CSE432: Automata and Computability
Term Project

Abdelrahman Aboel Nasr:	**REPORT**	**19P6517**
AbdulRaouf Monir:	**NFA_DFA + GUI**	**19P4442**
Salma Osama:	**REPORT**	**18P5097**
Serag Eldin Mohamed:	**REPORT**	**19P1183**
Sohayla Ihab:	**CFG_PDA + REPORT**	**19P7343**
Yasmin Haitham:	**GRAMMAR + CFG**	**18P3102**

Submitted to:

Prof. Gamal A. Ebrahim

TA Sally E. Shaker

Contents

Definition :

An automaton that makes use of a stack is known as a pushdown automaton (PDA) in the theory of computation, a subfield of theoretical computer science.
Pushdown automata are used in theories about what can be computed by machines. They are more capable than finite-state machines but less capable than Turing machines

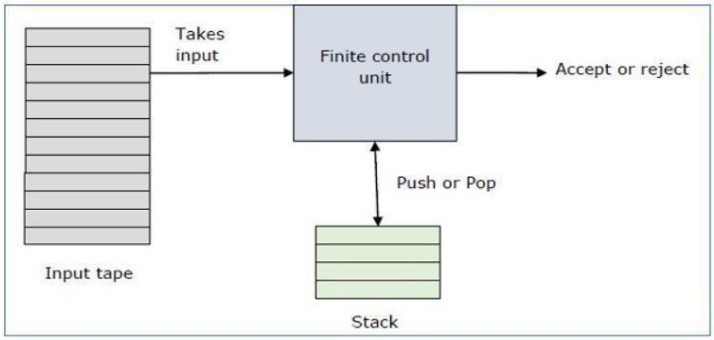
Deterministic pushdown automata can recognize all deterministic context-free languages while nondeterministic ones can recognize all context-free languages, with the former often used in parser design.

The phrase "pushdown" describes how the stack can be thought of as being "pushed down" because operations are only ever performed on the top element.

A pushdown automata has three components

- An input tape
- A Finite control unit
- A stack with infinite size

It may or may not read an input symbol, but it has to read the top of the stack in every transition.



A PDA can formally described as a 7 tuple $(Q, \sum, S, \delta, q_0, I, F)$

- **Q** is the finite number of states
- $\sum$ is input alphabet
- **S** is stack symbols
- **δ** is the transition function: $Q \times (\sum \cup \{\varepsilon\}) \times S \times Q \times S^*$
- **q_0** is the initial state ($q_0 \in Q$)
- **I** is the initial stack top symbol ($I \in S$)
- **F** is a set of accepting states ($F \in Q$)

Advantages & Disadvantages:

Advantages :

- Pushdown Automata (PDA) can verify regular language as well as context free language. • A stack is an additional component of a PDA. Additional memory is provided by the stack.

- Languages can be accepted by the PDA in two different ways: by the empty stack and by the final state. • There is a known algorithm available to construct

PDA from a grammar (if the grammar is context free). • For all regular languages,

we can construct a deterministic PDA.

Disadvantages:

- All nondeterministic finite automata can be converted to their corresponding deterministic automata, however all nondeterministic PDAs cannot be converted to deterministic PDAs.

- There is no known algorithm to determine if there exists a deterministic PDA for a grammar.

- The expressive power of non-deterministic PDA is more than the expressive power of deterministic PDA. It means there exists a context free grammar for which we cannot construct deterministic PDA.

For example, we cannot construct deterministic PDA for L = { W | W is Palindrome }

- It is undecidable if a PDA recognizes, Σ^* set of all strings over input characters.

- PDA is unable to understand context-sensitive and permissive language.

Problems it Solves Or Overcome:

Pushdown automata used to solve many problems such that it's used for solving Tower of Hanoi Problem which considered of high complexity. Also, FAs can recognize all regular languages, but sometimes PDAs can do it "better".

Sometimes the stack can actually help. For example, PDAs are very good at counting things.

Consider the problem of designing an automaton to recognize strings with k occurrences of 101, where k is some constant.

Also pushdown automata can recognize context free languages. The classic example of a context-free language which is not regular is the language $\{ 0^n 1^n \mid n \quad N \}$

Since basic pushdown automata 'stack machine' capable of an arbitrary level of recursions. such a machine can cope with calculating Fibonacci(n) recursively for any large n, for instance, without croaking. This a FMS cannot do because, with only a finite set of states (however large that set might be), it cannot have enough to cope with an *arbitrarily* large n if required to calculate Fibonacci(n) recursively - by definition. This holds true for all calculations involving recursion.

Also since, Finite state automata cannot parse context free languages that are not regular For instance, there is no FSA for solving the decision problem: In this particular case, the problem is the absence Given w ∈ {1,0 }*, Does w∈ L = { 0^n 1^n: n ≥ 0} ?

In this particular case, the problem is the absence Given w∈ 1,0{ }*, Does w∈ L = 0{ 1 : n ≥ ?}0 n n In this particular case, the problem is the absence in FSAs of a mechanism for keeping track of the occurrences of a symbol in a string. Such a problem needs an additional string which is a stack so, Pushdown automata solves this problem.

Shortages and Drawbacks:

It is known that the class of PDAs is not closed under intersection and complementation. Thus, decision problems like inclusion and equivalence are undecidable for PDAs.

Extra Knowledge:

Deterministic vs. nondeterministic PDA

A pushdown automaton is a deterministic pushdown automaton (DPDA) if

∀ q ∈ Q ∧ ∀ a ∈ A ∧ ∀ γ ∈ Γ ∪ { λ } ∧ ∀ α ∈ Γ ∪ { λ }

∂ (q, a, γ, α) is a non-empty singleton set

A pushdown automaton is a nondeterministic pushdown automaton (NPDA) if

∃ q ∈ Q ∨ ∃ a ∈ A ∪ { λ } ∨ ∃ γ ∈ Γ ∪ { λ } ∨ ∃ α ∈ Γ ∪ { λ }

∂ (q, a, γ, α) is a empty or not singleton set

Computation

A computation with input string W = a_1 ….. a_k in a DPDA is a list of pairs in Q X Γ , denoted

C(w) =< (q_0 ,λ),(q_1,σ_1),...,(q_k ,σ_k) >

Where :

$(\forall i = 1,\ldots,k)\ (\exists \gamma_i \in \Gamma \cup \{\lambda\} \vee \exists \alpha_i \in \Gamma \cup \{\lambda\})$

$\{(q_0, \sigma_i)\} = \partial\ (q_{i-1}, a_i, \gamma_i, \alpha_i)$

With $\sigma_0 = \lambda$ and $\forall i = 1, \ldots, k$ σ_i is the current top symbol in the stack at the particular stage in the computation

Accepting and Rejecting Computations

The previous computation is declared to be an accepting computation if its last pair (q_k, σ_k) satisfies $P(q_k, \sigma_k) : q_k \in F$

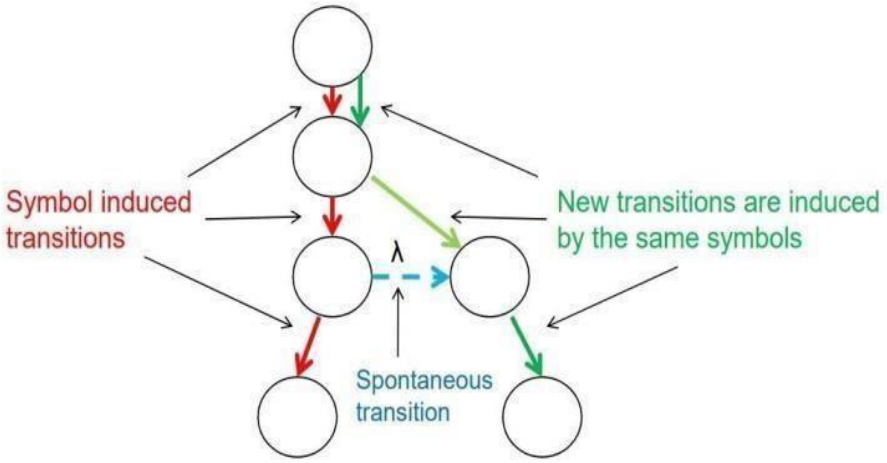
$\wedge\ \sigma_k = \lambda$

Conversely, the previous computation is a rejecting computation if its last pair satisfies instead

$$\neg P(q_k, \sigma_k) \Leftrightarrow q_k \notin F \wedge \sigma_k \neq \lambda$$

Nondeterministic spontaneous transitions

As with NFSAs, spontaneous transitions are modeled as the spawn of a new thread in an existing thread of computation. Graphically,



Nondeterministic Computations

The union of all computation threads generated either by the reading of W or created spontaneously as new computation threads during the reading constitutes the parsing or computation of a string w with a nondeterministic pushdown automaton.

A nondeterministic pushdown automaton accepts a string W if at least one accepting computation thread is finished with its computation.

Example

Let's illustrate the previous concepts by building a nondeterministic pushdown automaton for recognizing the context free language:

$$L = \{ 0^n 1^n : n \geq 0 \}$$

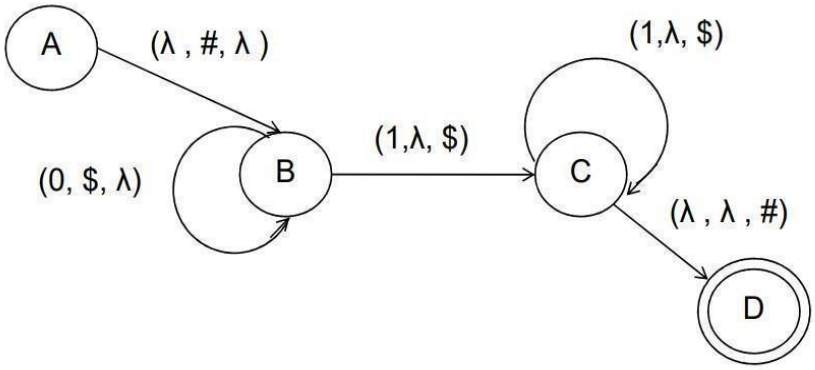
The automaton must have a mechanism for deciding whether there is the same number of 0's and 1's.

- This is done by pushing a symbol into the stack for each 0 read, and popping it out for each 1 read

- Define P=({A, B, C, D}, {0, 1}, {$,#}, t, A, {D}), where t is defined by the table

t	(λ,#,λ)	(0,$,λ)	(1,λ,$)	(λ,λ,#)
A	{B}			
B		{B}	{C}	
C			{C}	{D}

As usual, undefined transitions are assumed to be empty transitions



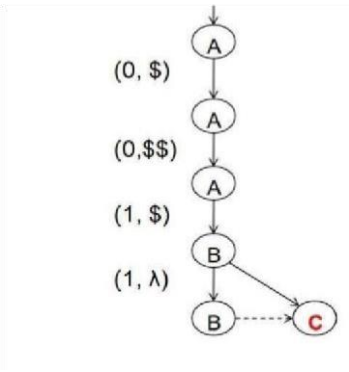
Threads of Configuration

A configuration is a triple in $A \times Q \times \Gamma^*$ where and are the alphabet and stack alphabet of a pushdown automaton, respectively.

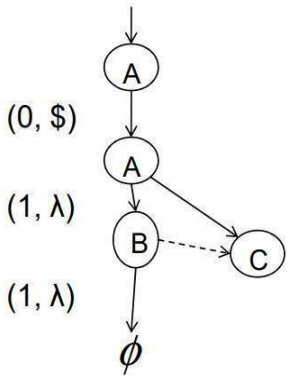
A thread of configurations is a sequence of configurations, denoted as

$(\lambda , q_0 , \lambda) \rightarrow (a_1 , q_1, s_1) \rightarrow .... \rightarrow (a_k , q_k, s_k)$ which is used to convey all information in a computation thread

Represent Some Computations



String 0011



String 011

Turing Machine:

Definition:

The turing machine is a mathematical model of computation which describes a machine that manipulates symbols on a strip of tape based on tabulated rules . This tape is divided into cells, each one of those cells is capable of holding a single symbol. however, it has the capability to execute any computing algorithm. Associated with the tape is a read-write head that can travel right or left on the tape and that can read and write a single symbol on each move. A Turing machine M is defined by

M = (Q,Σ, Γ, δ, q0,, F) , where where

Q is the set of internal states,

Σ is the input alphabet,

Γ is a finite set of symbols called the tape alphabet, δ is the transition

function,

∈ Γ is a special symbol called the blank, q0 ∈
Q is the initial state,

F ⊆ Q is the set of final states.

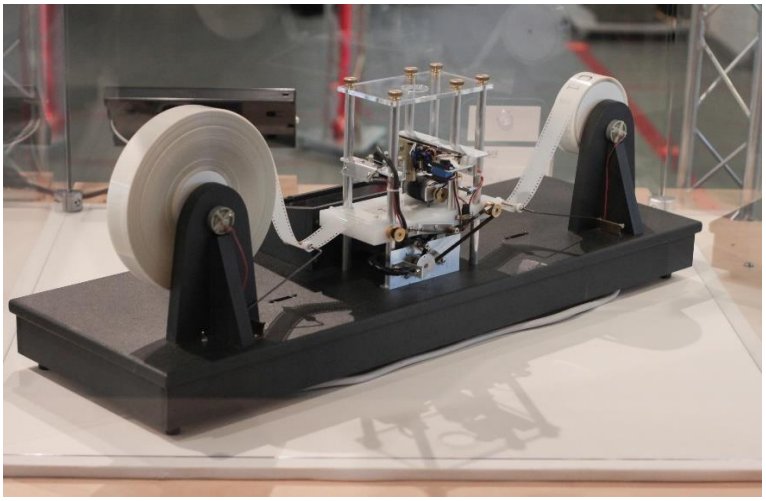


Figure 1 a turing machine

Implementations:

-The turing machine could be used as an example of a CPU that has the control of data manipulation done by a computer by using sequential memory to store data the machine is capable of identifying some arbitrary subset of valid strings of an alphabet. In addition to this ,The turing machine has the capability of processing an unspecified grammar, which indicated that it can evaluate first-order logic in an unlimited amount of methods. a famous example of this is the lambda calculus .The turing device can also mathematically models a machine that mechanically operates on a tape where there are symbols which the machine can read and write, at one time through using the tape head. Operation is determined by infinte set of elementary instructions such as in state 36 if the symbol seen is 0, write a 1 else if the symbol seen is 1 the state would be modified into 18; in other words the turing machine storage is simply one-dimensional array of cells, each of which can hold a single symbol. This array extends indefinitely in both directions and is therefore capable of holding an unlimited amount of information. We can visualize the Turing machine as a rather simple computer. It has a processing unit, which has a finite memory, and in its tape, it has a secondary storage of unlimited capacity. The instructions that such a computer can carry out are very limited: It can sense a symbol on its tape and use the result to decide what to do next. The only actions the machine can perform are to rewrite the current symbol, to change the state of the control, and to move the read-write head.

Advantages and disadvantages:

Disadvantages:

-Turing machines do not model the strengths of a particular arrangement reliably for instance, modern stored-program computers are actually examples of a more special form of abstract machine known as the random-access stored-program machine or abbreviation as RASP machine model

-Another disadvantage is that the do not model concurrency very well for example: there is a limit on the size of integer that can be calculated by an always halting nondeterministic Turing machine starting on a blank tape on the other hand there is

halting concurrents systems that can compute an integer of unbounded size

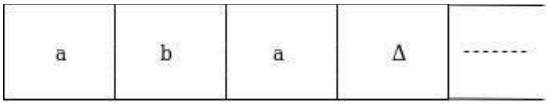
Advantages:

Turing devices has the advantage of unlimited memory in addition to this they are capable of simulating common computers. The turing machine have the merit of being simple in a sense that the current machine state has only constant size all the information you need in order to determine the next machine state is one symbol and one control state number. another advantage of Turing machine is that it has a tape head which can be propelled backward or forward , and the input tape can be scanned **Problems it can solve:**

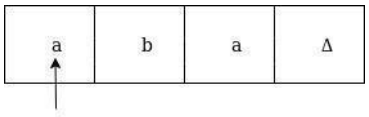
Turing machines are much more powerful than the automata machines . In fact, they solve precisely the set of all problems that can be solved by any digital computing device. Turing machines can do some simple addition , subtraction, multiplaction and division operations, perform string manipulations, and make some simple comparisons. Turing machines could be used as Language

Accepters for example: A string w is written on the tape, with blanks filling out the unused portions. The machine is started in the initial state q0 with the read-write head positioned on the leftmost symbol of w. If, after a sequence of moves, the Turing machine enters a final state and halts, then w is considered to be accepted.

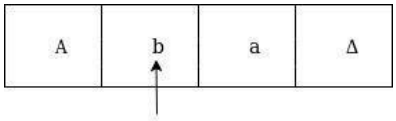
Lets take a look at the following example: let us consider a turing machine which accepts the language of aba over ∑ = {a, b}. if we suppose the string aba is placed like this:



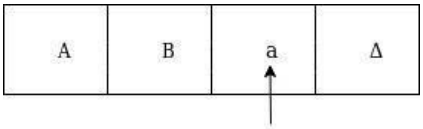
The tape head would read up to the ‘delta’ symbol , If the tape head is readout 'aba' string then Turing device will stop after reading the ‘delta symbol’. Now, we will see how this turing tool will work for the string of aba. To begin with the , state is q0 and head points to a as:



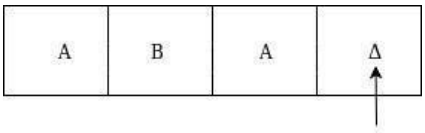
The move will be δ(q0, a) = δ(q1, A, R) which means it will move to the state of q1, replaced a by A and head will traverse towards the right as:



The move will be δ(q1, b) = δ(q2, B, R) which means it will traverse to state q2, replaced b by B and head will move to right as:



The move will be δ(q2, a) = δ(q3, A, R) which means it will go to state q3, replaced a by A and head will change its position towards right as:



The move δ(q3, Δ) = (q4, Δ, S) which means it will go to state q4 which is the HALT state and HALT state is always an accepted state for any kind of Turing Machine.

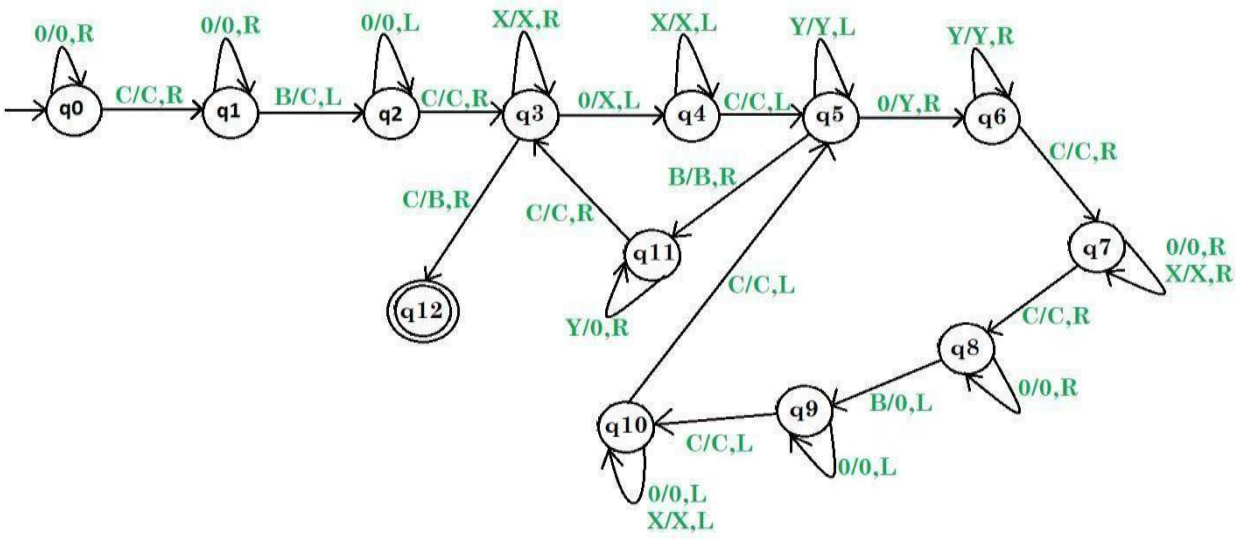


Figure 2 turing machine for multiplaction

Extra Knowledge:

History:

The turing machine was invented by the famous English mathematician in 1936 by Alan turing. Turing is considered to be the father of theortical computer science and artificial intelligence . Alan Turing described what became known as the "Turing Machine" in his 1936 document , "On Computable Numbers, with an application to the

Entscheidungsproblem," which was published in the Proceedings of the London Mathematical Society back in the late 1930s . The machine was not a piece of equipment but a concept ,however , its principles set the way for the development of digital computers in the 20th century. As mentioned previously all problems can be solved by digital computers can be solved by turing machine, however, there are some problems that are unsolvable by the turing machine such as the halting problem which is defined as : Given an arbitrary Turing machine M over alphabet = { a , b } , and an arbitrary string w over , does M halt when it is given w as an input ?

```python
x = input()
while x:
    pass
```

Figure 3 halt python code

Computing with Turing Machines:

Turing machines were intended to formalize the notion of computability in order to solve fundamental mathematical problems. Alonzo Church gave a different but logically equal formulation . Today, most computer scientists agree that

Turing's, or any other logically equal, formal notion captures all computable problems, viz. for any computable problem, there is a Turing machine which can compute it. It indicates that, if accepted, any problem not computable by a Turing device , it would be impossible to be computed using any finite method. Lets consider the following example: we will represent the number n as a block of n+1 copies of the symbol '1' on the tape. Thus we will represent the number

0 as a single '1' and the number 3 int he form of block of four '1's. This method is called as unary notation. We will also have to produce some assumptions about the configuration of the tape when the machine is started, and when it ends, in order to interpret the computation. We will assume that if the function to be computed requires n arguments, then the Turing machine will start with its head scanning the leftmost '1' of a sequence of n blocks of '1's. The blocks of '1's representing the arguments must be separated by a single occurrence of the symbol '0'. For example, to compute the sum 3+4 , a Turing machine will start in the configuration shown in Figure 4 :

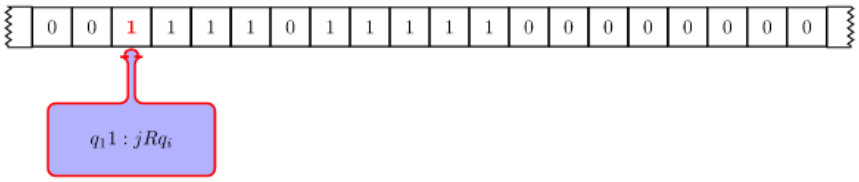


Figure 4 Initial configuration for a computation over two numbers n and m

The turing machine takes two arguments which symbolize the numbers to be added, beginning at the leftmost 1 of the first argument. The arguments are separated from each other by a single 0 as needed , and the first block contains four ‘1’s, representing the number 3, and the second contains five ‘1’s, representing the number 4 A machine must finish in standard configuration too. There must be a single block of symbols (a sequence of 1s which symbolize some number or a symbol representing another kind of output) and the machine must be scanning the leftmost symbol of that sequence. If the machine computes the function correctly then this block must represent the correct output

	0	1
q_1	/	$0\,R\,q_2$
q_2	$1\,L\,q_3$	$1\,R\,q_2$
q_3	$0\,R\,q_4$	$1\,L\,q_3$
q_4	/	$0\,R\,q_{\text{halt}}$

Table 1 Transition table for turing machine of adding 2 natural numbers

Table 1 displays the transition table of a Turing machine called TAdd2 which performs addition on 2 natural numbers *n* and *m*. We suppose that the machine starts in state q1 scanning the leftmost 1 of n+1 The idea of doing an addition with Turing machines when using unary representation is to shift the leftmost number *n* one square to the right. This is achieved by deleting the leftmost 1 of n+1 (this is done in state q1) and then setting the 0 between n+1and m+1 to 1 (state q2). We then have n+m+2 and so we still need to erase one additional 1. This is done by erasing the leftmost 1 (states q3 and q4). Figure 5 shows this computation for 3+4

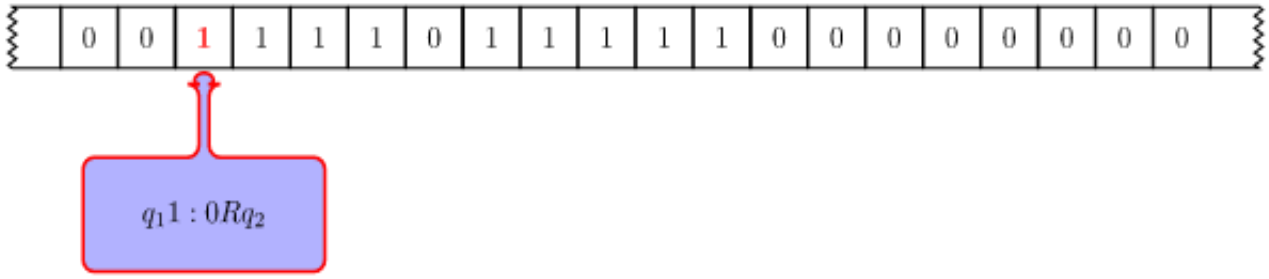


Figure 5

C++ Implementation: TASK 1: NFA_DFA

utility.h

```
#ifndef UTILITY_H_INCLUDED
#define UTILITY_H_INCLUDED

#include <iostream>
#include <queue>
#include <map>
#include <algorithm>
#include <iostream>
#include <set>


using namespace std;
```

```
struct FA
{
    string start;
    vector<string> epsilonStart;                          // start state
    vector<vector<string>> accept;                        // set of accepting states
    vector<string> lang;                                  // set of input symbols
    vector<vector<string>> states;                        // set of states
    map<pair<vector<string>, string>, vector<string>> transition_table; // transition
function; mapping state x lang => state
    map<string, vector<string>> epsilonClosure; // only used if FA is an epsilon-NFA
};

void printDFA(FA, bool);
FA convertNFAToDFA(FA);
FA convertEpsilonNFAToDFA(FA);

#endif // UTILITY_H_INCLUDED
```

utility.cpp

```
#include "utility.h"
#define EPSILON "#"
#define DEAD "$"

#define endl '\n'


bool find(vector<vector<string>> s1, vector<string> s2)
{
    string a;
    for (auto i : s2)
        a += i;
    sort(a.begin(), a.end());

    for (auto i : s1)
    {
        string b;
        for (auto j : i)
        {
            b += j;
        }
        sort(b.begin(), b.end());
        if (a == b)
            return true;
    }
    return false;
}

void unique(vector<string> &s)
{
    set<string> distinct(s.begin(), s.end());
    s.assign(distinct.begin(), distinct.end());
}

void setAcceptingStates(FA nfa, FA &dfa)
{
    vector<vector<string>> accepted;
    accepted.assign(nfa.accept.begin(), nfa.accept.end());
    for (auto &i : dfa.states)
    {
        for (auto j : i)
        {
            for (auto k : accepted)
            {
                for (auto m : k)
                {
                    if (j.find(m) != std::string::npos)
                    {
                        dfa.accept.push_back(i);
                        break;
                    }
                }
            }
        }
    }
```

```
}

void printDFA(FA dfa, bool isNFA)
{

    // print Start State
    if (isNFA)
        cout << "Start: " << dfa.start << endl;
    else
    {
        cout << "Start: ";
        for (auto s : dfa.epsilonStart)
            cout << s << " ";
        cout << endl;
    }

    // print accepted states
    cout << "Accepted States: ";
    for (auto i : dfa.accept)
    {
        cout << "[";
        for (auto j : i)
            cout << j << ",";
        cout << "\b], ";
    }
    // print input symbols
    cout << "\nLang: ";
    for (int i = 0; i < dfa.lang.size(); i++)
        cout << dfa.lang[i] << " ";
    cout << endl;

    // print states
    cout << "States: ";
    for (auto i : dfa.states)
    {
        cout << "[";
        for (auto j : i)
            cout << j << ",";
        cout << "\b], ";
    }
    cout << "\b\b" << endl;

    // print transition table
    cout << "Transition table:" << endl;
    for (auto pair : dfa.transition_table)
    {
        cout << "[";
        for (auto src : pair.first.first)
            cout << src << ", ";
        cout << "], ";
        cout << pair.first.second
             << " -> "
             << "[";
        for (auto dest : pair.second)
            cout << dest << ", ";
        cout << "]" << endl;
```

```
    }
    // print epsilon closure
    if (!isNFA)
    {
        cout << "Epsilon Closure:" << endl;
        for (auto pair : dfa.epsilonClosure)
        {
            cout << pair.first
                 << " -> ";
            for (auto val : pair.second)
                cout << val
                     << " ";
            cout << endl;
        }
    }
}

FA convertNFAToDFA(FA nfa)
{

    FA dfa;
    dfa.start.assign(nfa.start.begin(), nfa.start.end());
    dfa.lang.assign(nfa.lang.begin(), nfa.lang.end());

    vector<string> start{nfa.start};
    dfa.states.push_back(start);

    queue<vector<string>> newStates;
    newStates.push(start);
    while (!newStates.empty())
    {
        vector<string> curState;
        for (auto i : newStates.front())
            curState.push_back(i);
        int stateLen = curState.size();
        int langLen = nfa.lang.size();
        vector<string> newState;
        for (int i = 0; i < langLen; ++i)
        {

            newState.clear();
            for (int j = 0; j < stateLen; ++j)
            {
                vector<string> state{curState[j]};
                if (nfa.transition_table.find({state, nfa.lang[i]}) !=
nfa.transition_table.end())
                {
                    for (auto i : nfa.transition_table[{state, nfa.lang[i]}])
                        newState.push_back(i);
                }
            }

            if (newState.empty())
                newState.push_back(DEAD);

            unique(newState);
```

```
            dfa.transition_table[{curState, nfa.lang[i]}] = newState;
            bool isFound = find(dfa.states, newState);
            if (!isFound)
            {
                newStates.push(newState);
                dfa.states.push_back(newState);
            }
        }
        newStates.pop();
    }
    setAcceptingStates(nfa, dfa);

    // Dummy state for initial node
    vector<string> emptyFrom;
    emptyFrom.push_back("&");
    string emptyOn = "&";
    dfa.transition_table[{{emptyFrom},emptyOn}].push_back(dfa.start);
    dfa.states.push_back(emptyFrom);
    return dfa;
}

void buildEpsilonClosure(FA &epsilonNFA)
{
    for (auto states : epsilonNFA.states)
    {
        string s;
        s += states[0];
        if (epsilonNFA.transition_table.find({states, EPSILON}) ==
epsilonNFA.transition_table.end())
        {
            epsilonNFA.epsilonClosure[s].push_back(s);
            continue;
        }

        queue<vector<string>> epsilonClosure;
        epsilonClosure.push(states);
        vector<string> newTransitions;
        for (auto s : states)
            newTransitions.push_back(s);
        while (!epsilonClosure.empty())
        {
            for (auto st : epsilonClosure.front())
            {
                vector<string> cl;
                cl.push_back(st);
                if (epsilonNFA.transition_table.find({cl, EPSILON}) !=
epsilonNFA.transition_table.end())
                {
                    epsilonClosure.push(epsilonNFA.transition_table[{cl, EPSILON}]);
                    for (auto s : epsilonNFA.transition_table[{cl, EPSILON}])
                        newTransitions.push_back(s);
                }
            }

            epsilonClosure.pop();
        }
```

```
        unique(newTransitions);
        epsilonNFA.epsilonClosure[s].assign(newTransitions.begin(),newTransitions.end());
    }
}

FA convertEpsilonNFAToDFA(FA epsilonNFA)
{
    FA dfa;
    dfa.lang.assign(epsilonNFA.lang.begin(), epsilonNFA.lang.end());

    buildEpsilonClosure(epsilonNFA);
    dfa.epsilonClosure = epsilonNFA.epsilonClosure;

    vector<string> start{epsilonNFA.epsilonClosure[epsilonNFA.start]};
    dfa.epsilonStart.assign(start.begin(), start.end());
    dfa.states.push_back(start);

    queue<vector<string>> newStates;
    newStates.push(start);

    while (!newStates.empty())
    {
        vector<string> curState;
        for (auto i : newStates.front())
            curState.push_back(i);
        int stateLen = curState.size();
        int langLen = epsilonNFA.lang.size();
        vector<string> newState;
        for (int i = 0; i < langLen; ++i)
        {
            newState.clear();
            for (int j = 0; j < stateLen; ++j)
            {
                vector<string> state{curState[j]};
                if (epsilonNFA.transition_table.find({state, epsilonNFA.lang[i]}) !=
epsilonNFA.transition_table.end())
                {
                    for (auto it : epsilonNFA.transition_table[{state,
epsilonNFA.lang[i]}])
                        newState.push_back(it);
                    queue<string> epsilonClosure;
                    for (auto it : newState)
                        epsilonClosure.push(it);
                    while (!epsilonClosure.empty())
                    {
                        for(auto k: dfa.epsilonClosure[epsilonClosure.front()])
                            newState.push_back(k);
                        epsilonClosure.pop();
                    }
                }
            }

            if (newState.empty())
                newState.push_back(DEAD);
```

```
            unique(newState);
            dfa.transition_table[{curState, epsilonNFA.lang[i]}] = newState;
            bool isFound = find(dfa.states, newState);
            if (!isFound)
            {
                newStates.push(newState);
                dfa.states.push_back(newState);
            }
        }
        newStates.pop();
    }
    setAcceptingStates(epsilonNFA, dfa);
    // Dummy state for initial node
    vector<string> emptyFrom;
    emptyFrom.push_back("&");
    string emptyOn = "&";
    dfa.transition_table[{{emptyFrom},emptyOn}].push_back(dfa.start);
    dfa.states.push_back(emptyFrom);


    return dfa;
}
```

C++ Implementation: TASK 2: CFG_PDA

pda.cpp

```
#include "pda.h"
#include "graph.h"
#include <QDebug>

#define EPSILON "#"
#define DOLLAR "$"
#define endl '\n'

void uniques(vector<string> &s)
{
    set<string> distinct(s.begin(), s.end());
    s.assign(distinct.begin(), distinct.end());
}

bool isNonTerminal(string symbol, PDA pda)
{
    vector<string>::iterator it;
    it = find(pda.nonTerminals.begin(), pda.nonTerminals.end(), symbol);
    return it != pda.nonTerminals.end();
}

void pushMandatoryStates(PDA &pda)
{

    pda.transitionTable.insert({{{"q0"}, {EPSILON, EPSILON, DOLLAR}}, {"q1"}});
    pda.transitionTable.insert({{{"q1"}, {EPSILON, EPSILON, pda.start}}, {"qloop"}});
    pda.transitionTable.insert({{{"qloop"}, {EPSILON, DOLLAR, EPSILON}}, {"qAccept"}});
    pda.states.push_back("q0");
    pda.states.push_back("q1");
    pda.states.push_back("qloop");
    pda.states.push_back("qAccept");
}
```

```
void convertCFGToPDA(PDA &pda)
{
    pushMandatoryStates(pda);

    // Handling terminals
    for (auto terminal : pda.terminals)
    {
        vector<string> trans = {terminal, terminal, EPSILON};
        pda.transitionTable[{{"qloop"}, trans}].push_back({"qloop"});
    }

    int stateNum = 2;

    // Handling non-terminals
    for (auto nonTerminal : pda.nonTerminals)
    {
        string prevState = "qloop";
        string curState = "q" + to_string(stateNum++);
        vector<string> trans = {EPSILON,nonTerminal,EPSILON};
        pda.transitionTable[{{prevState},trans}].push_back(curState);
        for (auto production : pda.productionRules[nonTerminal])
        {
            string innerPrevState = curState;
            string innerCurState = "q" + to_string(stateNum++);
            string reversedRule = string(production.rbegin(), production.rend());
            int len = reversedRule.size();
            for (int i = 0; i < len - 1; i++)
            {
                if(i==0) pda.states.push_back(curState);
                string symbol;
                symbol += reversedRule[i];
                vector<string> trans = {EPSILON, EPSILON, symbol};
                pda.transitionTable[{{innerPrevState},trans}].push_back(innerCurState);
                innerPrevState = innerCurState;
                pda.states.push_back(innerCurState);
                innerCurState = "q" + to_string(stateNum++);
```

```
                if(i == len - 2) stateNum--;
            }
            string symbol;
            symbol += reversedRule[len - 1];
            vector<string> trans = {EPSILON, EPSILON, symbol};
            pda.transitionTable[{{innerPrevState},trans}].push_back({"qloop"});
        }
    }

    uniques(pda.states);

}

void printPDA(PDA pda)
{
    for (auto pair : pda.transitionTable)
    {
        for(auto s: pair.first.first)
            qDebug() << s;
        qDebug() << ", {"
             << pair.first.second[0]
             << ", "
             << pair.first.second[1]
             << ", "
             << pair.first.second[2]
             << "} -> "
             << "[";
        for(auto state: pair.second)
            qDebug() << state << ",";
        qDebug() << "]";
        qDebug() << endl;
    }
    qDebug() << "STATES" << endl;
    for(auto state: pda.states)
        qDebug() << state << " ";

}
```

References:

https://www.cs.odu.edu/~zeil/cs390/latest/Public/pda-jflap/index.html

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

https://cstheory.stackexchange.com/questions/9673/why-is-non-determinism-push-downautomata-necessary

https://www.geeksforgeeks.org/introduction-of-pushdownautomata/

https://www.javatpoint.com/automata-turing-machine https://en.wikipedia.org/wiki/Turing_machine

https://plato.stanford.edu/entries/turingmachine/#CompTuriMach

https://www.slideshare.net/HimanshuSirohi6/turing-machine-132443033