in class:

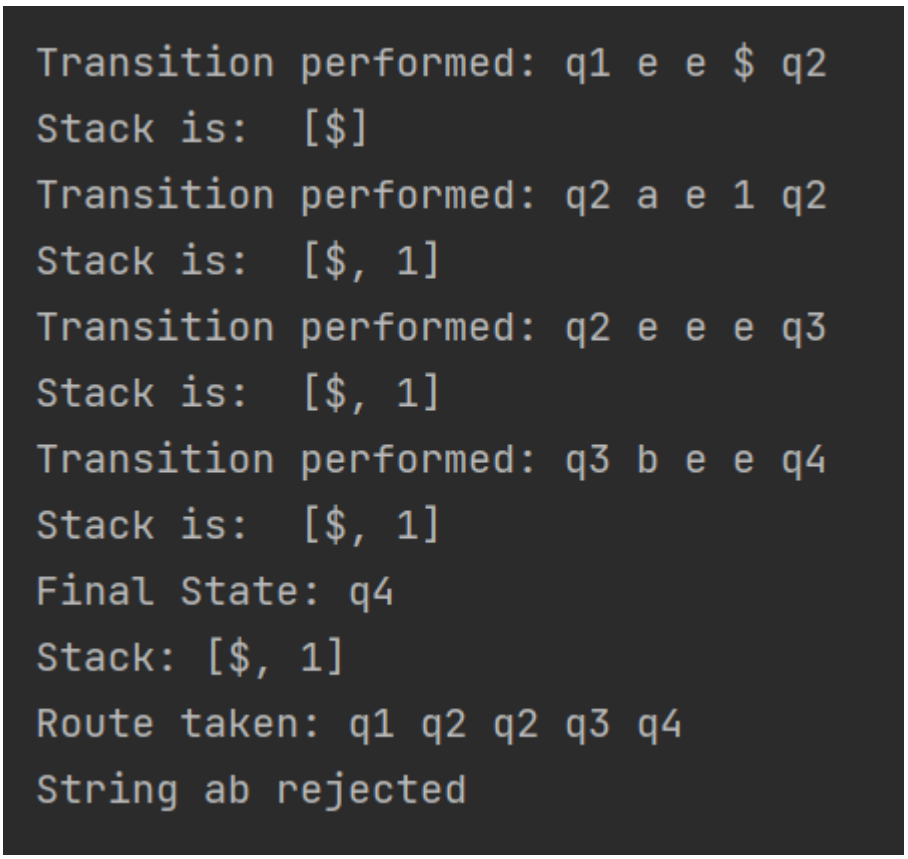


The language for the PDA is:

$$L = \{ a^n b^{2n} \mid n \geq 0 \}$$

Again, I wrote a text file for that and it worked completely fine:


File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
CS410-Assignment2 - deneme.txt
CS410-Assignment2
deneme.txt
Main.java
deneme.txt
Run:
Main
2
3
1
5
q1 q2 q3 q4 q5
q1
q5
0 1 $
$
a b
q1 e e $ q2
q2 a e 1 q2
q2 e e e q3
q3 b e e q4
q4 b 1 e q3
q3 e $ e q5
aabbbb
ab
Transition performed: q1 e e $ q2
Stack is: [$]
Transition performed: q2 a e 1 q2
Stack is: [$, 1]
Transition performed: q2 a e 1 q2
Stack is: [$, 1, 1]
Transition performed: q2 e e e q3
Stack is: [$, 1, 1]
Transition performed: q3 b e e q4
Stack is: [$, 1, 1]
Transition performed: q4 b 1 e q3
Stack is: [$, 1]
Transition performed: q3 b e e q4
Stack is: [$, 1]
Transition performed: q4 b 1 e q3
Stack is: [$]
Transition performed: q3 e $ e q5
Stack is: []
Final State: q5
Stack: []
Route taken: q1 q2 q2 q2 q3 q4 q3 q4 q3 q5
String aabbbb accepted
Build: Build Output


Transition performed: q1 e e $ q2
Stack is: [$]
Transition performed: q2 a e 1 q2
Stack is: [$, 1]
Transition performed: q2 e e e q3
Stack is: [$, 1]
Transition performed: q3 b e e q4
Stack is: [$, 1]
Final State: q4
Stack: [$, 1]
Route taken: q1 q2 q2 q3 q4
String ab rejected

The most important thing about my code is not to forget to change the "transitionLineCount" according to how many transition line you wrote on the input file.

For the error handling I used the IDE's tools. I have three exceptions: IOException, FileNotFoundException and NumberFormatException.

6 Challenges Faced

In this assignment, it was hard to determine the logic and implement it especially because of the epsilon transitions. There were so many problems about the logic and while trying to solve them I used so many debug prints, it helped me to check if the if statements are reachable or not.