Deterministic Finite Automaton (DFA)

Hany Adel (Faculty of Science)
Hadeer Ahmed (Faculty of Science)
Mannar Mohamed

Index

1-Introduction

1.1.Finite Automata(FA) :

These are good models for computers with a particularly limited amount of memory. A finite automaton is the simplest machine to acknowledge patterns. The finite automaton or finite state machine is an abstract machine that has five elements or tuples that are: {Q, Σ,q,δ,F}.

Finite automaton is a collection of states and rules for moving from one state to another, but it depends upon the applied input symbol. Basically, it's an abstract model of a computing device.

Finite automata have many advantages such as:

1) their simplicity makes inexperienced developers implement with little knowledge,

2) easy to Transfer from summary illustration to coded implementation.

Also, they have some disadvantages as:

1) It can be difficult in implementation of large systems using finite automata to manage and maintain without studied design,

2) State transitions have conditions that are fixed but this disadvantage we can fix by using the Fuzzy State Machine".

1.2. Deterministic Finite Automaton (DFA):

This type of finite automata accepts or rejects a given string of alphabets, via running through a state sequence uniquely determined by the string. In a DFA, for a specific input character, the machine goes to at least one state only. A transition function is defined on every state for each input alphabet. Also, in DFA null (or ε) move isn't allowed so (ε) does not appear in DFA. DFA has many applications that are used in our daily life directly or indirectly such as: security analysis, text parsing, speech recognition, etc.

2-Machine Design

2.1.Formal Definition Of Deterministic Finite Automaton:

A deterministic finite automaton is a 5-tuple, (Q, Σ, δ, q0, F), consisting of:

1- Q: finite set of states

2- Σ: a finite set of input symbols called the alphabet

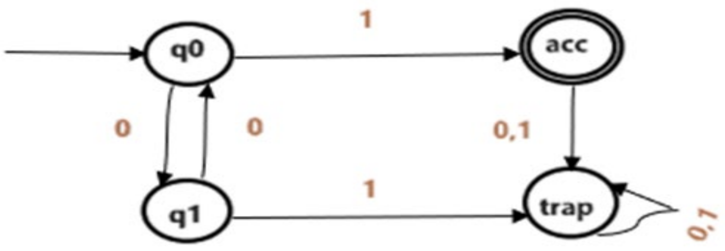
3- δ: a transition function Q × Σ → Q. This function mapping is usually represented by a transition table or a transition diagram.

4- q0: an initial or start state. q0 ∈ Q.

5-F: a set of accepted states. F ⊆ Q. There may be more than one final state.

2.2.Examples:

1- FA with ∑ = {0, 1} accepts the strings with an even number of 0's followed.



Q= {q0, q1, acc,trap},

Σ= {0,1},

q0:is the first state,

δ is described as

States	0	1
q0	q1	acc
q1	q0	trap
acc	trap	trap
trap	trap	trap

1-The machine read the inputs “001”.

q0 → q1→q0→acc. (Accept string)

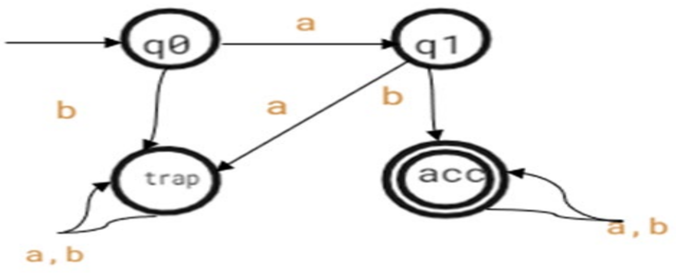
2-The machine read the inputs “010”.

q0→ q1→ trap → trap. (Reject string)

3- The machine reads the inputs “00001”.

q0 →q1 → q0 → q1→ q0 → acc. (Accept string)

2- Draw a DFA for the language accepting strings starting with ‘ab’ over input alphabets ∑ = {a, b}.



Q= {q0, q1, acc,trap},

Σ={a,b},

q0 is the first state,

δ is described as

States	A	b
q0	q1	trap
q1	Trap	acc
acc	acc	acc
trap	trap	trap

1-The machine reads the inputs “abbb”.

q0→ q1→acc→acc→acc. (Accept string)

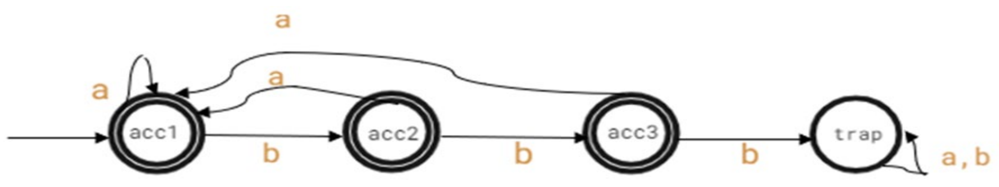
2- The machine reads the inputs “aba”.

q0→q1→acc→acc. (Accept string)

3- The machine reads the inputs “babb”.

q0→trap. (Reject string)

3-Design a deterministic finite automaton M that accepts the language L(M) = {w ∈ {a,b}* : w does not contain three consecutive b’s}.



Where

Q= {acc1, acc2, acc3, trap},

Σ={a,b},

acc1 is the first state,

δ is described as

States	a	b
acc1	acc1	acc2
acc2	acc1	acc3
acc3	acc1	trap
trap	trap	trap

1-The machine reads the inputs “aaab”.

acc1→acc1→acc1→acc1→acc2. (Accept string)

2-The machine reads the inputs “abba”.

acc1→ acc1→acc2→acc3→acc1. (Accept string)

3-The machine reads the inputs “abbb”.

acc1→ acc1→acc2→acc3→trap. (Reject string)

3-Machine Implementation

▪ 3.1. codes' functions explaining

* Create function for assign transition table

```
map< pair< string,char >, string > creat_trans_table()
{

    map< pair< string,char >, string > trans_table;   // define the transition table

    //initialize q0 state
    trans_table.insert({make_pair("q0",'0'),"q1"});
    trans_table.insert({make_pair("q0",'1'),"acc"});

    //initialize q1 state
    trans_table.insert({make_pair("q1",'0'),"q0"});
    trans_table.insert({make_pair("q1",'1'),"trap"});

    //initialize accept state
    trans_table.insert({make_pair("acc",'0'),"trap"});
    trans_table.insert({make_pair("acc",'1'),"trap"});

    //initialize trap state
    trans_table.insert({make_pair("trap",'0'),"trap"});
    trans_table.insert({make_pair("trap",'1'),"trap"});

    cout<<" << TRANSITION TABLE >> "<<endl;
    for(auto p:trans_table)
    {
        cout<<"( "<< p.first.first << ", " <<p.first.second<<" ) ->"<<p.second <<endl;
    }

cout<<"<<==================================================================================
===============>>"<<"\n\n";
    return trans_table;
}
```

This function defines the transition table by using "map" data structure. According to the following expression "(Q*Σ) →Q", we decide to use map structure to assign the transition table. We use complex map which make the key to this map is "pair" which present (Q, Σ) the current state and the alphabet, to make a transition to a new state which is presented in map as value of the key. This means that the transition table is presented as a key and its value.

By using this structure to assign the transition table, the complexity of this function will be O (log n) for insertion and O(1) for access time so this makes the program work fast.

* create function to check the validity of the string

```
void checkString(string s)
{
    vector<string> state_sequance; // push the steps of state in this
vector to print it

    string intial_state="q0";// q0 is the start state
    state_sequance.push_back(intial_state);

    map< pair< string,char >, string > trans_table= creat_trans_table();
// call the defined transition table


    // var to check if state is trap state or not, if var still (0) this
mean that string does not enter in trap state
    int check_state_trap=0;


    // begin to iterate on the string to check state
    for(int i=0; i<s.size(); i++)
    {

        if(trans_table[ {intial_state,s[i]}]!="trap")
        {
            state_sequance.push_back(trans_table[ {intial_state,s[i]}]);
//push which state we locate in

            // if the state is not trap state,we should continue and
make initial state equal to state we locate in
            intial_state=trans_table[ {intial_state,s[i]}];
        }
        else
        {
            state_sequance.push_back(trans_table[ {intial_state,s[i]}]);
//push which state we locate in

            check_state_trap=1;
            break;
        }

    }

    //     cout<<"string which entered : "<<s<<endl;
    for (int i=0;i<state_sequance.size();i++)
    {
        if(i==state_sequance.size()-1)
        {
            cout<<state_sequance[i]<<endl;
            break;
        }

        cout<<state_sequance[i]<<" -> ";
    }


if(check_state_trap==0&&state_sequance.back()!="q0"&&state_sequance.back
()=="acc") //check that there is no trap state and the string is not
empty
    {
        cout<<"the string is Accepted"<<endl;
    }

    else{
        cout<<"this string is REJECTED"<<endl;
    }
}
```

this function takes the string as a parameter, then we create var of vector data type to push the steps of state in this vector to print it later, then we call the defined transition table from `creat_trans_table()` and assign it on new var which have the same data type "map". then declare a var `check_state_trap` to check if state is trap state or not, if var still (0) this mean that the string does not enter in trap state, before we start to iterate the string we define a var `intial_state` and assign "q0" to it, then we begin to iterate on the string to check state, in each iteration we push in `state_sequance` which state we currently located in, then check if the state is not trap state, we should continue and make `intial_state` equal to the value of the key `{intial_state,s[i]}` which is presented as `trans_table[{intial_state,s[i]}]` so this mean that the new state we forward to it is assigned to `intial_state`, but if the state become "trap" we make `check_state_trap` equal to 1, and then break the iteration and exit the loop. After iteration we check if the string is accepted or rejected.

There are three machine we coded them each machine has specific conditions to say that the input string is accepted or not

Machine 1,2:

- The `check_state_trap==0` which mean that there is no trap state in the sequence
- `state_sequance.back()!="q0"` Which mean that this machine does not accept empty string, because when the last element is “q0” this mean that there are no string pushed to state sequence
- `state_sequance.back()=="acc"` This to make sure that we reach to accept state in final because first condition not enough to say that the string is accepted

```
if(check_state_trap==0&&state_sequance.back()!="q0"&&state_sequance.back
()=="acc") //check that there is no trap state and the string is not
empty
    {
        cout<<"the string is Accepted"<<endl;
    }

    else{
        cout<<"this string is REJECTED"<<endl;
    }
}
```

Machine 3:

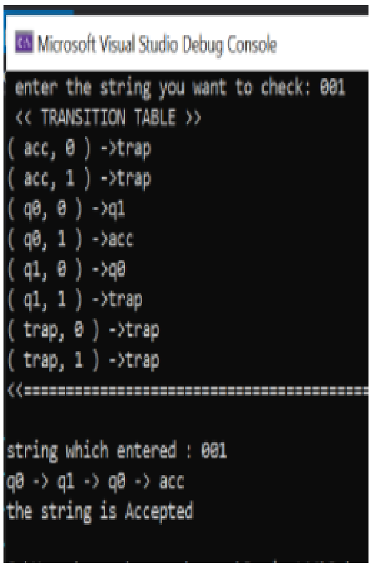
- Same conditions like machine 1,2 except that this machine accepts empty strings so `state_sequance.back()!="q0"` does not condition
- There are three accept state so make “or” operator to granite that any accept state led to accept string

```
if(check_state_trap==0&&(state_sequance.back()=="acc1"||state_sequance.back()=="
acc2"||state_sequance.back()=="acc3"))
    {
        cout<<"the string is Accepted"<<endl;
    }

    else
    {
        cout<<"this string is REJECTED"<<endl;
    }
```

▪ 3.2. Run results for machines

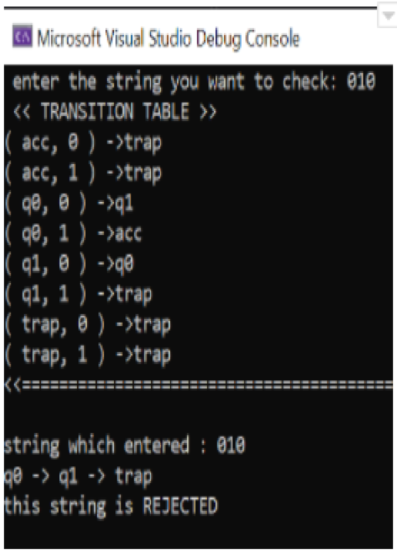
❖ Run result for Machine 1:



```
Microsoft Visual Studio Debug Console
enter the string you want to check: 001
<< TRANSITION TABLE >>
( acc, 0 ) ->trap
( acc, 1 ) ->trap
( q0, 0 ) ->q1
( q0, 1 ) ->acc
( q1, 0 ) ->q0
( q1, 1 ) ->trap
( trap, 0 ) ->trap
( trap, 1 ) ->trap
<<=========================================

string which entered : 001
q0 -> q1 -> q0 -> acc
the string is Accepted
```

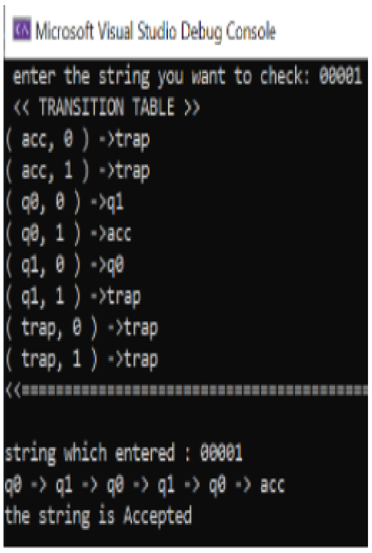
S1:001 (Accepted)



```
Microsoft Visual Studio Debug Console
enter the string you want to check: 010
<< TRANSITION TABLE >>
( acc, 0 ) ->trap
( acc, 1 ) ->trap
( q0, 0 ) ->q1
( q0, 1 ) ->acc
( q1, 0 ) ->q0
( q1, 1 ) ->trap
( trap, 0 ) ->trap
( trap, 1 ) ->trap
<<=========================================

string which entered : 010
q0 -> q1 -> trap
this string is REJECTED
```

s2:010 (Rejected)

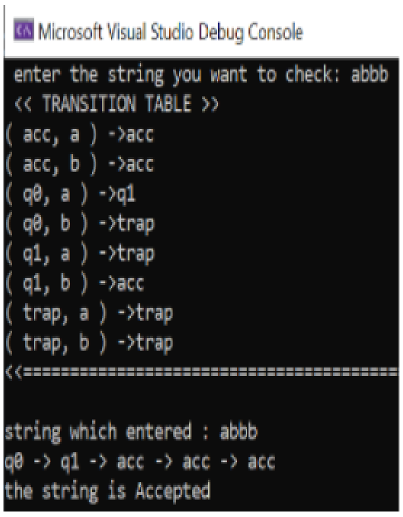


```
Microsoft Visual Studio Debug Console
enter the string you want to check: 00001
<< TRANSITION TABLE >>
( acc, 0 ) ->trap
( acc, 1 ) ->trap
( q0, 0 ) ->q1
( q0, 1 ) ->acc
( q1, 0 ) ->q0
( q1, 1 ) ->trap
( trap, 0 ) ->trap
( trap, 1 ) ->trap
<<=========================================

string which entered : 00001
q0 -> q1 -> q0 -> q1 -> q0 -> acc
the string is Accepted
```

S3:00001 (Accepted)

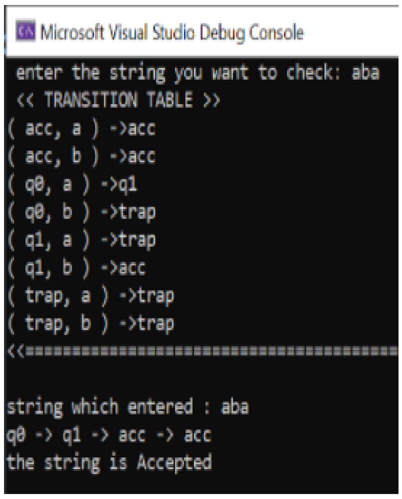
❖ Run result for Machine 2:(this machine accepts empty string)



```
Microsoft Visual Studio Debug Console
enter the string you want to check: abbb
<< TRANSITION TABLE >>
( acc, a ) ->acc
( acc, b ) ->acc
( q0, a ) ->q1
( q0, b ) ->trap
( q1, a ) ->trap
( q1, b ) ->acc
( trap, a ) ->trap
( trap, b ) ->trap
<<=========================================

string which entered : abbb
q0 -> q1 -> acc -> acc -> acc
the string is Accepted
```

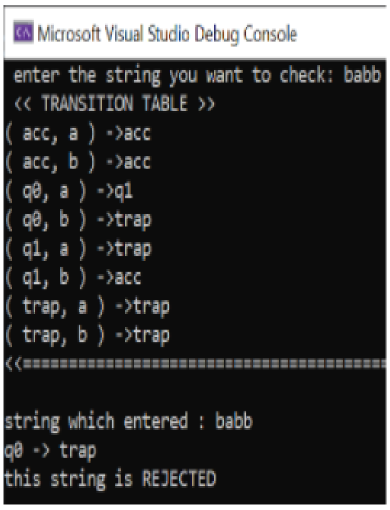
S1:abbb (Accepted)



```
Microsoft Visual Studio Debug Console
enter the string you want to check: aba
<< TRANSITION TABLE >>
( acc, a ) ->acc
( acc, b ) ->acc
( q0, a ) ->q1
( q0, b ) ->trap
( q1, a ) ->trap
( q1, b ) ->acc
( trap, a ) ->trap
( trap, b ) ->trap
<<=========================================

string which entered : aba
q0 -> q1 -> acc -> acc
the string is Accepted
```

s2:aba (Accepted)

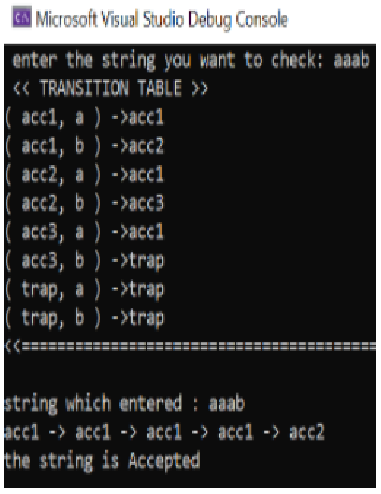


```
Microsoft Visual Studio Debug Console
enter the string you want to check: babb
<< TRANSITION TABLE >>
( acc, a ) ->acc
( acc, b ) ->acc
( q0, a ) ->q1
( q0, b ) ->trap
( q1, a ) ->trap
( q1, b ) ->acc
( trap, a ) ->trap
( trap, b ) ->trap
<<=========================================

string which entered : babb
q0 -> trap
this string is REJECTED
```

s3:babb(Rejected)

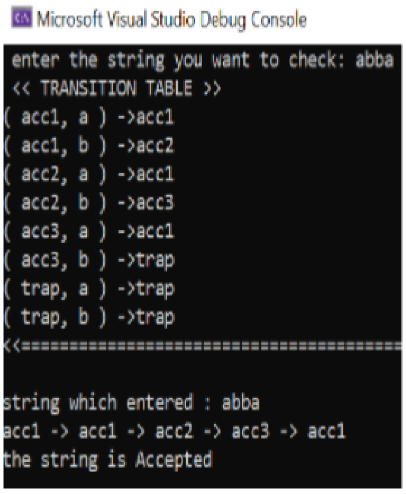
❖ Run result for Machine 3:(this machine accepts empty string)



```
Microsoft Visual Studio Debug Console
enter the string you want to check: aaab
<< TRANSITION TABLE >>
( acc1, a ) ->acc1
( acc1, b ) ->acc2
( acc2, a ) ->acc1
( acc2, b ) ->acc3
( acc3, a ) ->acc1
( acc3, b ) ->trap
( trap, a ) ->trap
( trap, b ) ->trap
<<========================================

string which entered : aaab
acc1 -> acc1 -> acc1 -> acc1 -> acc2
the string is Accepted
```

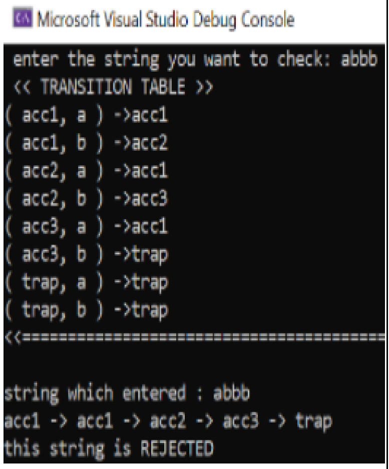
S1:aaab (Accepted)



```
Microsoft Visual Studio Debug Console
enter the string you want to check: abba
<< TRANSITION TABLE >>
( acc1, a ) ->acc1
( acc1, b ) ->acc2
( acc2, a ) ->acc1
( acc2, b ) ->acc3
( acc3, a ) ->acc1
( acc3, b ) ->trap
( trap, a ) ->trap
( trap, b ) ->trap
<<========================================

string which entered : abba
acc1 -> acc1 -> acc2 -> acc3 -> acc1
the string is Accepted
```

s2:abba (Accepted)



```
Microsoft Visual Studio Debug Console
enter the string you want to check: abbb
<< TRANSITION TABLE >>
( acc1, a ) ->acc1
( acc1, b ) ->acc2
( acc2, a ) ->acc1
( acc2, b ) ->acc3
( acc3, a ) ->acc1
( acc3, b ) ->trap
( trap, a ) ->trap
( trap, b ) ->trap
<<========================================

string which entered : abbb
acc1 -> acc1 -> acc2 -> acc3 -> trap
this string is REJECTED
```

s3:abbb(Rejected)

Appendix 1

❖ Code of three machine is available on GitHub:
https://github.com/Hany-Farag/DFA-machines-codes

▪ 4.1. code for first machine:

```
#include<bits/stdc++.h>
using namespace std;

map< pair< string,char >, string > creat_trans_table()
{
    /*
    ===============
    state|  0 | 1
    ===============
    q0   |q1  |acc
    -----|----|-----
    q1   |q0  |trap
    -----|----|-----
    acc  |trap|trap
    -----|----|-----
    trap |trap|trap
    -----|----|-----

    */
    map< pair< string,char >, string > trans_table;   // define the transition table

    //initialize q0 state
    trans_table.insert({make_pair("q0",'0'),"q1"});
    trans_table.insert({make_pair("q0",'1'),"acc"});

    //initialize q1 state
    trans_table.insert({make_pair("q1",'0'),"q0"});
    trans_table.insert({make_pair("q1",'1'),"trap"});

    //initialize accept state
    trans_table.insert({make_pair("acc",'0'),"trap"});
    trans_table.insert({make_pair("acc",'1'),"trap"});

    //initialize trap state
    trans_table.insert({make_pair("trap",'0'),"trap"});
    trans_table.insert({make_pair("trap",'1'),"trap"});

    cout<<" << TRANSITION TABLE >> "<<endl;
    for(auto p:trans_table)
    {
        cout<<"( "<< p.first.first << ", " <<p.first.second<<" ) ->"<<p.second
<<endl;
    }

cout<<"<<===============================================================================
====================>>"<<"\n\n";
    return trans_table;
}

void checkString(string s)
{
    vector<string> state_sequance; // push the steps of state in this vector to
print it

    string intial_state="q0";// q0 is the start state
    state_sequance.push_back(intial_state);

    map< pair< string,char >, string > trans_table= creat_trans_table(); // call the
defined transition table


    // var to check if state is trap state or not, if var still (0) this mean that
string does not enter in trap state
    int check_state_trap=0;


    // begin to iterate on the string to check state
    for(int i=0; i<s.size(); i++)
    {
      state_sequance.push_back(trans_table[ {intial_state,s[i]}]); //push which
state we locate in
        if(trans_table[ {intial_state,s[i]}]!="trap")
        {
            // if the state is not trap state,we should continue and make initial
state equal to state we locate in
            intial_state=trans_table[ {intial_state,s[i]}];
        }
        else
        {
            check_state_trap=1;
            break;
        }

    }

    //     cout<<"string which entered : "<<s<<endl;
    for (int i=0;i<state_sequance.size();i++)
    {
        if(i==state_sequance.size()-1)
        {
            cout<<state_sequance[i]<<endl;
            break;
        }

        cout<<state_sequance[i]<<" -> ";
    }


if(check_state_trap==0&&state_sequance.back()!="q0"&&state_sequance.back()=="acc")
//check that there is no trap state and the string is not empty
    {
        cout<<"the string is Accepted"<<endl;
    }

    else{
        cout<<"this string is REJECTED"<<endl;
    }
}
int main()
{
    string str;
    cout<<" enter the string you want to check: ";
    cin>>str;
    checkString(str);
    return 0;
}
```

▪ 4.2. code for second machine:

```
#include<bits/stdc++.h>
using namespace std;

map< pair< string,char >, string > creat_trans_table()
{
    /*
    ===============
    state|  a | b
    ===============
    q0   |q1  |acc
    -----|----|-----
    q1   |trap|acc
    -----|----|-----
    acc  |acc |acc
    -----|----|-----
    trap |trap|trap
    -----|----|-----

    */
    map< pair< string,char >, string > trans_table;   // define the transition table

    //initialize q0 state
    trans_table.insert({make_pair("q0",'a'),"q1"});
    trans_table.insert({make_pair("q0",'b'),"trap"});

    //initialize q1 state
    trans_table.insert({make_pair("q1",'a'),"trap"});
    trans_table.insert({make_pair("q1",'b'),"acc"});

    //initialize accept state
    trans_table.insert({make_pair("acc",'a'),"acc"});
    trans_table.insert({make_pair("acc",'b'),"acc"});

    //initialize trap state
    trans_table.insert({make_pair("trap",'a'),"trap"});
    trans_table.insert({make_pair("trap",'b'),"trap"});

    cout<<" << TRANSITION TABLE >> "<<endl;
    for(auto p:trans_table)
    {
        cout<<"( "<< p.first.first << ", " <<p.first.second<<" ) ->"<<p.second
<<endl;
    }

cout<<"<<==========================================================================
====================>>"<<"\n\n";
```

```
    return trans_table;
}

void checkString(string s)
{
    vector<string> state_sequance; // push the steps of state in this vector to 
print it

    string intial_state="q0";// q0 is the start state
    state_sequance.push_back(intial_state);

    map< pair< string,char >, string > trans_table= creat_trans_table(); // call the 
defined transition table


    // var to check if state is trap state or not, if var still (0) this mean that 
string does not enter in trap state
    int check_state_trap=0;


    // begin to iterate on the string to check state
    for(int i=0; i<s.size(); i++)
    {
      state_sequance.push_back(trans_table[ {intial_state,s[i]}]); //push which 
state we locate in
        if(trans_table[ {intial_state,s[i]}]!="trap")
        {
            // if the state is not trap state,we should continue and make initial 
state equal to state we locate in
            intial_state=trans_table[ {intial_state,s[i]}];
        }
        else
        {
            check_state_trap=1;
            break;
        }

    }

    // print the string and state seq. of this string
    cout<<"string which entered : "<<s<<endl;
    for (int i=0;i<state_sequance.size();i++)
    {
        if(i==state_sequance.size()-1)
        {
            cout<<state_sequance[i]<<endl;
            break;
        }

        cout<<state_sequance[i]<<" -> ";
    }


if(check_state_trap==0&&state_sequance.back()!="q0"&&state_sequance.back()=="acc")
//check that there is no trap state and the string is not empty
    {
        cout<<"the string is Accepted"<<endl;
    }

    else{
        cout<<"this string is REJECTED"<<endl;
    }
}
int main()
{
    string str;
    cout<<" enter the string you want to check: ";
    cin>>str;
    checkString(str);
    return 0;
}
```

▪ 4.3. code for third machine:

```
#include<bits/stdc++.h>
using namespace std;

map< pair< string,char >, string > creat_trans_table()
{
    /*
    this machine accept empty string
    ===============
    state|  a | b
    ===============
    acc1 |acc1|acc2
    -----|----|-----
    acc2 |acc1|acc3
    -----|----|-----
    acc3 |acc1|trap
    -----|----|-----
    trap |trap|trap
    -----|----|-----

    */
    map< pair< string,char >, string > trans_table;   // define the transition table

    //initialize acc1 state
    trans_table.insert({make_pair("acc1",'a'),"acc1"});
    trans_table.insert({make_pair("acc1",'b'),"acc2"});

    //initialize acc2 state
    trans_table.insert({make_pair("acc2",'a'),"acc1"});
    trans_table.insert({make_pair("acc2",'b'),"acc3"});

    //initialize acc3 state
    trans_table.insert({make_pair("acc3",'a'),"acc1"});
    trans_table.insert({make_pair("acc3",'b'),"trap"});

    //initialize trap state
    trans_table.insert({make_pair("trap",'a'),"trap"});
    trans_table.insert({make_pair("trap",'b'),"trap"});

    cout<<" << TRANSITION TABLE >> "<<endl;
    for(auto p:trans_table)
    {
        cout<<"( "<< p.first.first << ", " <<p.first.second<<" ) ->"<<p.second
<<endl;
    }

cout<<"<<===========================================================================
=====================>>"<<"\n\n";
    return trans_table;
}

void checkString(string s)
{
    vector<string> state_sequance; // push the steps of state in this vector to
print it

    string intial_state="acc1";// acc1 is the start state
    state_sequance.push_back(intial_state);

    map< pair< string,char >, string > trans_table= creat_trans_table(); // call the
defined transition table

    // var to check if state is trap state or not, if var still (0) this mean that
string does not enter in trap state
    int check_state_trap=0;


    // begin to iterate on the string to check state
    for(int i=0; i<s.size(); i++)
    {
       state_sequance.push_back(trans_table[ {intial_state,s[i]}]); //push which
state we locate in
        if(trans_table[ {intial_state,s[i]}]!="trap")
        {
            // if the state is not trap state,we should continue and make initial
state equal to state we locate in
            intial_state=trans_table[ {intial_state,s[i]}];
        }
        else
        {
            check_state_trap=1;
            break;
        }

    }

    // print the string and state seq. of this string
    cout<<"string which entered : "<<s<<endl;
    for (int i=0; i<state_sequance.size(); i++)
    {
        if(i==state_sequance.size()-1)
        {
            cout<<state_sequance[i]<<endl;
            break;
        }

        cout<<state_sequance[i]<<" -> ";
    }

    //check that there is no trap state and the string is not empty

if(check_state_trap==0&&(state_sequance.back()=="acc1"||state_sequance.back()=="acc2
"||state_sequance.back()=="acc3"))
    {
        cout<<"the string is Accepted"<<endl;
    }

    else
    {
        cout<<"this string is REJECTED"<<endl;
    }
}
int main()
{
    string str;
    cout<<" enter the string you want to check: ";
    cin>>str;
    if(str=="")
    {

        checkString("");
        cout<<"<<This machine accept empty string>>"<<endl;
    }
    else
    {

        checkString(str);
    }
    return 0;
}
```

Appendix 2

The role of each member of the team

1-Hany Adel (the team leader) do the following:

A. Responsible for machine implementation.
B. Illustrate functions in the code of every machine.
C. Search for the introduction of the report.
D. Revise the report sections and edit them
E. Code the three machine and add them to Appendix 1 section
F. Followed-up the team and forward tasks for them

2- Mannar Mohamed:

A. Design the cover.
B. participate in illustrating the functions in the code of every machine.
C. organising and writing the report as a whole.
D. Search for examples about machines.

3-Hadeer Ahmed:

A. Searching for the introduction and rewriting it.
B. Responsible for machine design.
C. Searching for references and writing them in a scientific way.
D. Design(making) appendix 2.

References

[1] Ramaiah K.D., *Introduction to Automata and Compiler Design* (2011).

[2] Sipser, M., n.d. *Introduction to the theory of computation* (2013).

[3] GeeksforGeeks. 2021. Introduction of Finite Automata - GeeksforGeeks. [online] Available at: <https://www.geeksforgeeks.org/introduction-of-finite-automata/> [Accessed 25 December 2021].

[4] www.javatpoint.com. 2021. Examples of DFA - Javatpoint. [online] Available at: <https://www.javatpoint.com/examples-of-deterministic-finite-

automata> [Accessed 26 December 2021].

[5] Singhal, A. and Vidyalay, G., 2021. DFA Solved Examples | How to Construct DFA | Gate Vidyalay. [online] Gatevidyalay.com. Available at: <https://www.gatevidyalay.com/construction-of-dfa-examples-dfa-solved-examples/> [Accessed 26 December 2021].