ASSIGNMENT 3

Serhat Gündem
2200356820
BBM203
17-12-2021

Advisor : Ahmet Alkılınç

2. Software Using Documentation

2.1 Software Usage

This software runs through 1-character strings. It accepts the string if it belongs to the language that is defined before and rejects it otherwise (it gives error otherwise).

2.2 Error Messages

There will only one error message which comes out when the stack alphabet, the input alphaber or the states do not consist any of the rules taken as input.

The message will be "Error [1]:DPDA description is invalid!\n"

3. Software Design Notes

3.1 Description of the program

3.1.1 Problem

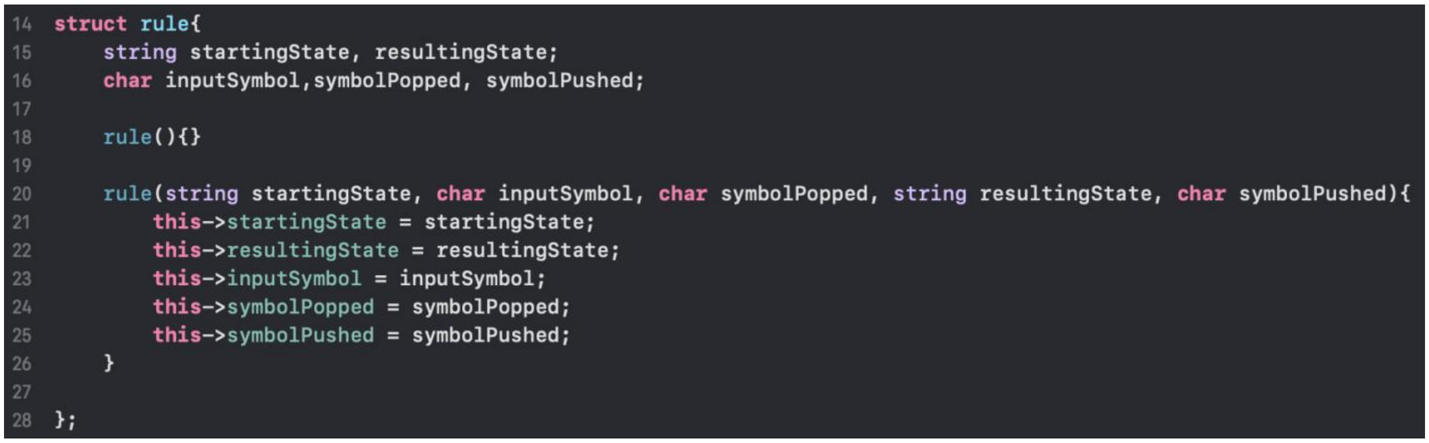
It is asked to write a program using DPDA. The program will be accepting or rejecting the string taken from the console in case of the validity of the string for the language. A string is valid for a language if the string belongs to that language.

3.1.2 Solution

I've used the DPDA rules and a stack to store the strings that is taken from the console. In my code, the stack ADT is used from the standard library so I didn't need to create a stack class to solve the problem. First, input taking part is done, the input string is divided into parts by taking care of the comma that separates the inputs.

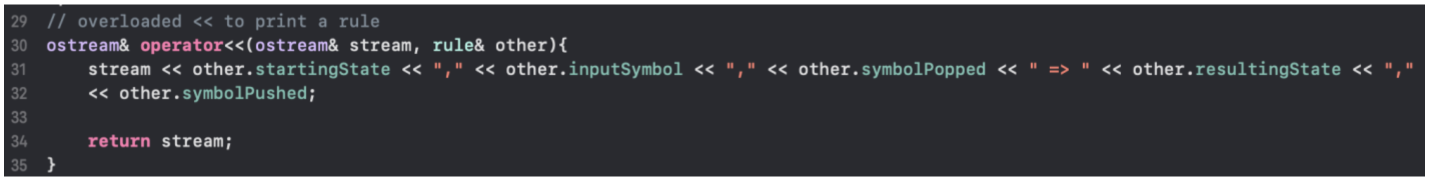
3.4 Algorithm

Below is the explanation of the solution to the problem.



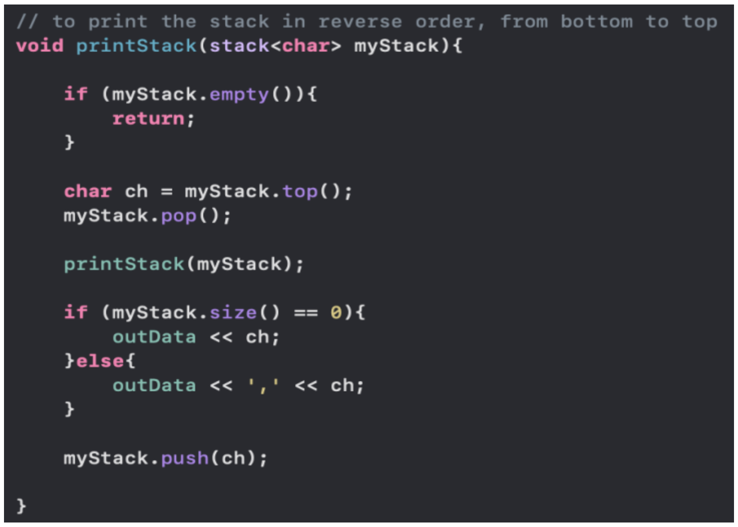
```cpp
struct rule{
    string startingState, resultingState;
    char inputSymbol,symbolPopped, symbolPushed;

    rule(){}

    rule(string startingState, char inputSymbol, char symbolPopped, string resultingState, char symbolPushed){
        this->startingState = startingState;
        this->resultingState = resultingState;
        this->inputSymbol = inputSymbol;
        this->symbolPopped = symbolPopped;
        this->symbolPushed = symbolPushed;
    }

};
```

To control and access easier, when taking the rules from the console, a rule struct is created to store all the rules instead of storing them in a vector and traversing through the vector by index. (Of course the rule objects were stored in a vector but this is different than storing the parts of every rule in a vector)



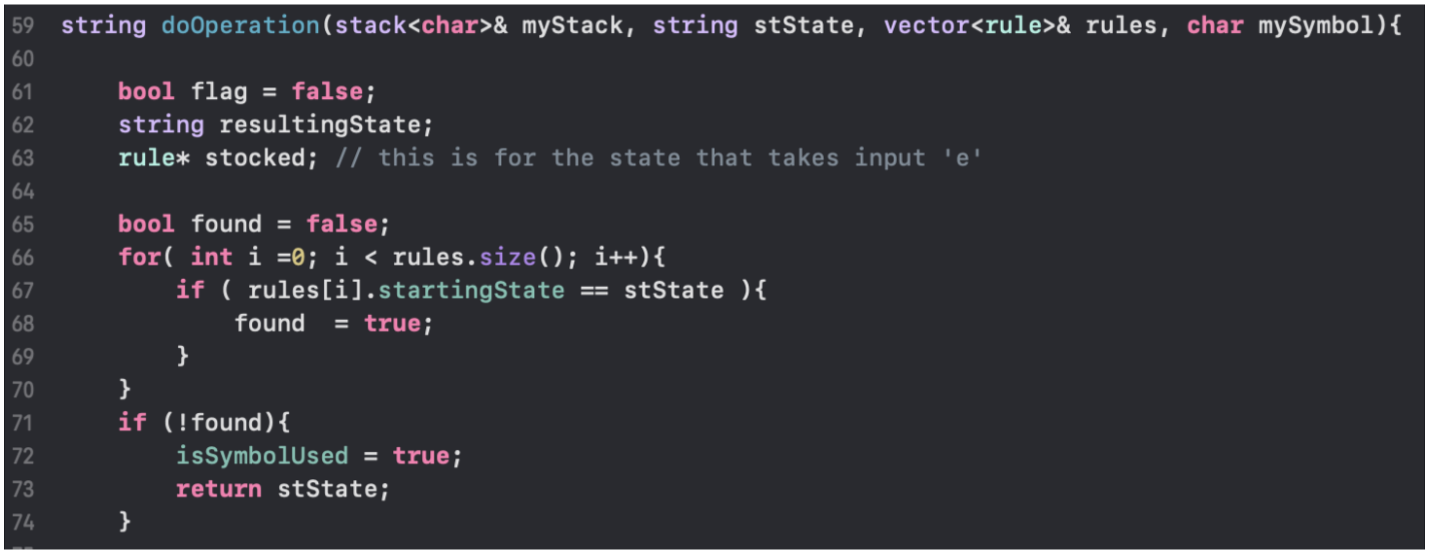
```cpp
// overloaded << to print a rule
ostream& operator<<(ostream& stream, rule& other){
    stream << other.startingState << "," << other.inputSymbol << "," << other.symbolPopped << " => " << other.resultingState << ","
    << other.symbolPushed;

    return stream;
}
```

Then, << operator is overloaded to print every rule



```cpp
// to print the stack in reverse order, from bottom to top
void printStack(stack<char> myStack){

    if (myStack.empty()){
        return;
    }

    char ch = myStack.top();
    myStack.pop();

    printStack(myStack);

    if (myStack.size() == 0){
        outData << ch;
    }else{
        outData << ',' << ch;
    }

    myStack.push(ch);

}
```

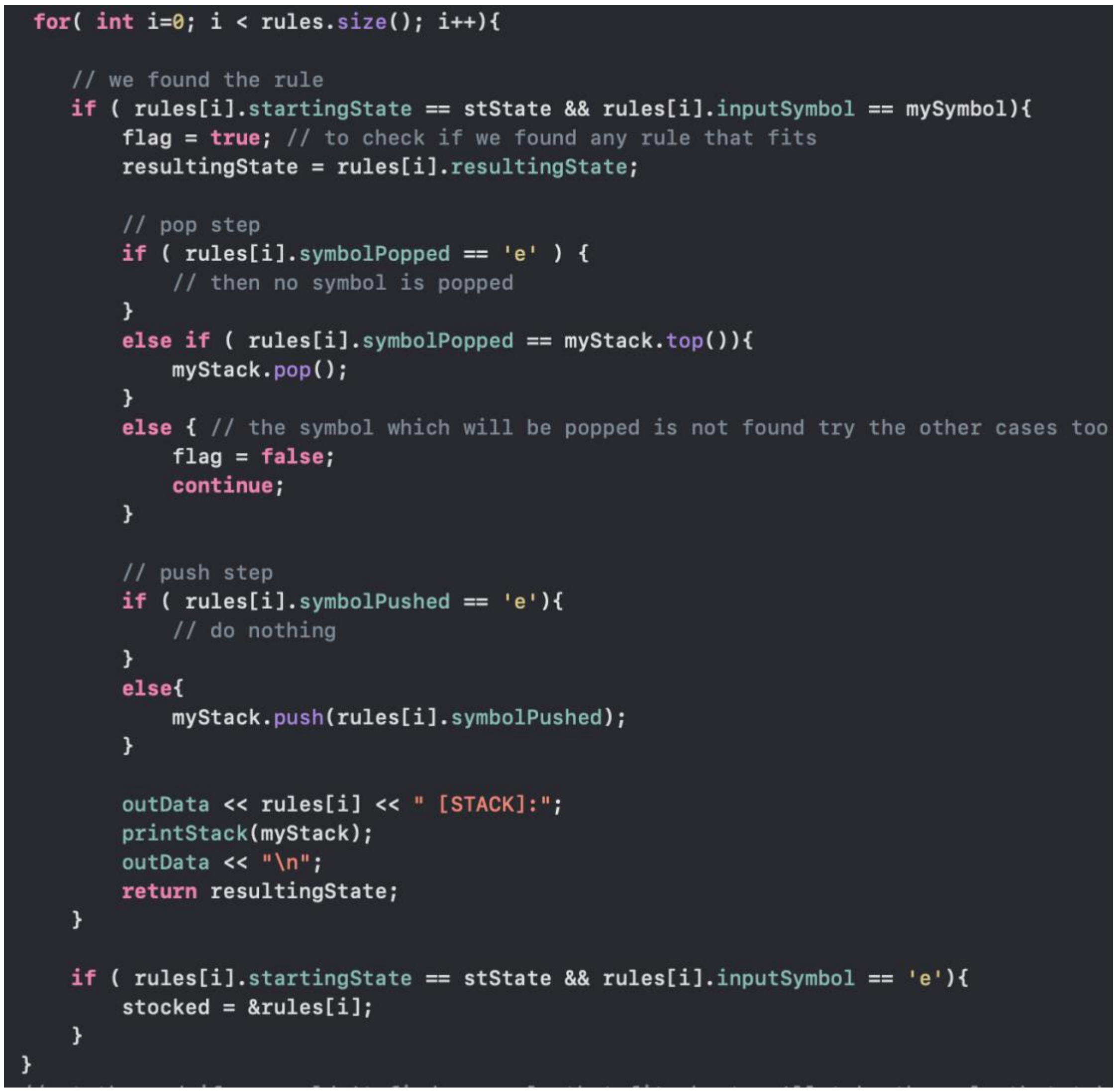
In above part, a stack that is taken as parameter is printed using recursive algorithm. In this algorithm, it is first checked if this stack is empty or not, then, the top element of the stack is stored to a value named ‘ch’ and popped the top element from the stack. After that, the same function is called again and with this way, we are going up to the last element of the stack and firstly print it. This is why, this algorithm starts printing from bottom to top.


```
string doOperation(stack<char>& myStack, string stState, vector<rule>& rules, char mySymbol){

    bool flag = false;
    string resultingState;
    rule* stocked; // this is for the state that takes input 'e'

    bool found = false;
    for( int i =0; i < rules.size(); i++){
        if ( rules[i].startingState == stState ){
            found  = true;
        }
    }
    if (!found){
        isSymbolUsed = true;
        return stState;
    }
```

This is where the main operation is done. We have a function that takes a stack which represents the stack we're already using, a string called stState which represents the state that we'll use as the initial step, a vector that consists of the rules that we already defined before, and a char named 'mySymbol' which represents the input character.

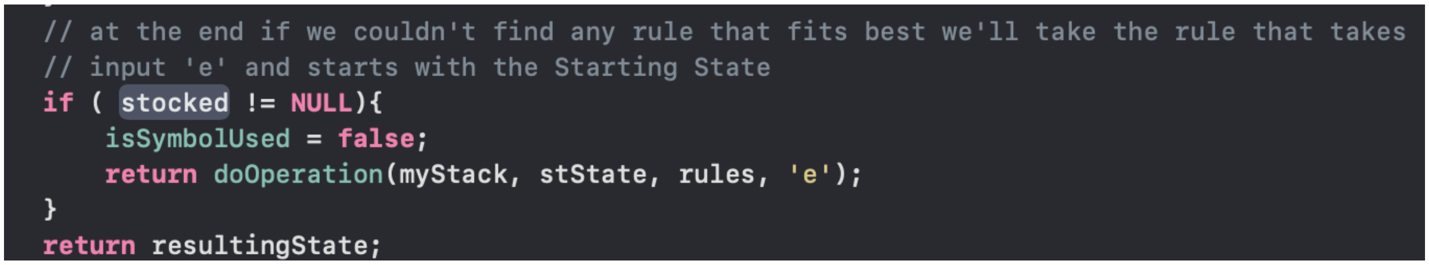
Inside the function, we have a variable 'flag' which returns true whenever the true transition rule is found, a string representing the resulting state and a rule pointer that'll be explained later.

First, it is checked if there is rule that starts with the initial state which is 'stState', if there is no any rule that starts with we're just ignoring that input and return the initial state again which will be the initial state of the next input.



```
  for( int i=0; i < rules.size(); i++){

      // we found the rule
      if ( rules[i].startingState == stState && rules[i].inputSymbol == mySymbol){
          flag = true; // to check if we found any rule that fits
          resultingState = rules[i].resultingState;

          // pop step
          if ( rules[i].symbolPopped == 'e' ) {
              // then no symbol is popped
          }
          else if ( rules[i].symbolPopped == myStack.top()){
              myStack.pop();
          }
          else { // the symbol which will be popped is not found try the other cases too
              flag = false;
              continue;
          }

          // push step
          if ( rules[i].symbolPushed == 'e'){
              // do nothing
          }
          else{
              myStack.push(rules[i].symbolPushed);
          }

          outData << rules[i] << " [STACK]:";
          printStack(myStack);
          outData << "\n";
          return resultingState;
      }

      if ( rules[i].startingState == stState && rules[i].inputSymbol == 'e'){
          stocked = &rules[i];
      }
  }
```

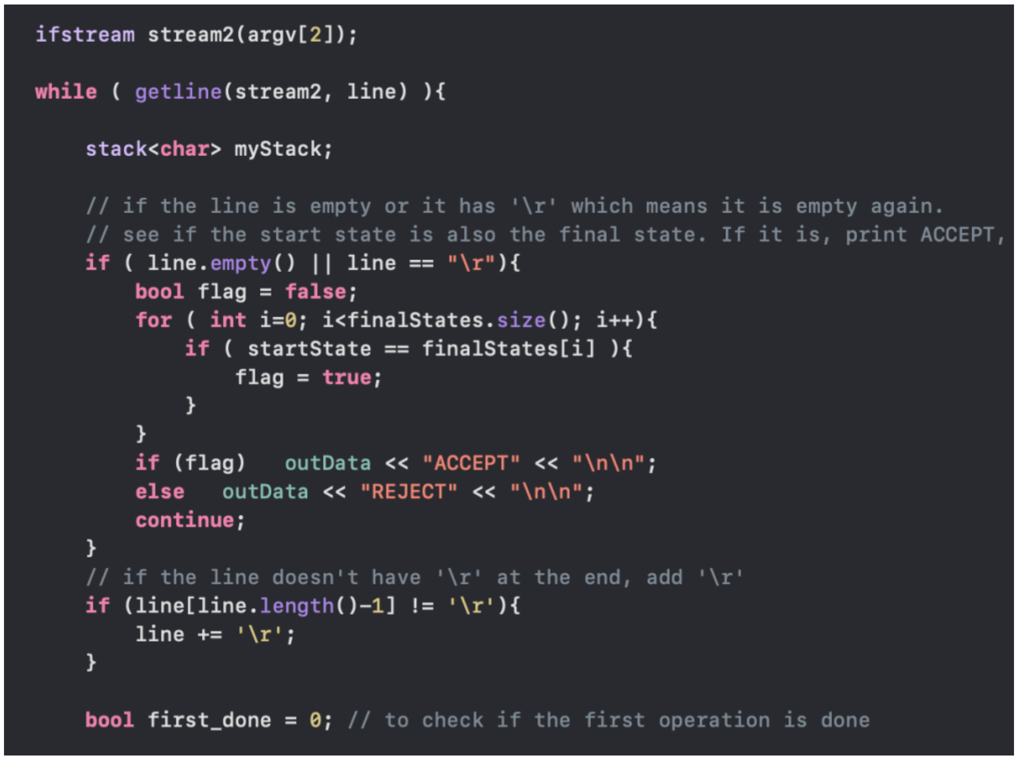
After checking we have a rule that starts with the initial rule, now we can check if there is a specific rule that starts with the initial state and also have the specific input symbol, if there is, then it goes inside the first if statement. Then, flag turns into true since we found the rule. Then check for what to pop, if the element that will be popped is not equal to ‘e’ or the top of the stack, that means the rule is not correct and flag turns into false again, after that, we keep looking for the correct rule. After popping step, we need to push something. If the pushed element is ‘e’, otherwise push what the rule says. At the and print the stack and return the resulting state of the rule.



```
  // at the end if we couldn't find any rule that fits best we'll take the rule that takes
  // input 'e' and starts with the Starting State
  if ( stocked != NULL){
      isSymbolUsed = false;
      return doOperation(myStack, stState, rules, 'e');
  }
  return resultingState;
```

And at the end, we’re just checking if the stocked rule is empty or not. If it is not empty, then it means we used symbol ‘e’ as an input and we should make it forward by calling the function again with the ‘e’ input.

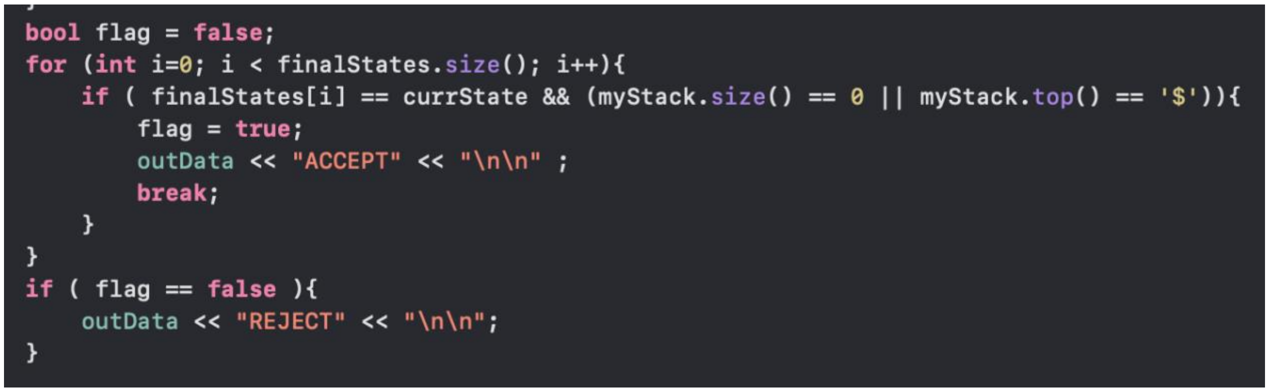
In main function, there are just some input taking parts and checking if the inputs are appropriate according to the alphabet. If they're not appropriate, then, print an error message.



```
ifstream stream2(argv[2]);

while ( getline(stream2, line) ){

    stack<char> myStack;

    // if the line is empty or it has '\r' which means it is empty again.
    // see if the start state is also the final state. If it is, print ACCEPT,
    if ( line.empty() || line == "\r"){
        bool flag = false;
        for ( int i=0; i<finalStates.size(); i++){
            if ( startState == finalStates[i] ){
                flag = true;
            }
        }
        if (flag)   outData << "ACCEPT" << "\n\n";
        else   outData << "REJECT" << "\n\n";
        continue;
    }
    // if the line doesn't have '\r' at the end, add '\r'
    if (line[line.length()-1] != '\r'){
        line += '\r';
    }

    bool first_done = 0; // to check if the first operation is done
```

In this part, the input file is read. And since there was some confusion in the dev server because getline function adds an extra '\r' character at the end of the some lines, I decided to consider it and if there is no '\r' at the end of the line, I add it and take care of the string as there is always '\r' at the end of the every line. If the line is empty or just equals to '\r' ,then checks if the initial state is also the final state, if it is, print ACCEPT, and print REJECT otherwise.

Later on this part, the same steps as 'doOperation' part is done for the initial state. Since initial state is different than the others, I decided to implement it as a separate blocks of code.



```
bool flag = false;
for (int i=0; i < finalStates.size(); i++){
    if ( finalStates[i] == currState && (myStack.size() == 0 || myStack.top() == '$')){
        flag = true;
        outData << "ACCEPT" << "\n\n" ;
        break;
    }
}
if ( flag == false ){
    outData << "REJECT" << "\n\n";
}
```

At the end, it just checks whether the stack is empty or the top element of the stack is equal to the '$' sign and current state is one of the final states. If it is then, print accept. Otherwise, print REJECT.

REFERENCE:

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

https://www.cs.wcupa.edu/rkline/fcs/pdas.html