ASSIGNMENT 3

Subject: Stack

Programming Language: C++
Submission Date: 03.12.2021
Due Date: 17.12.2021

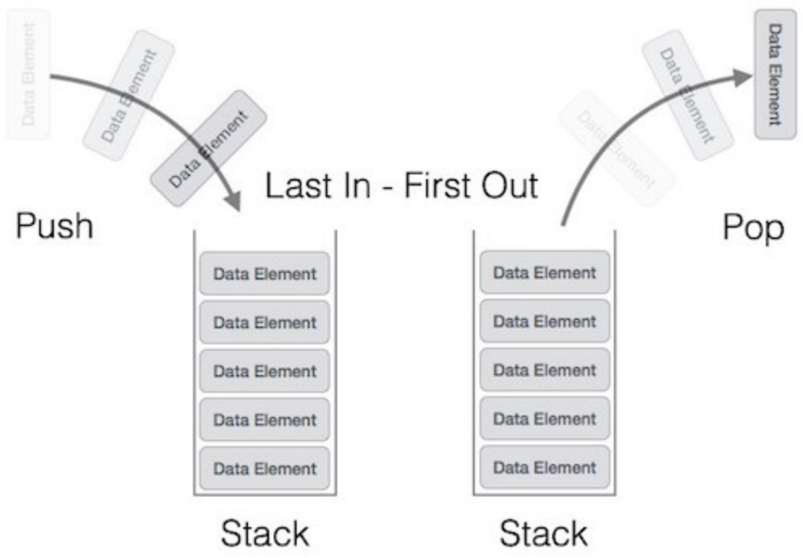
1. INTRODUCTION

The main purpose of the experiment is to develop your skill of using Stack Data Structures with C++ programming language. Other purposes of the experiment are to learn Deterministic Pushdown Automata (DPDA) and also to perform file operations by using input/output libraries of C++.

2. BACKGROUND INFORMATION

2.1 Stack

Stack is one of the fundamental data structures in computer science and an ordered list in which all insertions and deletions are made at one end, called the top or a Last-In-First-Out (LIFO) data structure. In a LIFO data structure, the last element added to the structure must be the first one to be removed.
.



.

Figure 1: Stack and its operation

2.2 Deterministic Pushdown Automata (DPDA)

In computer science, a pushdown automata (PDA) is a type of automata that employs a stack. The PDA is used in theories about what can be computed by machines. The term "pushdown" refers to the fact that the stack can be regarded as being "pushed down" since the operations never work on elements other than the top element.

In automata theory, a deterministic pushdown automata (DPDA) is a variation of the pushdown automata. The DPDA accepts the deterministic context-free languages, a proper subset of context-free languages. In other words, a language can be recognized by PDA. Machine transitions in PDAs are based on the current state and input symbol, and also the current topmost symbol of the stack. A deterministic pushdown automata has at most one legal transition for the same combination of input symbol, state, and top stack symbol. Pushdown automata choose a transition by indexing a table by an input signal, current state, and the symbol at the top of the stack. This means that those three parameters completely determine the transition path that is chosen.

A Pushdown Automata P = (Q, Σ, Γ, δ,q_0, Z_0, F) is given by the following data

- A finite set Q of states,
- A finite set Σ of symbols (the alphabet),
- A finite set Γ of stack symbols,
- A transition function (δ)
- An initial state q_0 ∈ Q,
- An initial stack symbol Z_0 ∈ Γ,
- A set of final states F ∈ Q

There are two possible acceptance criteria: acceptance by empty stack and acceptance by final state. You will use the final state acceptance criteria in the experiment.

In one transition the DPDA may do the following:

- ➔ Consume the input symbol. If e (empty string) is the input symbol, then no input is consumed.
- ➔ Pop the expected symbol at the top of the stack.
- ➔ Go to a new state, which may be the same as the previous state.
- ➔ Push the symbol. If e (empty string) is the symbol, then no symbol is pushed.

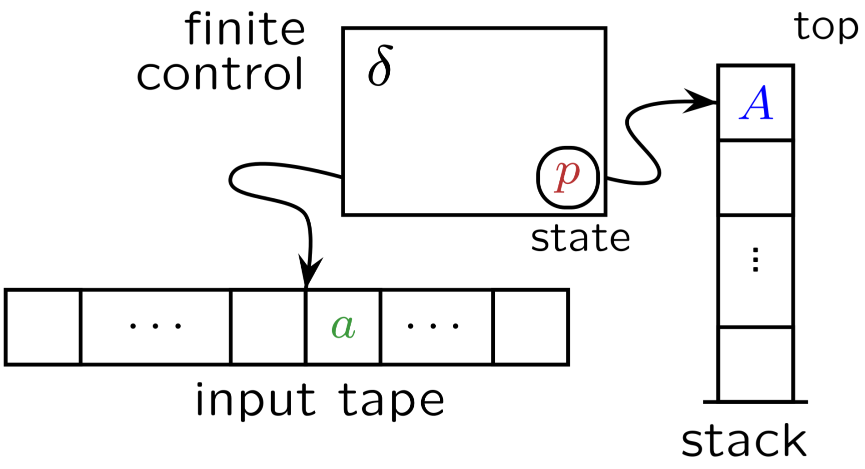


Figure 3: A diagram of a pushdown automata

2.3 File Operations in C++

C++ provides a large library for different file operations. In this experiment, you will use the basic I/O of C++ to perform file operations.

3. EXPERIMENT

You will write a program using DPDA. The program accepts a string belonging to a language or rejects a string not belonging to the language. The program is to read a description of a DPDA and one input for it. It will process an input and output the execution path. After processing one string, the program should reset the automata (this includes emptying the stack if necessary) and proceed with the next string.

3.1 DPDA Section

A DPDA description will be provided in a file. The file will consist of text lines defining all 4
parts of a DPDA. The format is as follows:

➔ A line starting with “Q:" will list the set of states, where the states are separated with a comma (with no spaces). Then “ => ” string is read. “()” will specify the start state. For example, we might have (q0). “[]” will specify the final (accept) state. For example, we might have [q1]. And the list is terminated by the end-of-line character. The maximum number of states is 50. For example, a list of states can look like Q:q0,q1,q2,q3,q4 => (q0),[q0],[q1].

➔ A line starting with “A:" will list the input alphabet ∑, where the symbols are separated with a comma (with no spaces) and the list is terminated by the end-of-line character. The maximum number of symbols is 50. For example, the alphabet for strings will be given as A:{,(,},).

➔ A line starting with “Z:" will list the stack alphabet, where the symbols are separated with a comma (with no spaces) and the list is terminated by the end-of-line character. The maximum number of symbols is 50. The maximum stack size is 200.

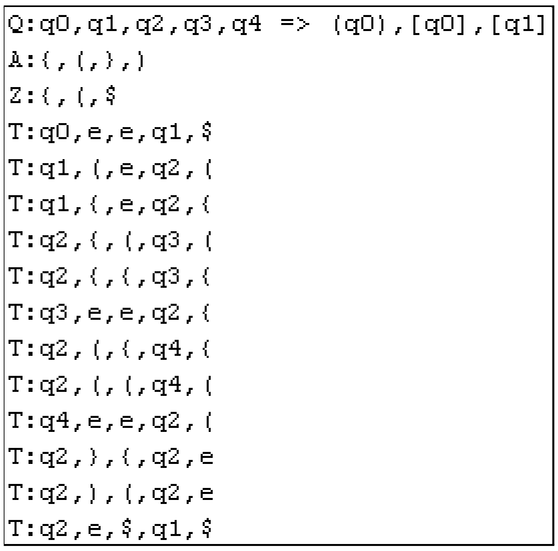
➔ A line starting with “T:" provides a single transition rule. The format of the line is the starting state followed by a comma, the input symbol followed by a comma, the expected symbol at the top of the stack (which is being popped), the resulting state, and the symbol to be written on the stack (all with no spaces). The maximum number of rules is 100. For example, we might have a line T:q0,e,e,q1,$. There may be multiple lines of this form per machine description. The program must check that both of the states given in the transition function are valid states from the Q set, the input character is a valid character from the input alphabet, and both stack symbols are from the stack alphabet. A transition rule can be shown that: (state,read,pop,next state,push).

The transition rules might include the empty string, which will be denoted in the machine description by “e”. This will be a special character that will not be permitted to appear as the name of a DPDA's state or as a symbol of the input or stack alphabets.

Since the program deals with deterministic PDA, upon reading all transition rules from the DPDA's description, it must verify that computation can proceed deterministically. That is, no configuration has a choice of more than one move, as specified in the definition of a DPDA.

Also, the transition rules specified for the current state will determine whether the DPDA is to read the next input character, pop the stack, or do neither or both. Assume that the DPDA will make as many transitions as possible if its description contains transitions on the empty string.

Stack is assumed to be empty at the beginning of each simulation. If more than one input is specified, the simulator must empty the stack after each simulation and reset the state. There will be a description of a single DPDA in a file. The file name will be given as an argument to the program. As a sample description, consider the machine:



```
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,$,q1,$
```

3.2 Input Section

Once the program is initialized with a DPDA description, it will take tape strings from the input.
The format is as follows:

- ➔ The lines will contain strings, one sequence per line. Each line will consist of a list of alphabet symbols, each separated by a comma. For example, if the alphabet is ∑ = { {,(,},) }, then one input line could be:

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

3.3 Output Section

The program should print all information to the output file. For each input string, the program's output will consist of a trace of program execution. Traces corresponding to different strings should be separated by a blank line. For each input string, for each transition (one transition per line) the program is to print:

- ➔ The initial state is followed by a comma(with no spaces);
- ➔ The input(read) character used in the transition (if none was used, e is to be printed) followed by a comma(with no spaces);
- ➔ The stack symbol popped during the transition (if none, e is to be printed) followed by a single space,”=>” string and a single space;
- ➔ The resulting state is followed by a comma(with no spaces);
- ➔ The stack symbol pushed during the transition (if none, e is to be printed) followed by a single space and “[STACK]:” string;
- ➔ And the contents of the entire stack start from the bottom, with each symbol separated by a comma (no spaces).

If the end of the string is reached and the machine is in an accept state, it should print ACCEPT on a line by itself, and should output REJECT at the end of execution otherwise. Then the machine will reset and proceed to process the next string if more strings are remaining. For example, given the DPDA above, if the input is:

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

The program's output will be:

```
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
```

It is possible that execution has to terminate prior to processing the entire input, in which case the simulator will print REJECT at the point of execution when the program is unable to proceed.

Another example is shown below (L = $\{0^n1^n | n \geq 0\}$):

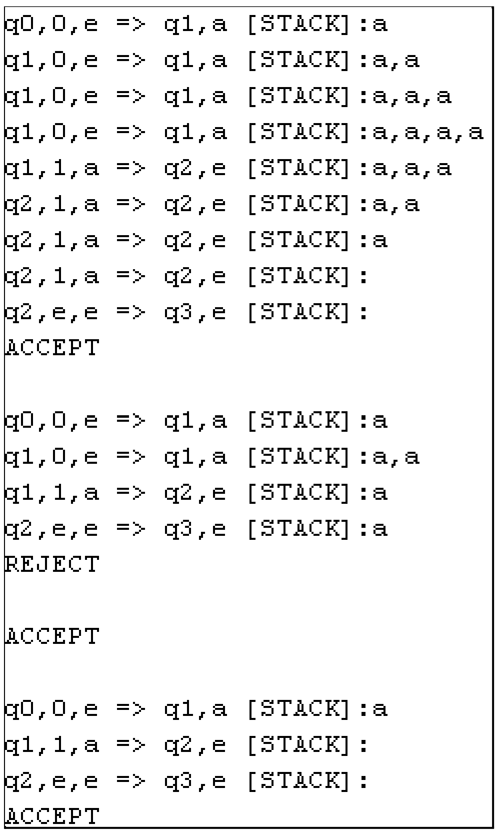
```
Q:q0,q1,q2,q3 => (q0),[q0],[q3]
A:0,1
Z:a
T:q0,0,e,q1,a
T:q1,0,e,q1,a
T:q1,1,a,q2,e
T:q2,1,a,q2,e
T:q2,e,e,q3,e
```

(a) DPDA File

```
0,0,0,0,1,1,1,1
0,0,1,0

0,1
```

(b) Input File



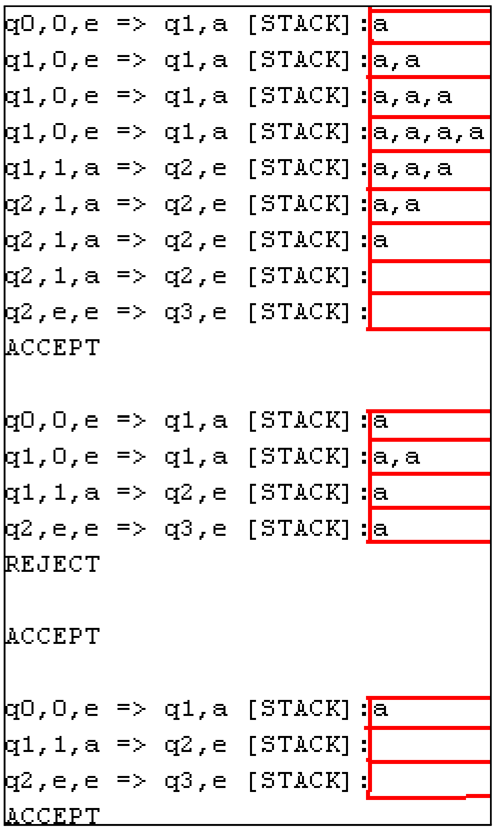
```
q0,0,e => q1,a [STACK]:a
q1,0,e => q1,a [STACK]:a,a
q1,0,e => q1,a [STACK]:a,a,a
q1,0,e => q1,a [STACK]:a,a,a,a
q1,1,a => q2,e [STACK]:a,a,a
q2,1,a => q2,e [STACK]:a,a
q2,1,a => q2,e [STACK]:a
q2,1,a => q2,e [STACK]:
q2,e,e => q3,e [STACK]:
ACCEPT

q0,0,e => q1,a [STACK]:a
q1,0,e => q1,a [STACK]:a,a
q1,1,a => q2,e [STACK]:a
q2,e,e => q3,e [STACK]:a
REJECT

ACCEPT

q0,0,e => q1,a [STACK]:a
q1,1,a => q2,e [STACK]:
q2,e,e => q3,e [STACK]:
ACCEPT
```

(c) Output File



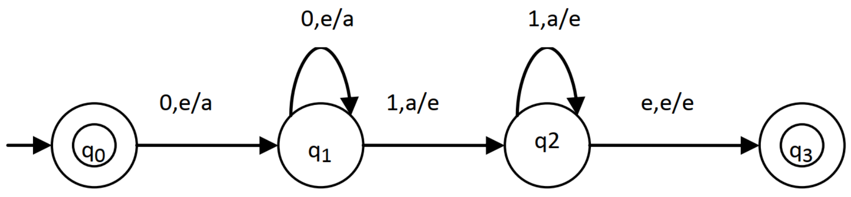
```
q0,0,e => q1,a [STACK]:a
q1,0,e => q1,a [STACK]:a,a
q1,0,e => q1,a [STACK]:a,a,a
q1,0,e => q1,a [STACK]:a,a,a,a
q1,1,a => q2,e [STACK]:a,a,a
q2,1,a => q2,e [STACK]:a,a
q2,1,a => q2,e [STACK]:a
q2,1,a => q2,e [STACK]:
q2,e,e => q3,e [STACK]:
ACCEPT

q0,0,e => q1,a [STACK]:a
q1,0,e => q1,a [STACK]:a,a
q1,1,a => q2,e [STACK]:a
q2,e,e => q3,e [STACK]:a
REJECT

ACCEPT

q0,0,e => q1,a [STACK]:a
q1,1,a => q2,e [STACK]:
q2,e,e => q3,e [STACK]:
ACCEPT
```

(d) Stack Representation of Output



(e) DPDA

3.4 Execution Of The Program

Command-Line Arguments:

Main <dpda_file> <input_file> <output_file>

- <dpda_file> - DPDA description file (ie. dpda1.txt)
- <input_file> - file which consists of strings (ie. input1.txt)
- <output_file> - output's file name

Usage Example:

> g++ -o Main *.cpp -std=c++11

> ./Main dpda1.txt input1.txt output1.txt

The program must run on DEV UNIX machines. So make sure that it compiles and runs in one of the UNIX labs. If we are unable to compile or run, the project risks getting zero points. It is recommended that you test the program using the same mechanism on the sample files (provided) and your own inputs. You must compare your own output and sample output. If your output is different from the sample, the project risks getting zero points, too.

3.5 Your Report

You must write a report which is related to your program.

Notes

- The output file of your program should be in the same format as the sample output file. Please create your output format carefully. Commas, spaces, newlines, and other symbols should be as in the sample output file.
- Do not miss the deadline.
- Save all your work until the assignment is graded.
- The assignment must be original, individual work. Duplicate or very similar assignments are both going to be considered cheating.
- Write **READABLE SOURCE CODE** block

  and you are supposed to be aware of everything discussed in Piazza.
-
- File hierarchy must be zipped before submitted (**Not .rar, only .zip files are supported by the system**). You must submit your work with the file hierarchy stated below:

```
<student_id.zip> (Required)
    →src/(Required)
        →Main.cpp (Required)
        →*.cpp(optional)
        →*.h(optional)
    →Report.pdf (Required)
```