Hacettepe University
Computer Science and Engineering Department

Name and Surname	: Mert Ali Yalçın
Identity Number	: 21946682
Course	: BBM203
Experiment	: Assignment 3
Subject	: Deterministic Push Down Automata
Data Due	: 17.12.2021
Advisors	: Ahmet Alkılınç
e-mail	: b21946682@cs.hacettepe.edu.tr
	mertaliyalcin07@gmail.com
Main Program	: main.cpp

2.Software Using Documentation

2.1Software Usage

This program takes dpda.txt and input.txt files as inputs and returns output.txt.

- ➔ An example of dpda.txt:

```
Q:q0,q1,q2,q3,q4 => (q0),[q0],[q1]
A:{,(,},)
Z:{,(,$
T:q0,e,e,q1,$
T:q1,(,e,q2,(
T:q1,{,e,q2,{
T:q2,{,(,q3,(
T:q2,{,{,q3,{
T:q3,e,e,q2,{
T:q2,(,{,q4,{
T:q2,(,(,q4,(
T:q4,e,e,q2,(
T:q2,},{,q2,e
T:q2,),(,q2,e
T:q2,e,$,q1,$
```

- ➔ An example of input.txt:

```
{,(,),}
(,(
```

➔ An example of output.txt:

```
q0,e,e => q1,$ [STACK]:$
q1,{,e => q2,{ [STACK]:$,{
q2,(,{ => q4,{ [STACK]:$,{
q4,e,e => q2,( [STACK]:$,{,(
q2,),( => q2,e [STACK]:$,{
q2,},{ => q2,e [STACK]:$
q2,e,$ => q1,$ [STACK]:$
ACCEPT

q0,e,e => q1,$ [STACK]:$
q1,(,e => q2,( [STACK]:$,(
q2,(,( => q4,( [STACK]:$,(
q4,e,e => q2,( [STACK]:$,(,(
REJECT
```

The program takes the inputs from input.txt, process the inputs according to the states and their relative operations in dpda.txt and returns the steps of each of the process and the final state of the machine. If the stack is empty and the machine is in an end state, then the program writes “ACCEPT” and continues with the next line of input with an empty stack and initial state. Else, the program prints “REJECT” then continues the process.

2.3 Error Messages

There is only one message. In case of there is an invalid element in the dpda.txt file, the program prints: “Error [1]:DPDA description is invalid!” and stops the execution.

3.Software Design Notes

3.1.Description of the program

3.1.1. Problem

We have lines of input and we should process them according to the dpda.txt file in our program. Execution must start in an initial state and then we must choose the correct transition rule and perform our push, pop operations to the stack. When the input line is over, the process must still continue if there is an empty transition left. When the operations are over, we should examine the final condition of the stack and the machine and print out “ACCEPT” or “REJECT” accordingly.

3.1.2. Solution

I have created a class called “Machine” and a stack as its field. In C++, some functions do not exist by default. So, I have created them in my “Machine” class and thus I have avoided repetition of the code as much as possible. First part of my solution was taking the necessary input from the 2 “.txt” files and initialize them in my “Machine” class. First, I have thought about doing this process with an empty constructor but in the end, I have decided to do it in my constructor, so that the main code can be easier to read. After doing all of initializations, I have started to look for transition rules with initial states. The transition rule with initial state and with a non-empty “read” comes first in precedence against a transition rules with an empty “read”. So, I have decided my first transition rule accordingly. There is a rare situation that input line is empty and there is no empty “read” statements. I have checked this situation as first thing to do in my code and the rest is the same as choosing my initial state. The program continued in this manner for each input line and ended accordingly.

3.4 Algorithm

1.Read the input data from the files and put them in a vector.

2. Create a Machine called “m”. Use the two vectors to initialize the machine.

3.Check for DPDA file.

    3.1. If the file contains some invalid elements, print the error message and exit from the program

4.Put every line of input in a vector.

5.If the vector is not empty, start execution.

6.If the input line is empty and there is no empty “read” transition rule,

    6.1. write the final state of the file then move to the next input line.

7.Choose the first transition rule.

    7.1.If there is something to push, push to the stack

8.Switch between the transition rules.

    8.1.If there is a transition rule with “read” field as same as the next input in the input line, choose that rule.

    8.2.If there are two of this, choose the one with the correct “pop” operation.

    8.3.If there is no such transition rule, choose the one with empty “read” field.

9.Perform the “pop” and “push” operations relatively.

    9.1.If the top of the stack is not same as the “pop” field, print “REJECT” and continue with the next line of inputs.

10.If the inputs and empty “read” transition rules are ended, write the state of the machine and continue with the next line of inputs.

3.5 Special Design Properties

I don’t have new approaches to the problem, but if I had any, these would be adding more functions and properties to the program to increase its modularity. According to me, I find this design necessary and sufficient for the program that I have.

4.Software Testing Notes

4.1. Bugs and Software Reliability

I had many bugs to solve when my program is finished but in the end, I have solved them all. The program works correctly in the school’s DEV server with the samples provided to us.

Only thing is that when I drag and drop the txt files to the DEV server, the txt file has some problems with extra newlines at the end of each line and this causes some errors within the program. This has something to do with Windows Operating System I guess.

This bug can be avoided very easily by copying and pasting the contents of the txt files to an empty txt file created in the DEV server.

Other than that, I have not encountered with anymore bugs in my testing in the DEV server. The program should work fine.

REFERENCES

BBM201 Lecture Notes

BBM203 Lecture Notes