Hacettepe University

Computer Science and Engineering Department

BBM203 Software Laboratory I - 2021 Fall

Assignment 3
STACK

Date :18.12.2021

Advisor: Ahmet ALKILINÇ

Student Name and Surname:
Melike Nur DULKADİR

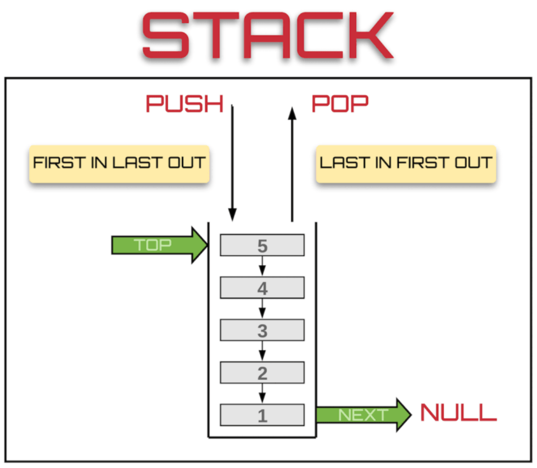
Student Number:
b21992919

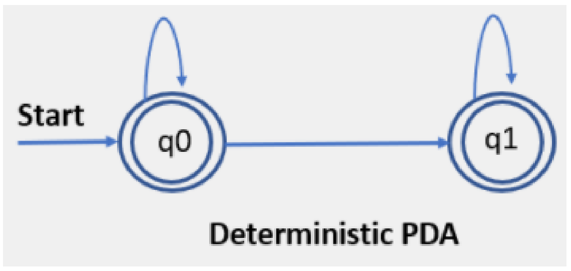
Student email:

b21992919[at]cs.hacettepe.edu.tr

1- INTRODUCTION

Our aim in this assignment was to use the stack structure, which is a data structure, with the C++ language. It was also to perform file operations using this structure in the implementation of the Deterministic Pushdown Automata (DPDA).





2- PROBLEM DEFINITION

With the program we will write using DPDA, we will check whether the strings that come as input belong to a language. If a string that does not belong, we will reject it. First of all, we will read an description to create a DPDA and then accept an input behind it. When the reading is finished after the incoming strings, the DPDA will reset itself and continue reading over the other strings.

3- DEVELOPMENT

a) Plan

First of all, I planned how I should use the stack structure and the DPDA system together. Afterwards, I planned how to process the incoming inputs and how to store them. After all this, I calculated the conditions under which the program should be terminated and created condition blocks.

b) Analysis

We needed to use stack data structure in this assignment. For this reason, I analyzed the basic functions and usage rules of the stack structure. Then, with the help of the information written in the assignment pdf, I understood the working logic of the DPDA system. After all this, I started to implement this information in code blocks and continued the analysis process in the development part.

c) Design and Method

I created QList, AList and Zlist in Main class. In addition to these, after reading the input, I created the necessary structures and lists for operations. Then, to read the DPDA file, I wrote a header file named DpdaSystem.h with the necessary utility functions for file reading operations and line splitting operations. In this header file, we perform the necessary checks at the bottom of the readCommands function. For example, we check whether the symbols in Transition rules exist in QList, AList and ZList.

Main class

```
#include <iostream>
#include <vector>
#include <string>
#include "DpdaSystem.h"
#include "Operations.h"
using namespace std;

ifstream dpdaFileObj;
ifstream inputFileObj;

int main(int argc, char** argv) {
    dpdaFileObj.open(argv[1]);             // Open dpda file for dpda system
    inputFileObj.open(argv[2]);            // Open input file
    ofstream output;
    output.open(argv[3], ios::out);        // Create output file
    vector<string> QList;                  // Creating set of states
    vector<string> AList;                  // Creating input alphabet
    vector<string> ZList;                  // Creating stack alphabet
    vector<vector<string>> rules;  // Creating rules list for transition rule
    string starting_state;
    string inputSymbol;
    vector<string> final_states;           // Creating list for final_States
    vector<string> inStack;
    readCommands(QList,AList,ZList,rules,starting_state,final_states,output);

readInput(inputSymbol,starting_state,final_states,rules,inStack,ZList,output)
;
    dpdaFileObj.close();                             // Closing dpda file
    inputFileObj.close();                            // Closing input file
    output.close();                                  // Closing output file
    return 0;
}
```

DpdaSystem.h (top of the file, example utility function)

vector<string> split (const string &s, char delim):

```
#include <vector>          // Including necessary "external" library files.
#include <sstream>
#include <string>
#include <fstream>
#include <algorithm>

using namespace std;

extern ifstream dpdaFileObj;

// Function for splitting string in dpda input file
vector<string> split (const string &s, char delim) {
    vector<string> result;
    istringstream ss (s);
    string item = " ";
    while (getline (ss, item, delim)) {
        result.push_back (item);
    }
    result[0] = result[0].substr(2,result[0].size()-2);

    return result;
}
```

Operations.h

I am using the readInput function in the Operations header in the Main class. After each input line is read, we store them in a vector structure. We try to find the appropriate transition rule and reach the resulting state by calling the transition function for each input symbol in each vector.

I created many condition blocks to find the appropriate rules. Afterwards, if the resulting state is not in the final states, although we have reached the last input symbol in the readInput function, the transition function is called once again. "REJECTED" output is showed if the resulting state is not found in final states, and "ACCEPTED" output is showed if it exists and the **stack** is empty.

The cornerstone of the implementation is the **transition** function. However, since it is a large code block, I cannot include it in the report.

Code block from readInput function to read input lines and storing it in the tokens vector:

```
// This function for reading dpda input file
void readInput(string& inputSymbol,string& starting_state,vector<string>&
final_states,vector<vector<string>>& rules,vector<string>&
inStack,vector<string>& ZList,ofstream& output) {
    string fixed_state = starting_state;
    vector<string> tokens;
    string commandLine;
    while (getline(inputFileObj, commandLine)) {
        istringstream iss(commandLine);
        if (commandLine.empty()){
            string empty = "empty";
            tokens.push_back(empty);
        }
        string token;
        while (getline(iss, token)) {
            tokens.push_back(token);
        }
    }
```

More from Operations.h …

"ACCEPTED" if line is empty and starting state is in final stateS, otherwise "REJECTED".

```
for (auto & token : tokens) {
    starting_state = fixed_state;
    if (token == "empty"){                                                  // If
the line is empty
        if (find(final_states.begin(), final_states.end(),starting_state)!=
final_states.end()){
            output<<"ACCEPT\n"<<endl;
            continue;
        }else{
            output<<"REJECT\n"<<endl;
            continue;
        }
    }
    // Creating new stack for each line
    stack<string> stack;
    inStack.clear();
    vector<string> inputList = splitInput(token,',');
    for(auto & symbol: inputList) {
        inputSymbol = symbol;
```

4- RESULTS

As a result, we have developed a system where we can keep multiple types of employees separately across multiple data structures. In this system, we created linked lists using two different implementations and performed their operations. In the last part, we tested it and saw it working. It has been seen that using dynamic allocation during the development phase is much more convenient and easy. In addition, it was very difficult to run the application on different systems due to the compiler difference. We got to know the basic linked list structure in the end.

REFERENCES

https://www.geeksforgeeks.org/stack-data-structure/

https://en.wikipedia.org/wiki/Deterministic_pushdown_automaton

https://dev.to/theoutlander/implementing-the-stack-data-structure-in-javascript-4164

https://www.etutorialspoint.com/index.php/theory-of-computation/theory-of-computation-deterministic-pushdown-automata