Introduction to Automata Theory

Automata theory is basically about the study of different mechanisms for generation and recognition of languages. Automata theory is basically for the study of different types of grammars and automata. A *grammar* is a mechanism for the generation of sentences in a language. *Automata* is a mechanism for recognition of languages. Automata theory is mainly for the study of different kinds of automata as language recognizers and their relationship with grammars.

In theoretical computer science, **automata theory** is the study of abstract machines and problems they are able to solve. Automata theory is closely related to formal language theory as the automata are often classified by the class of formal languages they are able to recognize.

An automaton is a mathematical model for a finite state machine (FSM). A FSM is a machine that, given an input of symbols, "**jumps**" through a series of states according to a transition function (which can be expressed as a table). In the common "Mealy" variety of FSMs, this transition function tells the automaton which state to go to next given a current state and a current symbol.

The input is *read* symbol by symbol, until it is consumed completely (think of it as a tape with a word written on it, that is read by a reading head of the automaton; the head moves forward over the tape, reading one symbol at a time). Once the input is depleted, the automaton is said to have *stopped*.

Depending on the state in which the automaton stops, it's said that the automaton either *accepts* or *rejects* the input. If it landed in an *accept state*, then the automaton *accepts* the word. If, on the other hand, it lands on a *reject state*, the word is *rejected*. The set of all the words accepted by an automaton is called the *language accepted by the automaton*.

Automata play a major role in compiler design and parsing.

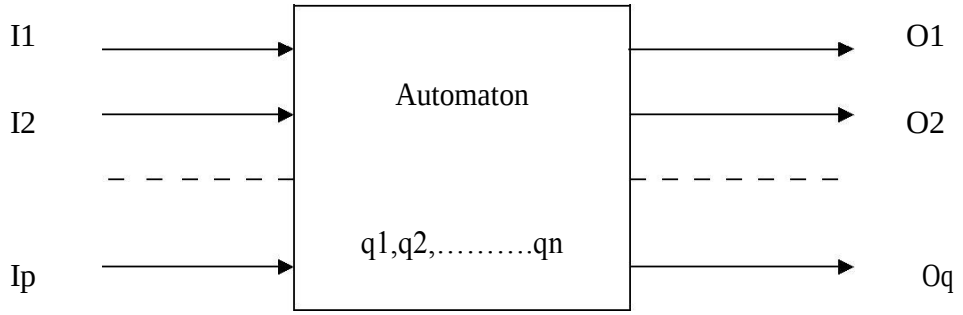
Finite Automata is the simplest one of different classes of automata. Mainly there are 3 variants of finite automata. They are:

Deterministic Finite Automata Non-
deterministic Finite Automata
Non-deterministic Finite Automata with Є-transition.

Here we define the acceptability of strings by finite automata.

Description of Automaton

An automaton can be defined in an abstract way by the following figure.



Model of a discrete automaton

i) Input: - At each of the discrete instants of time t1,t2,…..input values I1,I2…… each of which can take a finite number of fixed values from the input alphabet ∑, are applied to the input side of the model.

ii) Output : - O1,O2….are the outputs of the model, each of which can take finite numbers of fixed values from an output O.

iii) States : - At any instant of time the automaton can be in one of the states q1,q2…..qn

iv) State relation : - The next state of an automaton at any instant of time is determined by the present state and the present input. ie, by the transition function.

v) Output relation : - Output is related to either state only or both the input and the state. It should be noted that at any instant of time the automaton is in some state. On 'reading' an input symbol, the automaton moves to a next state which is given by the state relation.

An automaton in which the output depends only on the input is called an automaton without a memory. An automaton in which the output depends on the states also is called automaton with a finite memory. An automaton in which the output depends only on the states of the machine is called a *Moore Machine.* An automaton in which the output depends on the state and the input at any instant of time is called a *Mealy machine.*

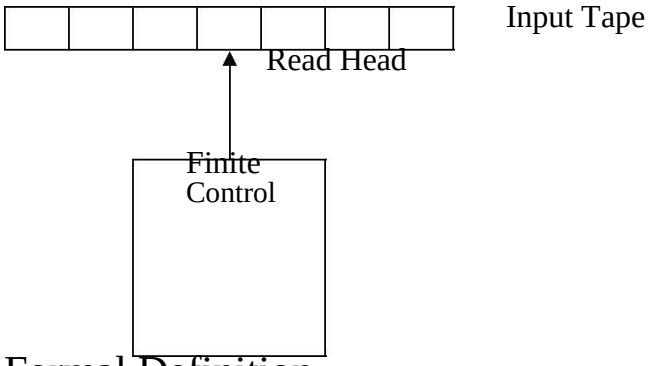
Definition of a finite automaton

Basic model of finite automata consists of :

(i) An *input tape* divided into cells, each cell can hold a symbol

(ii) A *read head* which can read one symbol at a time from a finite alphabet

(iii) A *finite control* which works within a finite set of states. At each step, it changes its state depending on the current state and input read. Its change of state is specified by a transition function. It accepts the input if it is in a set of accepting states.



a) Finite Automata – Formal Definition

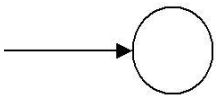
A finite automaton can be represented by a 5-tuple $(Q,\sum,\delta,q_0,F)$ where Q – finite set of *internal states*
$\sum$ - finite set of symbols called *input alphabet*
δ – *transition function*
$q_0 \in Q$ – start state or *initial state*
$F \in Q$ – set of accepting states or *final states*

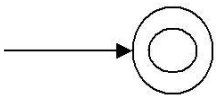
Transition function describes the change of states during the transition. This mapping is usually represented by *Transition diagram or Transition table.*

Transition diagrams and Transition Systems

A transition graph or a transition system is a finite directed labeled graph in which each vertex (node) represents a state and the directed edges indicate the transition of a state and the edges are labelled with input. A transition graph contains:

(i) A *finite set of states*, one of which are designated as start state and some of which are designated as final states.


Start state


Final state

(ii) An *alphabet* □ of possible input letters from which input strings are formed.
(iii) A *finite set of transitions* that show, how to go from some states to some other states.
So a transition system is a 5-tuple $(Q,\sum,\delta,q_0,F)$

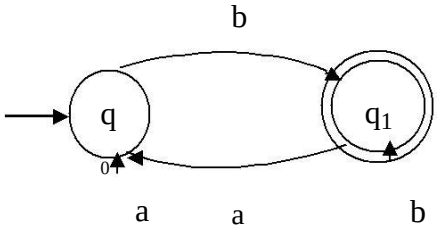
If $\delta(q_i,a)=q_j$, there is an edge labeled by 'a' from q_i to q_j . A transition system accepts a string 'w' in $\sum^*$ if

1) there exists a path which originates from some initial state, goes along the arrows, and terminates at some final state.
2) the path value obtained by concatenation of all edge-labels of the path is equal to 'w'.

Transition Table

The description of the automation can be given in the form of transition table also, in which we tabulate the details of the transitions defined be the automaton from one state to another.

eg: Draw the transition diagram and transition table for accepting the language
L={all words ending in 'b' over (a,b)}



Transition Diagram

δ / □	a	b
q0	q0	q1
►* q1	q1	q0

Transition Table

b) Deterministic and Non deterministic Finite Automata

Deterministic fin ite automata (DFA)

Each state of an automaton of this kind has a transition for every symbol in the alphabet.

Deterministic Finite Automata can be defined as M=(Q,∑,δ,q0,F)
where Q is the set of states

∑ is the input symbols

δ is the transition function Q x ∑ Q q0 is

the start state

F is the final state

Nondeterministic finite automata (NFA)

States of an automaton of this kind may or may not have a transition for each symbol in the alphabet, or can even have multiple transitions for a symbol. The automaton accepts a word if there exists at least one path from q_0 to a state in F *labeled* with the input word. If a transition is *undefined*, so that the automaton does not know how to keep on reading the input, the word is rejected.
NFA is equivalent to the DFA.

Non-Deterministic Finite Automata also can be defined as M=(Q,∑,δ,q0,F)
where Q is the set of states

 is the input symbols

δ is the transition function Q x $\sum 2^Q$ q0 is

the start state

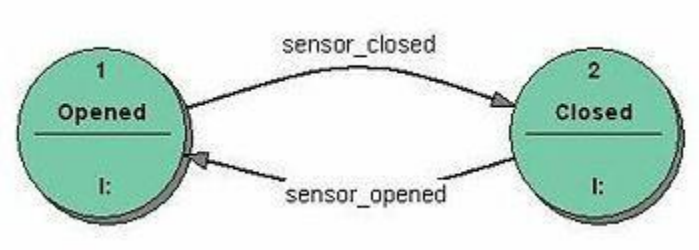
F is the final state

Nondeterministic finite automata, with ε transitions (FND-ε or ε-NFA)

Besides of being able to jump to more (or none) states with any symbol, these can jump on no symbol at all. That is, if a state has transitions labeled with ε, then the NFA can *be* in any of the states reached by the ε-transitions, directly or through other states with ε-transitions. The set of states that can be reached by this method from a state q, is called the ε-closure of q.

Moore machine

The FSM uses only entry actions, i.e. output depends only on the state. The advantage of the Moore model is a simplification of the behaviour. The example in figure 1 shows a Moore FSM of an elevator door. The state machine recognizes two commands: "command_open" and "command_close" which trigger state changes. The entry action (E:) in state "Opening" starts a motor opening the door, the entry action in state "Closing" starts a motor in the other direction closing the door. States "Opened" and "Closed" don't perform any actions. They signal to the outside world (e.g. to other state machines) the situation: "door is open" or "door is closed".



Transducer FSM: Mealy model example

Mealy machine

The FSM uses only input actions, i.e. output depends on input and state. The use of a Mealy FSM leads often to a reduction of the number of states. The example in figure 4 shows a Mealy FSM implementing the same behaviour as in the Moore example (the behaviour depends on the implemented FSM execution model and will work e.g. for virtual FSM but not for event driven FSM). There are two input actions (I:): "start motor to close the door if command_close arrives" and "start motor in the other direction to open the door if command_open arrives".

In practice mixed models are often used.

Extended Transition function of DFA

The language of a DFA is the set of labels along the paths that lead from the start state to any accepting state. Now we extended the transition function that describes what happens when we start in any state and follow any sequence of inputs. If δ is our transition function, the extended function constructed from δ will be called δ.

The extended transition function is the function that takes a state 'q' and a string 'w' and returns a state 'p', the state that automation reaches when starting in state 'q' and processing the sequence of inputs 'w'. We define δ by induction on the length of the input string as follows:

Basis : δ(q,Є)=q. ie, if we are in state q and read no input, then we are still in the state q. **Induction :** Suppose 'w' is a string of the form xa, that is a is the last symbol of w and x is the substring of w, consisting of all except the last symbol 'a'. For example,

w=1101 is broken into x=110 and a=1 then δ(q,w) = δ(δ(q,x),a)

ie, to compute δ(q,w), first compute δ(q,x), the state that the automation is in after processing all but the last symbol of w. Suppose this state is P, that is δ(q,x)=P. Then δ(q,w) is what we get by making a transition from state P on input a, the last symbol of w.

δ(q,w)= δ(P,a)

Extended Transition function of NFA

As for DFA's, we need to extended the transition function δ of an NFA to a function δ' that takes a state and string and return the set of states.

Basis : δ(q,Є)={q}. That is without reading any input symbols, we are only in the same state.

Induction : Suppose w is of the form w=xa, where a is the last symbol and x is the substring containing rest of w. Let us suppose that

$\delta(q,x)=\{p1,p2\ldots.pk\}$

Let $\cup\ \delta(pi,a)=\{r1,r2\ldots\ldots\ldots rm\}$ then

$\delta(q,w)=\{r1,r2\ldots\ldots..rm\}$. Less formally, we compute $\delta(q,w)$ by first computing $\delta(q,x)$, and by then following any transition from any of these states that is labeled a.

Language acceptability by Finite Automata

Suppose a1,a2,a3…….a_n is a sequence of input symbols, q0,q1,q2…….q_n are set of states where q_0 is start state and q_n is final state and transition function processed as
$\delta(q0,a1)=q1$
$\delta(q1,a2)=q2$
$\delta(q2,a3)=q3$
....
...
$\delta(q_{n-1},a_n)=q_n$

Input a1,a2,a3……..a_n is said to be 'accepted ' since q_n is a member of the final state, and if not then it is 'rejected'.

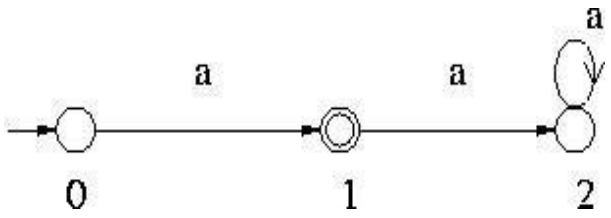
Language accepted by DFA 'M' written as
$L(M)=\{w/\ \delta(q_0,w)=q_f$ for some q_f in F$\}$

Examples of DFA

Example 1: Q = { 0, 1, 2 }, Σ = { a }, A = { 1 }, the initial state is 0 and δ is as shown in the following table.

State (q)	Input (a)	Next State (δ(q, a))
0	a	1
1	a	2
2	a	2

A state transition diagram for this DFA is given below.



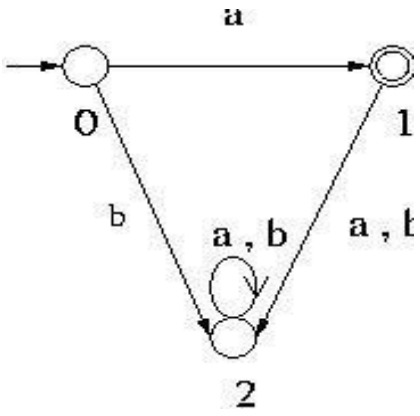
If the alphabet Σ of the Example 1 is changed to { a, b } instead of { a }, then we need a DFA such as shown in the following example to accept the same string a. It is a little more complex DFA.

Example 2: Q = { 0, 1, 2 }, Σ = { a, b }, A = { 1 }, the initial state is 0 and δ is as shown in the following table.

State (q)	Input (a)	Next State (δ (q, a))
0	a	1
0	b	2
1	a	2
1	b	2
2	a	2
2	b	2

Note that for each state there are two rows in the table for δ corresponding to the symbols a and b, while in the Example 1 there is only one row for each state.

A state transition diagram for this DFA is given below.

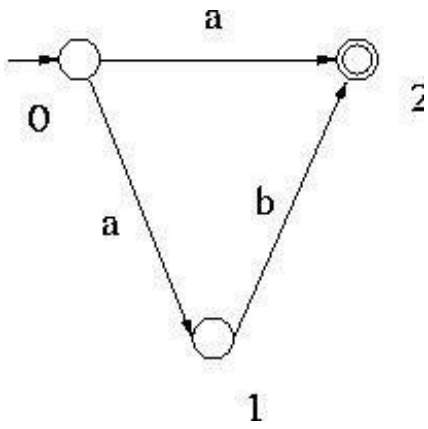


State (q)	Input (a)	Next State (δ (q, a))
0	a	{ 1 , 2 }
0	b	$\emptyset$
1	a	$\emptyset$
1	b	{ 2 }
2	a	$\emptyset$
2	b	$\emptyset$

Examples of NFA

Example 1: Q = { 0, 1, 2 }, Σ { a, b }, A = { 2 }, the initial state is 0 and is as shown in the following table.

Note that for each state there are two rows in the table for ε corresponding to the symbols a and b, while in the Example 1 there is only one row for each state.
A state transition diagram for this finite automaton is given below.



(i) Conversion of NFA to DFA

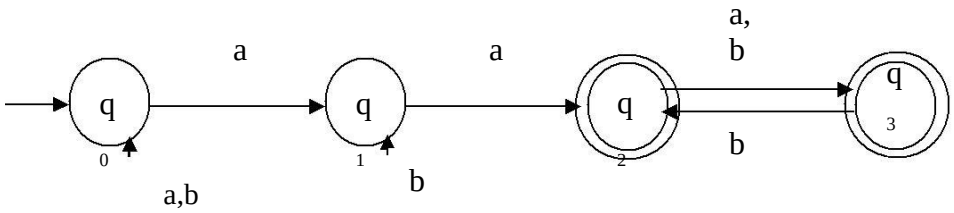
The conversion of a DFA equivalent to an NDFA involves three steps.

Step 1: Convert the given transition system into state transition table where each state corresponds to a row and each input symbol corresponds to a column.

Step 2: Construct the successor table that lists subsets of states reachable from the set of initial states.

Step 3: The transition graph given by the successor table is the required deterministic system. The final states contain some final state of NDFA.

eg: Convert the following NFA to DFA



δ(q0,a)={q0,q1} new state
δ(q0,b)={q0} old state

δ({q0,q1},a)= δ (q0,a)Uδ(q1,a)
={q0,q1}U{q2}

δ({q0,q1 },b)= δ (q0,b)Uδ(q1,b)
={q0}U{q1}
={q0,q1} old state

Similarly,
δ({q0,q1,q2},a)= δ(q0,a) U δ(q1,a) U δ(q2,a))
= {q0,q1,q2,q3} new state

δ({q0,q1,q2},b)= {q0,q1,q3} new state

δ({q0,q1,q2,q3},a)= {q0,q1,q2,q3} old state

δ({q0,q1,q2,q3},b)= {q0,q1,q3,q2} old state

δ({q0,q1,q3},a)= {q0,q1,q2,q3} old state

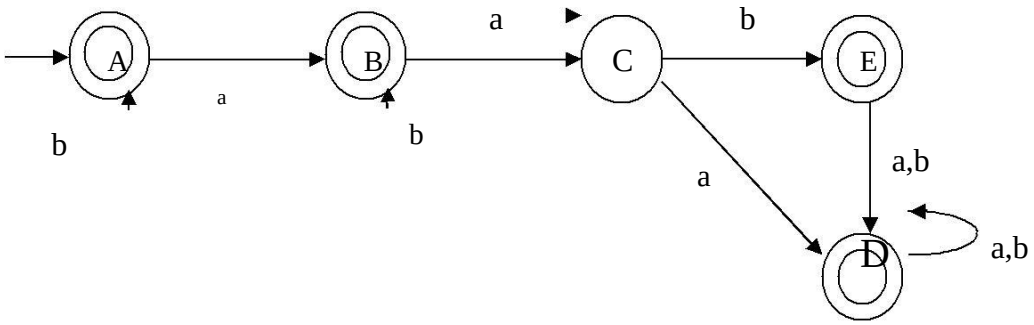
δ({q0,q1,q3},b)= {q0,q1,q3,q2} old state

Now we can draw the transition table for DFA.

	δ/□	a	b
A	{q0}	{ q0,q1}	{ q0 }

B	{q0,q1}	{ q0,q1,q2 }	{q0,q1}
C	{q0,q1,q2}	{q0,q1,q2,q3}	{q0,q1,q3}
D	*{q0,q1,q2,q3}	{q0,q1,q2,q3}	{q0,q1,q2,q3}
E	*{q0,q1,q3}	{q0,q1,q2,q3}	{q0,q1,q2,q3}

Now let us draw the transition diagram



c) Finite Automaton with Є-transition

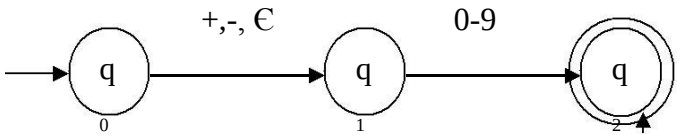
Besides of being able to jump to more (or none) states with any symbol, these can jump on no symbol at all. That is, if a state has transitions labeled with ε, then the NFA can *be* in any of the states reached by the ε-transitions, directly or through other states with ε-transitions. The set of states that can be reached by this method from a state q, is called the ε-closure of q.

An NFA can be modified to permit transition without input symbols, along with one or more transition on input symbols; we get a "NFA with Є transition". Since the transition is made without symbols the transition is called as "Є-transition". These transitions can occur when no input is applied. But it is possible to convert a transition system with Є-transition into an equivalent transition system without Є-moves.

It is to be noted that Є is not a symbol to appear on the tape. ie, Є-transition means a transition without scanning the symbol. ie, not moving the read head.

eg: δ(q, Є)=p means that from the state 'q', it can jump to 'p' without moving the read head. ie, it can be in 'p' or 'q'. Thus it introduces a hidden non-determinism. Є-transitions are useful in specifying optional items in a string.

eg: In a typical programming language, while specifying a numeric constant, the sign is optional.



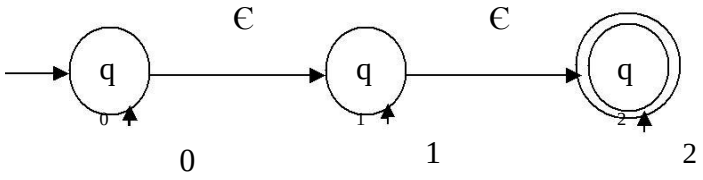
Transition Diagram

0-9

Formal Definition of Є-NFA

Є-NFA is defined as M=(Q,□ ,δ,q0,F) where Q,□ ,q0 and F are same as in NFA but δ includes Є moves and is defined as δ: Qx□ U{ Є } 2^Q.

To define the behavior of Є-NFA on strings, we require a function called Є-closure, which is defined as Є-closure(q) is the set of all states reachable from q using Є-transitions.
eg:



Є-closure (q0)={q0,q1,q2}
Є-closure (q1)={q1,q2} Є-
closure (q2)={q2}

(ii)Conversion of NFA with Є-transition to NFA with out Є-transition (Eliminating Є- transition)

Let M=(Q,□ ,δ,q0,F) be an Є-NFA. There are some steps for the conversion of NFA with Є-transition to NFA with out Є-transition.

Step 1: Find the states of NFA without Є-transition including initial states and final states.
Step 2: There will be same number of states. The initial state of NFA without Є-transition will be Є-closure of initial state of Є- NFA.

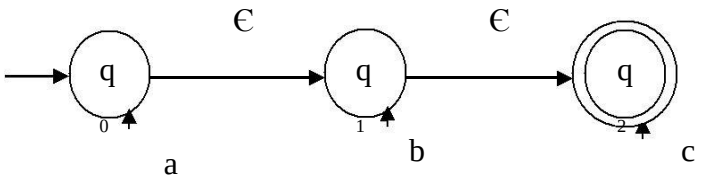
ie, Є-closure (q0)={q0,q1,q2} initial state for NFA without Є-transition, rest of the states are

Є-closure(q1)={q1,q2}
Є-closure(q2)={q2}

Step 3: The final states of NFA without Є-transitions are all those new states which contains final state of Є-NFA as member.

Step 4: Now find out δ' to find out the transitions for NFA without Є-transition. Ignore Φ entries and Є-transitions column.

eg: Convert the following NFA with Є-transition to NFA without Є-transition



From the above transition diagram,

δ/□	a	b	c	Є
q0	{ q0 }	{ Φ }	{ Φ }	{q1}
q1	{ Φ }	{q1}	{ Φ }	{q2}
* q2	{ Φ }	{ Φ }	{q2}	{ Φ }

Є-closure (q0)={q0,q1,q2}=qa *new initial state* for NFA without Є transition, rest of the states are
Є-closure(q1)={q1,q2}=qb *new state*
Є-closure(q2)={q2}=qc *new state*

δ'({q0,q1,q2},a)= δ'(qa,a)=Є-closure (δ(q0,q1,q2),a)
= Є-closure (δ(q0,a) U δ(q1,a) U δ(q2,a))
= Є-closure(q0 U Φ U Φ)
= Є-closure(q0)
={q0,q1,q2}

δ'({q0,q1,q2},b)= δ'(qa,b)=Є-closure (δ(q0,q1,q2),b)
= Є-closure (δ(q0,b) U δ(q1,b) U δ(q2,b))
= Є-closure(Φ U {q1} U Φ)
= Є-closure(q1)
={q1,q2}

δ'({q0,q1,q2},c)= δ'(qa,c)=Є-closure (δ(q0,q1,q2),c)
= Є-closure (δ(q0,c) U δ(q1,c) U δ(q2,c))
= Є-closure(Φ U Φ U q2)
= Є-closure(q2)
={q2}

.

δ'({q1,q2},a)= δ'(qb,a)=Є-closure (δ(q1,q2),a)
= Є-closure (δ(q1,a) U δ(q2,a))
= Є-closure(Φ U Φ)
= Є-closure(Φ)
= Φ

δ'({q1,q2},b)= δ'(qb,b)=Є-closure (δ(q1,q2),b)
= Є-closure (δ(q1,b) U δ(q2,b))
= Є-closure(q1 U Φ)
= Є-closure(q1)
= {q1,q2}

δ'({q1,q2},c)= δ'(qb,c)=Є-closure (δ(q1,q2),c)
= Є-closure (δ(q1,c) U δ(q2,c))
= Є-closure(Φ U q2)
= Є-closure(q2)
= {q2}

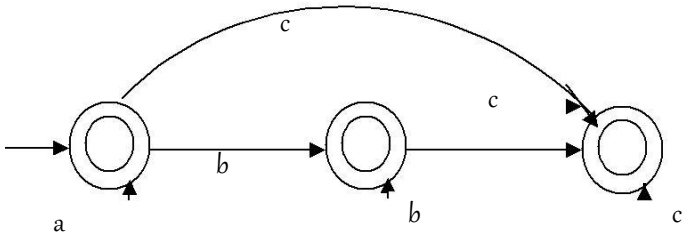
δ'({q2},a)= δ'(qc,a)=Є-closure (δ(q2),a)
= Φ

δ'({q2},b)= δ'(qc,b)=Є-closure (δ(q2),b)
= Φ

δ'({q2},c)= δ'(qc,c)=Є-closure (δ(q2),c)
= Є-closure(q2)
= {q2}

Now we can draw the transition table for NFA without Є-transition.

	δ/□	a	b	c
A	{q0,q1,q2} *	{ q0,q1,q2}	{ q1,q2 }	{ q2 }
B	* {q1,q2}	{ Φ }	{q1,q2}	{ q2 }
C	* {q2}	{ Φ }	{ Φ }	{q2}



Regular operations, Regular expressions and Regular languages

a) Regular expressions

Regular expressions are precisely defined by a set of rules. For each rule, we describe the corresponding language. The languages accepted by finite automata are easily described by simple expressions called regular expressions. Every regular expression specifies a language. Regular expression is a declarative way to express the strings, we want to accept.

Definition of Regular expression

The set of regular expression is defined by the following rules:

1) Every letter of Σ can be made into a regular expression, null string, ϵ itself is a regular expression.

2) If r1 and r2 are regular expression, then

i) (r1) ii) r1r2 iii) r1+r2 iv) r1* v) r1+

are also regular expression.

b) Regular languages

Regular languages are those that can be generated by applying certain standard operations like union, concatenation and closure, a finite number of times. They can be recognized by finite automata.

Let Σ be an alphabet. The regular expressions over Σ and the sets that they denote are defined recursively as follows.

1) Φ is a regular expression and denotes the empty set.
2) ε is a regular expression and denotes the set $\{\varepsilon\}$.
3) For each 'a' in Σ, a is a regular expression and denotes the set {a}.

These are known as simple regular languages. Regular language over an alphabet Σ is one that can be obtained from these basic (simple) languages using the operations of union, concatenation and closure, a finite number of times.

c) Regular operations

Mainly there are 3 operations on regular expressions. They are union, concatenation, and kleene closure operation.

If L1 and L2 are any elements of set R of regular languages over $\sum$ and r1 and r2 are the corresponding regular expressions,

i) Union – (L1 U L2) corresponding regular expression is (r1+r2)
ii) Concatenation – (L1.L2) corresponding regular expression is (r1. r2)
iii) Kleene closure – (L1*) corresponding regular expression is (r1)*

Algebra of Regular Expression

Regular expressions satisfy the following algebraic identities. These identities help us in simplifying regular expressions.

(1) Identity Law

$\in.R=R.\in=R$

$\Phi+R=R+\Phi=R$

(2) Idempotent Law

$R+R=R$

$(R^*)=R^*$

(3) Distributive Law

$A.(B+C)=A.B+A.C$

(4) Associative Law

$A.(B.C)=(A.B).C$

$A+(B+C)=(A+B)+C$

(5) Annihilation

$\Phi.R=R.\Phi=\Phi$

Example 1: It is easy to see that the RE (0+1)*(0+11) represents the language of all strings over {0,1} which are either ended with 0 or 11.

Example 2: Consider the language of strings over {0,1} containing two or more 1's.

Solution : There must be at least two 1's in the RE somewhere and what comes before, between, and after is completely arbitrary. Hence we can write the RE as (0+1)*1(0+1)*1(0+1)*. But following two REs also represent the same language, each ensuring presence of least two 1's somewhere in the string

i) 0*10*1(0+1)*

ii) (0+1)*10*10*

iii) Regular Expression to Finite state Automata :

Lemma : If *L*(*r*) is a language described by the RE *r*, then it is regular i.e. there is a FA such that *L*(*M*) ≅ *L*(*r*).

Proof : To prove the lemma, we apply structured index on the expression *r*. First, we show how to construct FA for the basis elements: ϕ , $\in$ and for any $a \in \Sigma$. Then we show how to combine these Finite Automata into Complex Automata that accept the Union, Concatenation, Kleen Closure of the languages accepted by the original smaller automata.

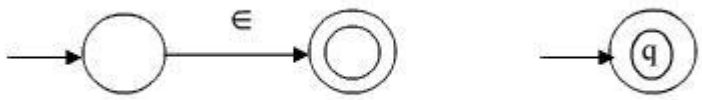
Use of NFAs is helpful in the case i.e. we construct NFAs for every REs which are represented by transition diagram only.

Basis :

- Case (i) **:** $r = \phi$. Then $L(r) = \phi$. Then $L(r) = \phi$ and the following NFA *N* recognizes *L*(*r*). Formally $N = (Q, \{q\}, \Sigma, \delta, q, F, \phi)$, where *Q* = {*q*} and $\delta(q,a) = \phi \; \forall \; a \in S, \; F = \phi$.

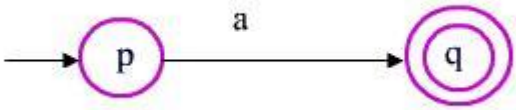
- Case (ii) **:** $r = \in$. $L(r) = \{\in\}$, and the following NFA N accepts L(r). Formally $N = (\{q\}, \Sigma, \delta, q, \{q\})$ where $\delta(q,a) = \phi \quad \forall a \in \Sigma$.

.



Since the start state is also the accept step, and there is no any transition defined, it will accept the only string $\in$ and nothing else.

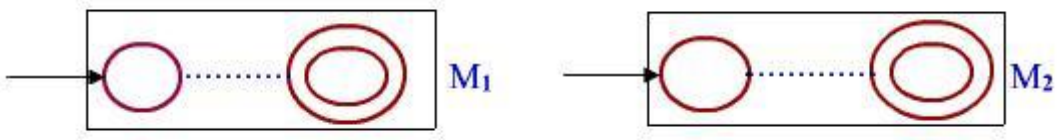
- Case (iii) : *r* = *a* for some $a \in \Sigma$. Then *L*(*r*) = {*a*}, and the following NFA *N* accepts *L*(*r*).



Formally, $N = (\{p,q\}, \Sigma, \delta, p, \{q\})$ where $\delta(p,q) = \{q\}$, $\delta(s, b) = \{\phi\}$ for $s \neq p$ or $b \neq a$

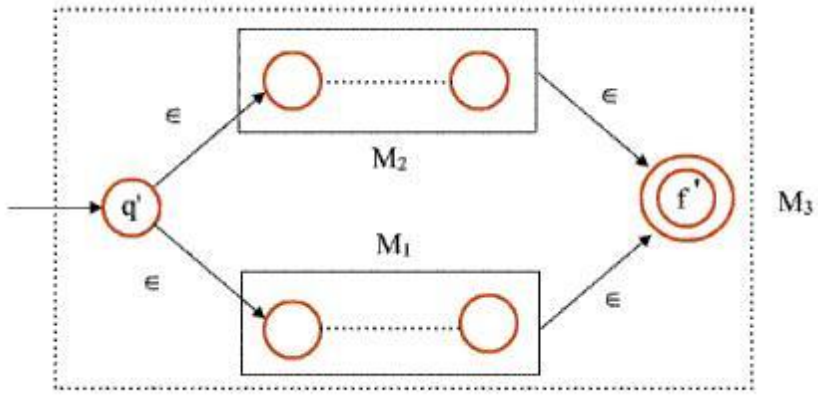
Induction :

Assume that the start of the theorem is true for REs r_1 and r_2 . Hence we can assume that we have automata M_1 and M_2 that accepts languages denoted by REs r_1 and r_2 , respectively i.e. $L(M_1) = L(r_1)$ and $L(M_2) = L(r_2)$. The FAs are represented schematically as shown below.



Each has an initial state and a final state. There are four cases to consider.

- Case (i) **:** Consider the RE $r_3 = r_1 + r_2$ denoting the language $L(r_1) \cup L(r_2)$. We construct FA M_3, from M_1 and M_2 to accept the language denoted by RE r_3 as follows **:**



Create a new (initial) start state q' and give $\in$ - transition to the initial state of M_1 and M_2 .This is the initial state of M_3.

- Create a final state f' and give $\in$-transition from the two final state of $M_1$and M_2 . f' is the only final state of $M_3$and final state of $M_1$and M_2 will be ordinary states in M_3.
- All the state of $M_1$and M_2 are also state of M_3.
- All the moves of $M_1$and M_2 are also moves of M_3. [Formal Construction]

It is easy to prove that $L(M_3) = L(r_3)$

Proof: To show that $L(M_3) = L(r_3)$ we must show that

= $L(r_1) \cup L(r_2)$

= $L(M_1) = L(M_2)$, by following transition of M_3.

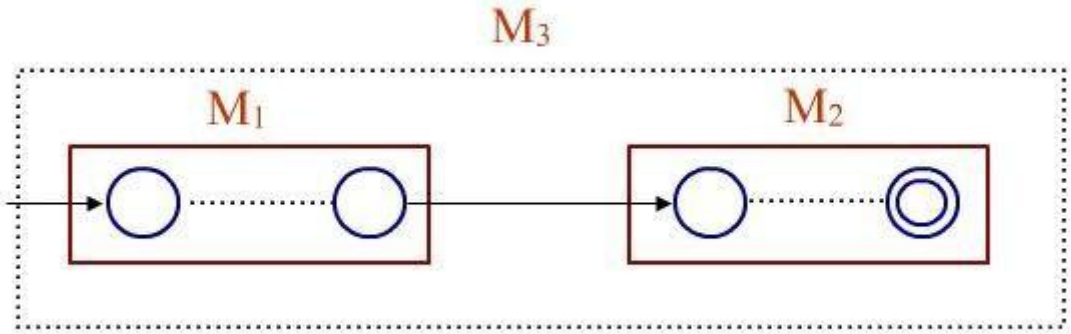
Starts at initial state q' and enters the start state of either M_1or M_2 following the transition i.e. without consuming any input. WLOG, assume that, it enters the start state of M_1. From this point onward it has to follow only the transition of $M_1$to enter the final state of M_1, because this is the only way to enter the final state of *M* by following the e-transition.(Which is the last transition & no input is taken at hte transition). Hence the whole input *w* is considered while traversing from the start state of $M_1$to the final state of M_1. Therefore M_1must accept w_3 .

Say, $w \in L(M_1)$ or $w \in L(M_2)$.

WLOG, say $w \in L(M_1)$

Therefore when M_1 process the string *w* , it starts at the initial state and enters the final state when *w* consumed totally, by following its transition. Then M_3 also accepts *w*, by starting at state q' and taking $\in$ -transition enters the start state of M_1 -follows the moves of M_1 to enter the final state of M_1 consuming input *w* thus takes $\in$ -transition to f' . Hence proved.

- Case(ii) **:** Consider the RE $r_3 = r_1 r_2$ denoting the language $L(r_1)L(r_2)$. We construct FA M_3 from M_1 & M_2 to accept $L(r_3)$ as follows **:**



Create a new start state q' and a new final state

1. Add $\in$ - transition from
   - q' to the start state of M_1
   - q' to f'
   - final state of M_1 to the start state of M_1
2. All the states of M_1 are also the states of M_3. M_3 has 2 more states than that of M_1 namely q' and f' .
3. All the moves of M_1 are also included in M_3.

By the transition of type (b), M_3 can accept $\in$.

By the transition of type (a), M_3 can enters the initial state of M_1 w/o any input and then follow all

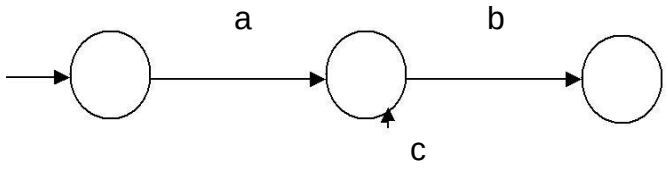
kinds moves of M_1 to enter the final state of M_1 and then following $\in$-transition can enter f'. Hence if any $w \in \Sigma^*$ is accepted by M_1 then *w* is also accepted by M_3. By the transition of type (b), strings accepted by M_1 can be repeated by any no of times & thus accepted by M_3. Hence M_3 accepts $\in$ and any string accepted by M_1 repeated (i.e. concatenated) any no of times. Hence

$L(M_3) = (L(M_1))^* = (L(r)_1)^* = r_1^*$

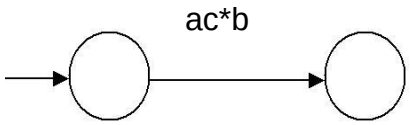
- Case(iv) **:** Let r_3 =(r_1). Then the FA M_1 is also the FA for (r_1), since the use of parentheses does not change the language denoted by the expression.

iv) Conversion of Finite Automata to Regular Expression by Elimination of States

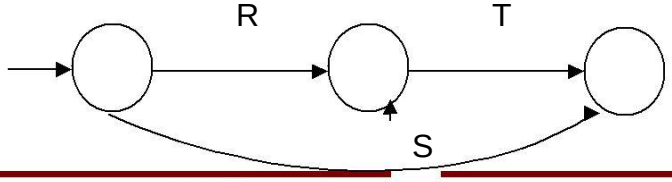
In this method, all intermediate states are eliminated in a systematic order. The principle is explained below. Consider a state 's' to be eliminated. Let 'p' be its successor and 'q' be its predecessor as shown:



It can be observed that all strings of the form ac*b take the automata from q to p and pass through s. now, s can be removed and we can attach an edge labeled ac*b from q to p directly.

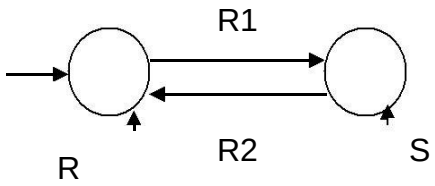


During this process, we obtain transitions labeled by regular expressions. Such diagrams are called 'Generalized transition diagrams'. The general rule can be described as below. If a state 's' is to be eliminated, we have to consider each pair of a predecessor and a successor (q,p). consider the general situation as given below.

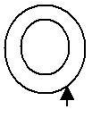


After elimination of ‘s’, we attach a direct edge labeled by R1+RS*T as shown below.

By this process, we eliminate all intermediate states leaving the start state and a final state. Configuration at the end is one of the following patterns.



Regular expression in this case is (R+R*R1S*R2)* R1S* or (R*R1S*(R2R*R1S*)*



In this case, the regular expression is R*.

Limitations of Finite Automata and Non regular Languages :

The class of languages recognized by FA s is strictly the regular set. There are certain languages which are non regular i.e. cannot be recognized by any FA

Consider the language $L = \{a^n b^n \mid n \geq 0\}$

In order to accept is language, we find that, an automaton seems to need to remember when passing the center point between *a*'s and *b*'s how many *a*'s it has seen so far. Because it should have to compare that with the number of *b*'s to either accept (when the two numbers are same) or reject (when they are not same) the input string.

But the number of *a*'s is not limited and may be much larger than the number of states since the string may be arbitrarily long. So, the amount of information the automaton need to remember is unbounded.

A finite automaton cannot remember this with only finite memory (i.e. finite number of states). The fact that *FA* s have finite memory imposes some limitations on the structure of the languages recognized. Inductively, we can say that a language is regular only if in processing any string in this language, the information that has to be remembered at any point is strictly limited. The argument given above to show that $a^n b^n$ is non regular is informal. We now present a formal method for showing that certain languages such as $a^n b^n$ are non regular.

Pumping Lemma for regular languages

In the theory of formal languages, a **pumping lemma** states that any language of a given class can be "pumped" and still belong to that class. A language can be pumped if any sufficiently long string in the language can be broken into pieces that can be repeated to produce an even longer string in the language. Thus, if there is a pumping lemma for a given language class, any language in the class will contain an infinite set of finite strings all produced by a simple rule given by the lemma. The two most important examples are the pumping lemma for regular languages and the pumping lemma for context-free languages. Unlike theorems, lemmas are specifically intended to facilitate streamlined proofs. These two lemmas are used to determine if a particular language is *not* in a given language class. However, they cannot be used to determine if a language is in a given class, since satisfying the pumping lemma is a necessary, but not sufficient, condition for class membership.

Pumping Lemma :

Let *L* be a regular language. Then the following property olds for *L*.

There exists a number $k \geq 0$ (called, the pumping length), where, if *w* is any string in *L* of length at least *k* i.e. $|w| \geq k$, then *w* may be divided into three sub strings *w* = *xyz*, satisfying the following conditions:

1. $y \neq \in$ i.e. $|y| > 0$

2. $|xy| \leq k$
3. $\forall i \geq 0,\ xy^i z \in L$

Proof : Since L is regular, there exists a *DFA* $M = (Q, \Sigma, \delta, q_0, F)$ that recognizes it, i.e. $L = L(M)$. Let the number of states in M is n.

Say, $Q = \{q_0, q_1, q_2, \cdots, q_n\}$

Consider a string $w \in L$ such that $|w| \geq n$ (we consider the language L to be infinite and hence such a string can always be found). If no string of such length is found to be in L , then the lemma becomes vacuously true.

Since $w \in L, \hat{\delta}(q_0, w) \in F$. Say $\hat{\delta}(q_0, w) = q_m$ while processing the string *w* , the *DFA M* goes through a sequence of states of states. Assume the sequence to be

$q_0, q_3, q_2, q_4, \cdots, q_i, \cdots q_l, \cdots, q_m$

start state final start

Since $|w| \geq n$, the number of states in the above sequence must be greater than $n + 1$. But number of states in M is only n. hence, by pigeonhole principle at least one state must be repeated.

Let q_i and q_l be the q_l same state and is the first state to repeat in the sequence (there may be some more, that come later in the sequence). The sequence, now, looks like

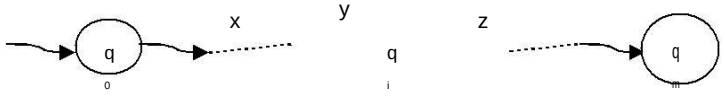
$q_0, q_3, q_2, q_4, \cdots, q_i, \cdots q_l, \cdots, q_n$

which indicates that there must be sub strings *x*, *y*, *z* of *w* such that

$\hat{\delta}(q_0, x) = q_i$

$\hat{\delta}(q_i, y) = q_i$

$\hat{\delta}(q_i, z) = q_m$



This situation is depicted in the figure

Since $q_l (= q_i)$ is the first repeated state, we have, $|xy| \leq n$ and at the same time *y* cannot be empty i.e $|y| > 0$. From the above, it immediately follows that $\hat{\delta}(q_0, xz) = q_m$. Hence $xz = xy^0 z \in L$. Similarly,

$\hat{\delta}(q_0, xy^2 z) = q_m$ implying $xy^2 z \in L$

$\hat{\delta}(q_0, xy^3 z) = q_m$ implying $xy^3 z \in L$

and so on.

That is, starting at the loop on state can be omitted, taken once, twice, or many more times, (by the *DFA M*) eventually arriving at the final state

Thus, accepting the string *xz*, *xyz*, xy^2z,... i.e. xy^iz for

all $i \geq 0$ Hence $\forall i \geq 0, xy^i z \in L$.

We can use the pumping lemma to show that some languages are non regular.

Pumping Lemma

In this section we give a necessary condition for an input string to belong to a regular set. The result is called *pumping lemma* as it gives a method of pumping (generating) many input strings from a given string. As pumping lemma gives a necessary condition, it can be used to show that certain sets are not regular.

Theorem 5.5 (Pumping Lemma) Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton with n states. Let L be the regular set accepted by M. Let $w \in L$ and $|w| \geq m$. If $m \geq n$, then there exists x, y, z such that $w = xyz$, $y \neq \Lambda$ and $xy^iz \in L$ for each $i \geq 0$.

Proof Let

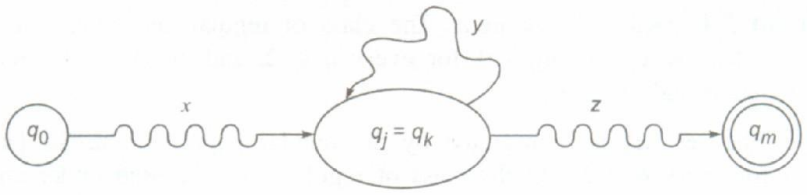
$$w = a_1a_2 \ldots a_m, \qquad m \geq n$$

$$\delta(q_0, a_1a_2 \ldots a_i) = q_i \quad \text{for } i = 1, 2, \ldots, m; \qquad Q_1 = \{q_0, q_1, \ldots, q_m\}$$

That is, Q_1 is the sequence of states in the path with path value $w = a_1a_2 \ldots a_m$. As there are only n distinct states, at least two states in Q_1 must coincide. Among the various pairs of repeated states, we take the first pair. Let us take them as q_j and $q_k (q_j = q_k)$. Then j and k satisfy the condition $0 \leq j < k \leq n$.

The string w can be decomposed into three substrings $a_1a_2 \ldots a_j$, $a_{j+1} \ldots a_k$ and $a_{k+1} \ldots a_m$. Let x, y, z denote these strings $a_1a_2 \ldots a_j$, $a_{j+1} \ldots a_k$, $a_{k+1} \ldots a_m$, respectively. As $k \leq n$, $|xy| \leq n$ and $w = xyz$. The path with the path value w in the transition diagram of M is shown in Fig. 5.27.

The automaton M starts from the initial state q_0. On applying the string x, it reaches $q_j (= q_k)$. On applying the string y, it comes back to $q_j (= q_k)$. So after application of y^i for each $i \geq 0$, the automaton is in the same state q_j. On applying z, it reaches q_m, a final state. Hence, $xy^iz \in L$. As every state in Q_1 is obtained by applying an input symbol, $y \neq \Lambda$. ∎



Application of Pumping Lemma

This theorem can be used to prove that certain sets are not regular. We now give the steps needed for proving that a given set is not regular.

Step 1 Assume that L is regular. Let n be the number of states in the corresponding finite automaton.

Step 2 Choose a string w such that $|w| \geq n$. Use pumping lemma to write $w = xyz$, with $|xy| \leq n$ and $|y| > 0$.

Step 3 Find a suitable integer i such that $xy^iz \notin L$. This contradicts our assumption. Hence L is not regular.

Note: The crucial part of the procedure is to find i such that $xy^iz \notin L$. In some cases we prove $xy^iz \notin L$ by considering $|xy^iz|$. In some cases we may have to use the 'structure' of strings in L.

Example 1:

Show that the set $L = \{a^{i^2} \mid i \geq 1\}$ is not regular.

Solution

Step 1 Suppose L is regular. Let n be the number of states in the finite automaton accepting L.

Step 2 Let $w = a^{n^2}$. Then $|w| = n^2 > n$. By pumping lemma, we can write $w = xyz$ with $|xy| \leq n$ and $|y| > 0$.

Step 3 Consider xy^2z. $|xy^2z| = |x| + 2|y| + |z| > |x| + |y| + |z|$ as $|y| > 0$. This means $n^2 = |xyz| = |x| + |y| + |z| < |xy^2z|$. As $|xy| \leq n$, we have $|y| \leq n$. Therefore,

$$|xy^2z| = |x| + 2|y| + |z| \leq n^2 + n$$

i.e.

$$n^2 < |xy^2z| \leq n^2 + n < n^2 + n + n + 1$$

Hence, $|xy^2z|$ strictly lies between n^2 and $(n + 1)^2$, but is not equal to any one of them. Thus $|xy^2z|$ is not a perfect square and so $xy^2z \notin L$. But by pumping lemma, $xy^2z \in L$. This is a contradiction.

Example 2:

Show that $L = \{a^p \mid p \text{ is a prime}\}$ is not regular.

Solution

Step 1 We suppose L is regular. Let n be the number of states in the finite automaton accepting L.

Step 2 Let p be a prime number greater than n. Let $w = a^p$. By pumping lemma, w can be written as $w = xyz$, with $|xy| \leq n$ and $|y| > 0$. x, y, z are simply strings of a's. So, $y = a^m$ for some $m \geq 1$ (and $\leq n$).

Step 3 Let $i = p + 1$. Then $|xy^iz| = |xyz| + |y^{i-1}| = p + (i - 1)m = p + pm$. By pumping lemma, $xy^iz \in L$. But $|xy^iz| = p + pm = p(1 + m)$, and $p(1 + m)$ is not a prime. So $xy^iz \notin L$. This is a contradiction. Thus L is not regular.

Example 3:

Show that $L = \{0^i 1^i \mid i \geq 1\}$ is not regular.

Solution

Step 1 Suppose L is regular. Let n be the number of states in the finite automaton accepting L.

Step 2 Let $w = 0^n 1^n$. Then $|w| = 2n > n$. By pumping lemma, we write $w = xyz$ with $|xy| \leq n$ and $|y| \neq 0$.

Step 3 We want to find i so that $xy^i z \notin L$ for getting a contradiction. The string y can be in any of the following forms:

Case 1 y has 0's, i.e. $y = 0^k$ for some $k \geq 1$.

Case 2 y has only 1's, i.e. $y = 1^l$ for some $l \geq 1$.

Case 3 y has both 0's and 1's, i.e. $y = 0^k 1^j$ for some $k, j \geq 1$.

In Case 1, we can take $i = 0$. As $xyz = 0^n 1^n$, $xz = 0^{n-k} 1^n$. As $k \geq 1$, $n - k \neq n$. So, $xz \notin L$.

In Case 2, take $i = 0$. As before, xz is $0^n 1^{n-l}$ and $n \neq n - l$. So, $xz \notin L$.

In Case 3, take $i = 2$. As $xyz = 0^{n-k} 0^k 1^j 1^{n-j}$, $xy^2z = 0^{n-k}\, 0^k 1^j 0^k 1^j 1^{n-j}$. As xy^2z is not of the form $0^i 1^i$, $xy^2z \notin L$.

Thus in all the cases we get a contradiction. Therefore, L is not regular.

Example 4:

Show that $L = \{ww \mid w \in \{a, b\}^*\}$ is not regular.

Solution

Step 1 Suppose L is regular. Let n be the number of states in the automaton M accepting L.

Step 2 Let us consider $ww = a^n b a^n b$ in L. $|ww| = 2(n + 1) > n$. We can apply pumping lemma to write $ww = xyz$ with $|y| \neq 0$, $|xy| \leq n$.

Step 3 We want to find i so that $xy^i z \notin L$ for getting a contradiction. The string y can be in only one of the following forms:

Case 1 y has no b's, i.e. $y = a^k$ for some $k \geq 1$.

Case 2 y has only one b.

We may note that y cannot have two b's. If so, $|y| \geq n + 2$. But $|y| \leq |xy| \leq n$. In Case 1, we can take $i = 0$. Then $xy^0z = xz$ is of the form $a^m b a^n b$, where $m = n - k < n$ (or $a^n b a^m b$). We cannot write xz in the form uu with $u \in \{a, b\}^*$, and so $xz \notin L$. In Case 2 too, we can take $i = 0$. Then $xy^0z = xz$ has only one b (as one b is removed from xyz, b being in y). So $xz \notin L$ as any element in L should have an even number of a's and an even number of b's.

Thus in both the cases we get a contradiction. Therefore, L is not regular.

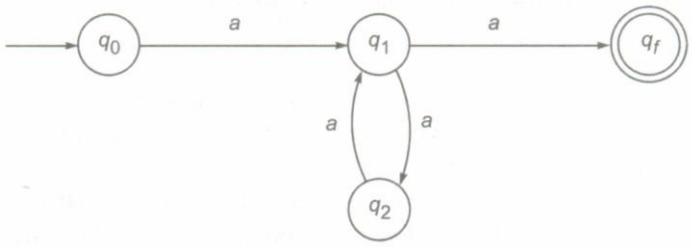
Note: If a set L of strings over Σ is given and if we have to test whether L is regular or not, we try to write a regular expression representing L using the definition of L. If this is not possible, we use pumping lemma to prove that L is not regular.

Example 5:

Is $L = \{a^{2n} \mid n \geq 1\}$ regular?

Solution

We can write a^{2n} as $a(a^2)^i a$, where $i \geq 0$. Now $\{(a^2)^i \mid i \geq 0\}$ is simply $\{a^2\}^*$. So L is represented by the regular expression $\mathbf{a(P)^*a}$, where $\mathbf{P}$ represents $\{a^2\}$. The corresponding finite automaton (using the construction given in Section 5.2.5) is shown in Fig. 5.28.



Applications of Finite state automata

i) Lexical Analyzers

In computer science, **lexical analysis** is the process of converting a sequence of characters into a sequence of tokens. Programs performing lexical analysis are called **lexical analyzers** or **lexers**. A lexer is often organized as separate **scanner** and **tokenizer** functions, though the boundaries may not be clearly defined.

Lexical grammar

The specification of a programming language will include a set of rules, often expressed syntactically, specifying the set of possible character sequences that can form a token or lexeme. The whitespace characters are often ignored during lexical analysis.

Token

A **token** is a categorized block of text. The block of text corresponding to the token is known as a lexeme. A lexical analyzer processes *lexemes* to categorize them according to function, giving them **meaning**. This assignment of meaning is known as **tokenization**. A token can look like anything: English, gibberish symbols, anything; it just needs to be a useful part of the structured text.

Tokenizer

Tokenization is the process of demarcating and possibly classifying sections of a string of input characters. The resulting tokens are then passed on to some other form of processing. The process can be considered a sub-task of parsing input.

Lexical analyzer generators

- Flex - Alternative variant of the classic 'lex' (C/C++).
- JLex - A Lexical Analyzer Generator for Java.
- Quex - (or 'Queχ') A Mode Oriented Lexical Analyzer Generator for C++.
- OOLEX - An Object Oriented Lexical Analyzer Generator.
- re2c
- PLY - An implementation of lex and yacc parsing tools for Python.

ii) Text Search

String searching algorithm (text searching)

String searching algorithms, sometimes called **string matching algorithms**, are an important class of string algorithms that try to find a place where one or several strings (also called patterns) are found within a larger string or text.

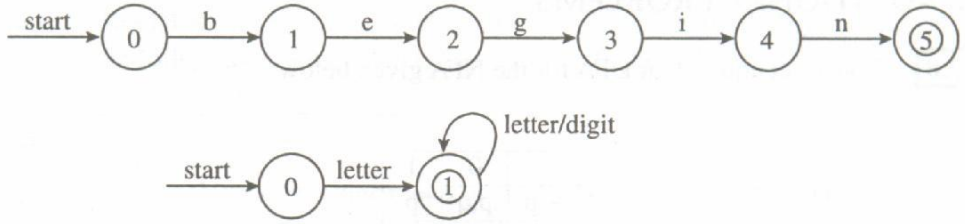
Let Σ be an alphabet (finite set). Formally, both the pattern and searched text are concatenations of elements of Σ. The Σ may be a usual human alphabet (for example, the letters A through Z in English). Other applications may use *binary alphabet* (Σ = {0,1}) or *DNA alphabet* (Σ = {A,C,G,T}) in bioinformatics.

In practice how the string is encoded can affect the feasible string search algorithms. In particular if a variable width encoding is in use then it is slow (time proportional to N) to find the Nth character. This will significantly slow down many of the more advanced search algorithms. A possible solution is to search for the sequence of code units instead, but doing so may produce false matches unless the encoding is specifically designed to avoid it.

The concept of finite automata has several applications in many areas like compiler design, special purpose hardware design, protocol specification etc. Some of the applications are described below.

(a) **Compiler Design:** Lexical analysis is an important phase in compiler. In lexical analysis, transition diagram (FA) is used in recognition of tokens.

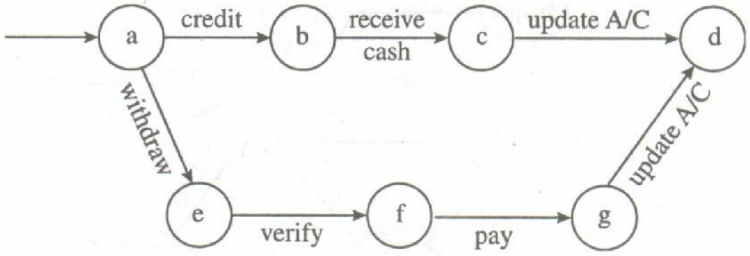
Example: Transition diagrams to recognize the tokens 'begin' and identifier are shown below.



Relation between FA and regular expressions (described later) has made many UNIX packages like LEX, text formatting tools possible. Many UNIX tools extensively use regular expressions to specify user inputs. Implementation is essentially, a program to simulate the equivalent DFA.

(b) **Hardware design:** In the design of computers, FA is used to design control unit of a computer. A typical sequence of operations in a computer consists of a repetition of instructions and each instruction involves the actions fetch, decode, fetch the operand, and execute. Each state of a FA represents a specific stage of instruction cycle.

(c) **Protocol specification:** Any system consists of an interconnected set of subsystems. A protocol is a set of rules for proper coordination of activities of a system. Such a protocol can be described by transition diagrams. An example of a Bank is shown in



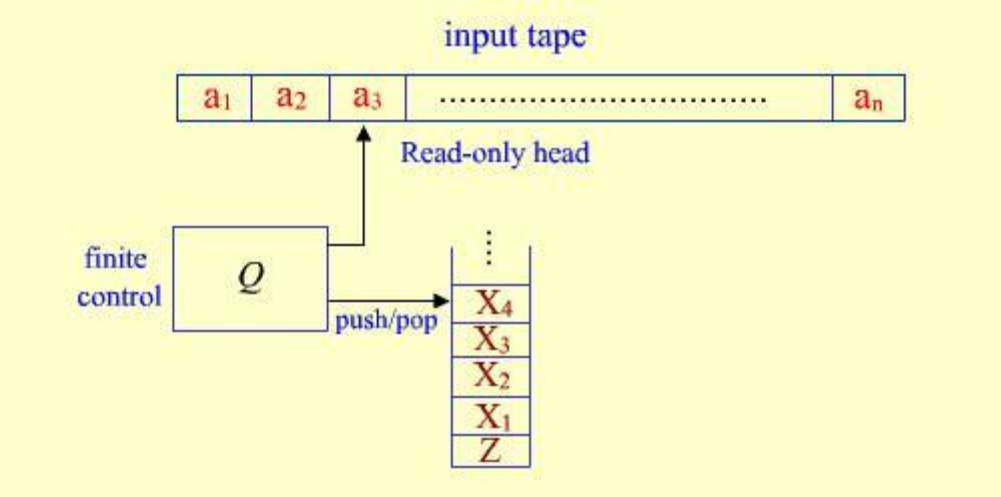
State a represents the entry of a customer. After he wishes to credit some cash to his a/c no, system is in state b. Then, the cashier receives cash (state c) and updates the a/c to reach the final state of this transaction. If the customer wants to withdraw cash, he submits the withdrawal slip (or Cheque). Then, system is in state e. In state e, his Cheque/slip is verified to confirm that there is sufficient balance. After verification, system is in state f. Then the customer is paid cash and system reaches state g. Then, customer's a/c is updated and the system reaches the final state d of the transaction.

Introduction Push Down Automaton

In the case of finite automata, several languages cannot be recognized by them. Some important programming constructs involving nested structures are beyond the capacity of finite automata. Normally regular languages are accepted by finite automata and Pushdown Automata are used to recognize Context Free Language.

It is observed that FA has limited capability. (in the sense that the class of languages accepted or characterized by them is small). This is due to the "finite memory" (number of states) and "no external memory" involved with them. A PDA is simply an NFA augmented with an "external stack memory". The addition of a stack provides the PDA with a last-in, first-out memory management capability. This "Stack" or "pushdown store" can be used to record potentially unbounded information. It is due to this memory management capability with the help of the stack that a PDA can overcome the memory limitations that prevents a FA to accept many interesting languages like $\{a^n b^n \mid n \geq 0\}$. Although, a PDA can store an unbounded amount of information on the stack, its access to the information on the stack is limited. It can push an element onto the top of the stack and pop off an element from the top of the stack. To read down into the stack the top elements must be popped off and are lost. Due to this limited access to the information on the stack, a PDA still has some limitations and cannot accept some other interesting languages.

Consider the problem of recognition of the context-free language $L=\{a^n b^n / n>=0\}$. The a's in the given string are added to the stack. When a symbol 'b' is encountered in the input string, an 'a' is removed from the stack. Thus the matching of number of 'a's and 'b's is accomplished. This type of arrangement where a finite automaton has a stack leads to the generation of a "Pushdown Automaton" (PDA). Pushdown automation is an extension of the NFA.



As shown in figure, a PDA has three components: an input tape with read only head, a finite control and a pushdown store.

The input head is read-only and may only move from left to right, one symbol (or cell) at a time. In each step, the PDA pops the top symbol off the stack; based on this symbol, the input symbol it is currently reading, and its present state, it can push a sequence of symbols onto the stack, move its read-only head one cell (or symbol) to the right, and enter a new state, as defined by the transition rules of the PDA.

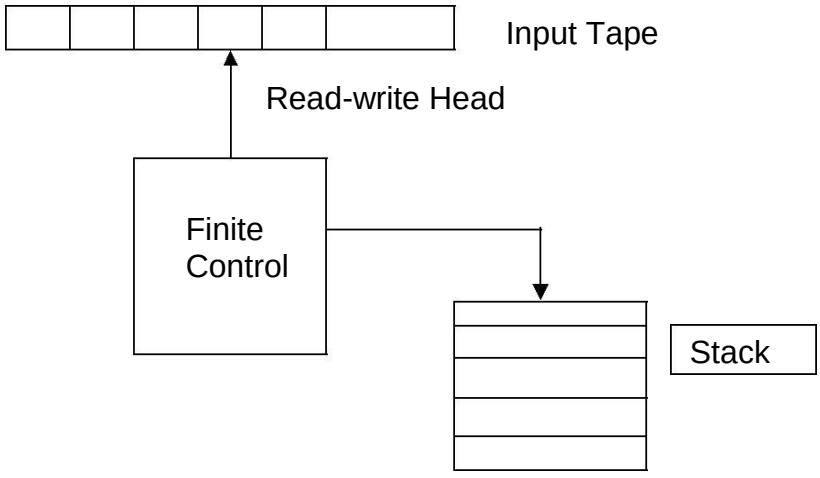
PDA are nondeterministic, by default. That is, $\in$ - transitions are also allowed in which the PDA can pop and push, and change state without reading the next input symbol or moving its read-only head. Besides this, there may be multiple options for possible next moves.

Pushdown Automata

Basic model of PDA consists of 3 components:

vi) an infinite tape
vii) a finite control
viii) a stack

Now let us consider the 'concept of PDA' and the way it 'operates'.



Each move of a PDA depends on the current state, input symbol and top of stack symbol. The finite control reads the input from the input tape, and same time it reads the symbol from stack top. It depends on finite control. The stack is also called 'Pushdown Store'. It is a read-

In this representation, there is a node for each state. A transition $\delta(q_i,a,A)=(q_j,\alpha)$ is represented by an edge from q_i to q_j and labeled by **a,A/**α

Eg: Construct a PDA to recognize the language $L=\{a^n b^n / n>=1\}$ Solution: PDA can be defined as $M=(Q,\Sigma,\Gamma,\delta,q_0,Z_0,F)$. For this PDA,

$Q=\{q_0,q_1,q_2\}$ $\Sigma=\{a,b\}$ $\Gamma=\{Z_0,A\}$ δ can be defined as follows:

= $\delta(\mathbf{q0},a,Z_0)=(q_0,AZ_0)$

= $\delta(q_0,a,A)=(q_0,AA)$

= $\delta(q_0,b,A)=(q_1,\epsilon)$

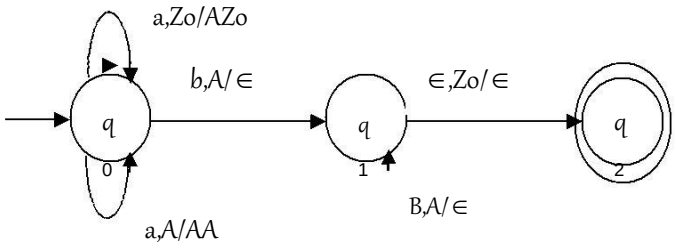
= $\delta(q_1,b,A)=(q_1,\epsilon)$

= $\delta(q_1,\epsilon,Z_0)=(\mathbf{q2},\epsilon)$

Now M can be defined as $M=(\{q_0,q_1,q_2\},\{a,b\},\{Z_0,A\},\delta,q_0,Z_0,q_2)$. The description of PDA by using Transition table and Transition diagram is as follows:

By using transition table:

State	TOS symbol	a	b	∈
q_0	Z_0	(q_0,AZ_0)	---	---
	A	(q_0,AA)	$(q_1,\in)$	---
q_1	A	---	$(q_1,\in)$	---
	Z_0	---	---	$(q_2,\in)$



Example 2 : We give an example of a PDA M that accepts the set of balanced strings of parentheses [] by empty stack. The PDA M is given below.

$M = (\{q\}, \{[,]\}, \{z,[\}, \delta, q, z, \phi)$ where δ is defined as

$\delta(q,[,z) = \{(q,[z)\}$
$\delta(q,[,[) = \{(q,[[)\}$
$\delta(q,],[) = \{(q,\in)\}$
$\delta(q,\in,z) = \{(q,\in)\}$

Informally, whenever it sees a [, it will push the] onto the stack. (first two transitions), and whenever it sees a] and the top of the stack symbol is [, it will pop the symbol [off the stack. (The third transition). The fourth transition is used when the input is exhausted in order to pop z off the stack (to empty the stack) and accept. Note that there is only one state and no final state.

The following is a sequence of configurations leading to the acceptance of the string [[ ] [ ]] [ ]

$(q,[[\,][\,]][\,],z) \vdash (q,[\,][\,]][\,],[z) \vdash (q,\,][\,]][\,],[[z) \vdash (q,[\,]][\,],[z) \vdash (q,\,]][\,],[[z)$

- $(q,\,]][\,],[[z) \vdash (q,\,][\,],[z) \vdash (q,[\,],z) \vdash (q,],[z) \vdash (q,\in,z) \vdash (q,\in,\in)$

Deterministic and nondeterministic PDA

A PDA is deterministic if there is never a choice of move in any situation. If δ(q,a,A) contains more than one pair, then surely the PDA is non deterministic because we can choose among these pairs when deciding on the next move. However even if δ(q,a,A) is always a singleton, we could still have a choice between using a real input symbol or making a move on ϵ. Thus we define a PDA as

P=(Q,Σ,Γ,δ,q_0,Z_0,F) to be deterministic if and only if the following conditions are met:

1. δ(q,a,A) has at most one member for any q in Q, a in Σ or a= ϵ and A in Γ.
2. δ(q,a,A) is nonempty, for some q in Q, a in Σ then δ(q,ϵ,A) must be empty.

Unlike finite automaton DPDA allows ϵ-transition. But if for any combination (q,A), ϵ-transition is present, transition on any other symbol is not allowed.

δ can be defined as:

$\delta : Q \times (\Sigma \cup \epsilon) \times \Gamma \rightarrow Q \times \Gamma^*$

eg:- for DPDA $\{wcw^R / w \epsilon \{0,1\}^*\}$

for NPDA $\{ww^R / w \epsilon \{0,1\}^*\}$

In the first one we cannot determine when to switch over from 'pushing state' to 'matching state'. In the second one PDA cannot determine whether to push A or pop A while scanning 0's. Hence a choice is necessary.
Here

δ is be defined as:

$\delta : Q \times (\Sigma \cup \epsilon) \times \Gamma \rightarrow 2^{(Q \times \Gamma^*)}$

Example 1: Construct a PDA that accepts the language $\{a^n b^n \mid n \geq 0\}$.

$M = (Q, \Sigma, \Gamma, \delta, q_1, Z, F)$

$Q = \{q_1, q_2, q_3, q_4\}$

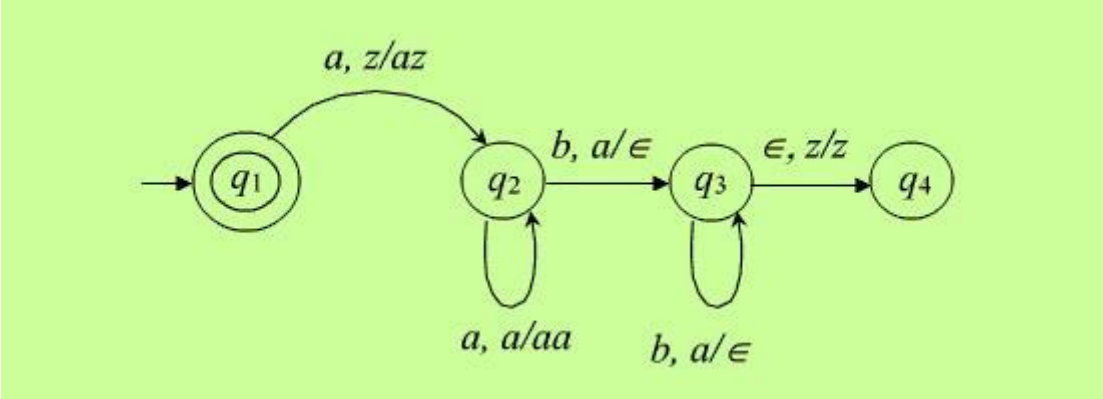
$\Sigma = \{a, b\}$

$\Gamma = \{a, b, z\}$

$F = \{q_1, q_4\}$, and δ consists of the following transitions

1. $\delta(q_1, a, z) = \{(q_2, az)\}$
2. $\delta(q_2, a, a) = \{(q_2, aa)\}$
3. $\delta(q_2, b, a) = \{(q_3, \in)\}$
4. $\delta(q_3, b, a) = \{(q_3, \in)\}$
5. $\delta(q_3, \in, z) = \{(q_4, z)\}$

The PDA can also be described by the adjacent transition diagram.



Informally, whenever the PDA *M* sees an input *a* in the start state q_1 with the start symbol *z* on the top of the stack it pushes *a* onto the stack and changes state to q_2 . (to remember that it has seen the first '*a*'). On state q_2 if it sees anymore *a*, it simply pushes it onto the stack.

Note that when *M* is on state q_2 , the symbol on the top of the stack can only be *a*. On state q_2 if it sees the first *b* with *a* on the top of the stack, then it needs to start comparison of numbers of *a*'s and *b*'s, since all the *a*'s at the beginning of the input have already been pushed onto the stack. It start this process by popping off the a from the top of the stack and enters in state q3 (to remember that the comparison process has begun). On state q_3 , it expects only b's in the input (if it sees any more a in the input thus the input will not be in the proper form of $a^n b^n$). Hence there is no more on input a when it is in state q_3 . On state q_3 it pops off an a from the top of the stack for every b in the input. When it sees the last b on state q3 (i.e. when the input is exhausted), then the last a from the stack will be popped off and the start symbol z is exposed. This is the only possible case when the input (i.e. on $\in$-input) the PDA *M* will move to state q_4 which is an accept state.

We can show the computation of the PDA on a given input using the IDs and next move relations. For example, following are the computation on two input strings.

Let the input be aabb. we start with the start configuration and proceed to the subsequent IDs using the transition function defined

$(q_1, aabb, z) \vdash (q_2, abb, az)$ (using transition 1)

$\vdash (q_2, bb, aaz)$ (using transition 2)

$\vdash (q_3, b, az)$ (using transition 3)

$\vdash (q_3, \in, z)$ (using transition 4)

$\vdash (q_4, \in, z)$ (using transition 5)

q_4 is final state. Hence ,accept. So the string *aabb* is rightly accepted by *M*.

we can show the computation of the PDA on a given input using the IDs and next move relations. For example, following are the computation on two input strings.

i) Let the input be *aabab*.

$(q_1, aabab, z) \vdash (q_2, abab, az)$

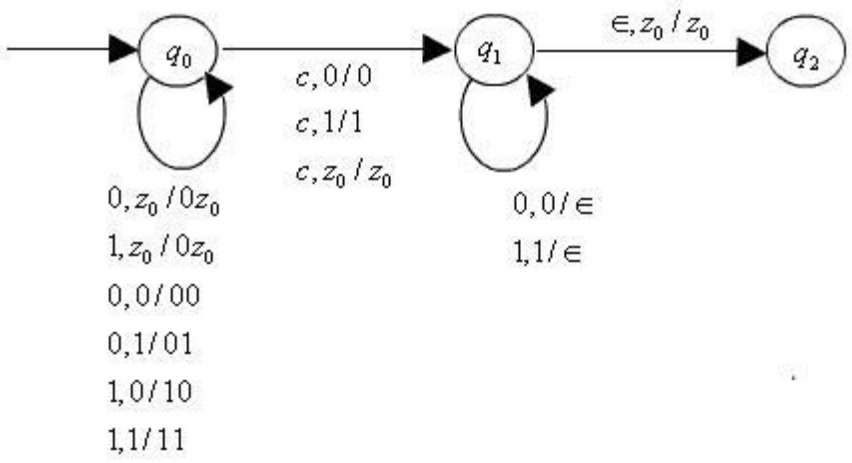
$\vdash (q_2, bab, aaz)$

$\vdash (q_3, ab, az)$

No further move is defined at this point.

Hence the PDA gets stuck and the string *aabab* is not accepted.

Example 2: The language $L = \{ wcw^i / w \in (0+1)^* \}$ is a *DCFL* . The following DPDA accepts *L*



The moves satisfying the conditions is given in the definition. As the PDA reads the first half of the input, it remains in the start state q_0 and pushes the input symbols on the stack. When it reads the symbol c it changes the state from q_0 to $q_1$without changing the stack. On state q_1 it simply matches input symbols with the stack symbols and erases in case of a match. The moves satisfying the conditions is given in the definition. As the PDA reads the first half of the input, it remains in the start state q_0 and pushes the input symbols on the stock. When it reads the symbol c , it changes the state from q_0 to q_1 without changing the stock. On state q_1 it simply matches input symbols with the stock symbols and erases in case of a match. That is, the symbol in wcw^R tells the m/c when to start looking for w^R Once the input is extended, then the symbol z_0 on stock indicates a proper match for the input to be wcw^R and hence it accepts by entering state q_2, which is a final state.

Example 3: Consider the language $L=\left\{ww^R / w\in(0+1)^*\right\}$. In this case there is no way to determine when to start comparison because of absence of the symbol c in the middle/ The *PDA* in this case has to guess non-deterministically when the middle symbol comes in the input.

DPDA s and FAs: *DCFL* s and Regular languages

Equivalence of *DFA* & *NFA* proves that non determination does not add power in case of *FA* s. But it is not true in case of *PDA* s, i.e., it can be shown that nondeterministic *PDA* s are more powerful than DPDA s. In fact, *DCFL* s is a class of languages that lies properly between the class of regular languages and *CFL* s. The following discussion proves this fact.

Theorem :If *L* is a regular language, then there is some *DPDA M* such that $L=L(M)$.

Proof : Since L is regular, then exists a *DFA D* such that $L=L(D)$.

The *PDA M* can be constructed from *D* (with an additional stock) that simulates all the moves of *D* on any input just by ignoring its stock. That is if $D=(Q,\Sigma,\{Z\},\delta',Z,F)$ when $\delta'(p,a,Z)=(q,Z)\forall p,q\in Q$ Such that $\delta(p,a)=q$ It is easy to see that $(q_0,w,Z)\overset{*}{\underset{M}{\vdash}}(p,\in,Z)$ iff $\delta^*(q_0,w)=p$

Again, the language wcw^R can be shown to be non-regular by using pumping lemma. But, the *DPDA* presented in the example above accepts this language. Hence the class of *DCFL* s properly includes the class of regular languages.

Context free grammar

A grammar is nothing but a set of rules to define valid sentences in any languages. Context free Grammar (CFG) is the most important class of languages in practical applications. They are used in compilers, text formatters and natural language processing etc.

Definition

A CFG can be defined as G=(V,T,P,S) where V is the set of non terminals, T is the set of terminals, S is the start symbol and P is the set of productions of the form A α where A$\in$V, $\alpha\in$(VUT)*. Productions with same LHS are clubbed together. ie, if

A α1
A α2
A α3

A αn then we can write A α1/α2/α3/……/αn

CFG derive their name from the fact that the substitution of the variable on the left of a production can be made at any time, such a variable appears in the sentential form. It does not depend on the symbols in the rest of the sentential form. ie, the contexts. This feature is the consequence of allowing only a single variable on the left side of the production.

If a CFG is given, we can infer whether a string ‘w’ can be generated by G wither by deriving ‘w’ from S or S from ‘w’. Former one is called “derivation” and latter procedure is called “recursive inference”.

Leftmost and Rightmost Derivations

In CFG, at each step, one non terminal is replaced by its RHS. In general, they can be replaced in any order. But, there are 2 systematic derivations, a) Leftmost derivation b) Rightmost derivation.

In leftmost derivation, at each step, leftmost non terminal is replaced by its RHS. The symbol ==> is used to denote leftmost derivation. The corresponding sentential form is called “left sentential form”.

In rightmost derivation, at each step, rightmost non terminal is replaced by its RHS. The symbol ==> is used to denote rightmost derivation. The corresponding sentential form is called "right sentential form".

Eg: Let regular expression (a+b)(a+b+0+1)*

Regular grammar to specify the expression is:

E E+E/ E-E/ E*E/I

I Ia/Ib/I0/I1/a/b

Let w=a11*b+a

Leftmost derivation

E==>E+E
 ==>E*E+E
 ==>I*E+E
 ==>I1*E+E
 ==>I11*E+E
 ==>a11*E+E
 ==>a11*I+E
 ==>a11*b+E
 ==>a11*b+I
 ==>a11*b+a

Rightmost derivation

E==>E+E
 ==>E+I
 ==>E+a
 ==>E*E+a
 ==>E*I+a
 ==>E*b+a
 ==>I*b+a
 ==>I1*b+a
 ==>I11*b+a
 ==>a11*b+a

Derivation tree (Parse tree)

The derivation in a CFG can be represented by using trees called 'derivation tree' or 'parse tree'. A derivation tree for a CFG is a tree satisfying the following :

i) every vertex has a label which is a variable (non terminal) or terminal.
ii) The root has label which is non terminal
iii) The label of an internal vertex is a variable.

ie, a derivation tree is a labeled tree in which each internal node is labeled by a non terminal and leaves are labeled by terminals. Strings formed by labels of the leaves traversed from left to right is called the 'yield of the parse tree'. Ie, the yield of a derivation tree is the concatenation of the labels of the leaves without repetition in the left-to-right ordering.

Eg: Let G=({S,A},{a,b},P,S) where P is defined as S aAS/a

A SbA/ba/SS

Show that S=*=>aabbaa and construct a derivation tree. Solution

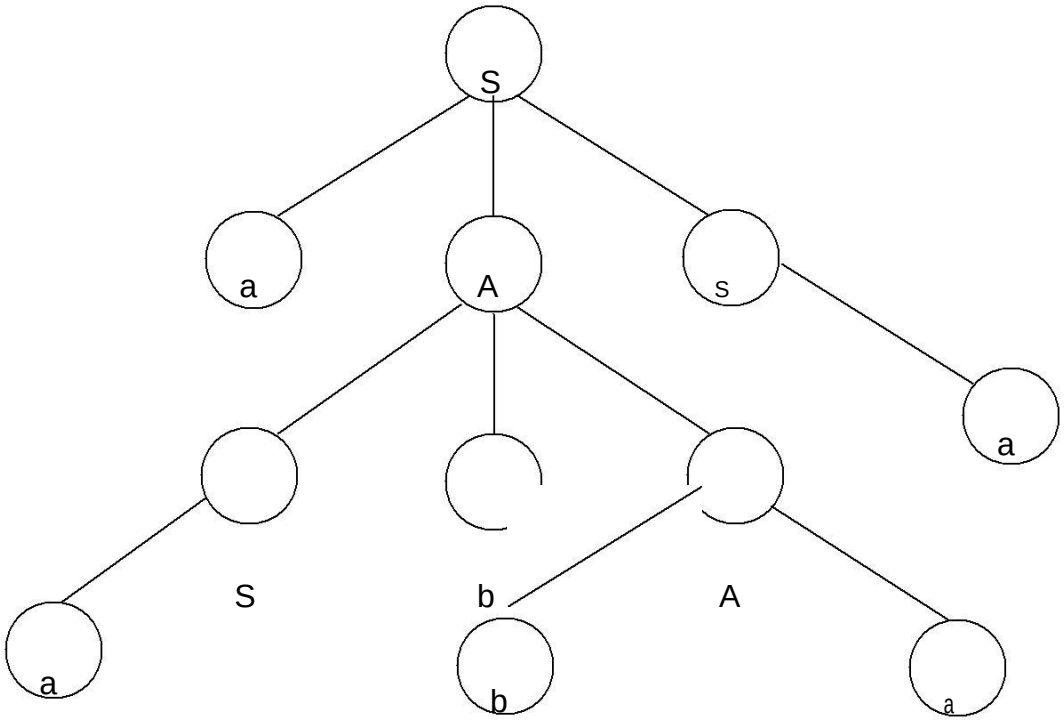
S aAS

aSbAS

aabAS

aabbaS

aabbaa (by leftmost derivation)



Ambiguity in CFGs

It is possible that a sentence might have 2 different parse trees with respect a given grammar G. a CFG G is said to be ambiguous if there is a sentence which has more than one parse tree. ie, the same terminal string may be the yield of 2 parse trees.

A terminal string $w \in L(G)$ is ambiguous if there exists 2 or more derivation trees for w.

Eg:S S+S	S S*S
a+S	S+S*S
a+S*S	a+S*S
a+a*S	a+a*S
a+a*b	a+a*b

Ambiguity is basically the property of the grammar and not the language. Same language may have more than one grammar. Some are ambiguous and others are unambiguous.

Unambiguous grammar for the above

a*b+b is E E+T/T

T T*I/I I

Ia/Ib/a/b

However, there exists language for which every grammar is ambiguous. Such languages are termed “inherently ambiguous”. A language L is said to be inherently ambiguous if every grammar for it is ambiguous.

Simplification of CFGs

Here there are some procedures for producing equivalent, but simpler grammars for a given grammar. Major steps are

i) Removal of useless symbols(non terminal)

ii) Removal of unit productions

iii) Removal of $\in$(null) productions

(i) Removal of useless symbols(non terminal)

Here we are going to identify those symbols which do not play any role in the derivation of any string ‘w’ in L(G). These symbols are called useless symbols. And then eliminate the identified production, which contains useless symbols, from the CFG.

A symbol in CFG is useful if and only if

a) Y=*=>w, where w$\in$L(G) and w is in T*. ie, Y leads to a string of terminals. Here Y is said to be “generating”.

b) if there is a derivation S=*=> αYβ=*=>w where w$\in$L(G) for some α,β; then Y is said to be “reachable”.

So surely a symbol that is useful will be both generating and reachable. Therefore the simplification of CFG involves the following steps.

i) Identify non-generating symbols in the given CFG and eliminate those productions which contain non generating symbols.

ii) Identify non-reachable symbols in the grammar and eliminate those productions which contain non reachable symbols.

After this process, CFG will have only useful symbols.

Eg: Remove useless symbols from the following grammar

S AB/a

A b

Solution:

Here B is a non generating symbol. Since B is not deriving any terminals. So eliminate S AB from CFG. So we get,

S a

A b

Here A is a non reachable symbol, since it cannot be reached by starting non-terminal S. so we can eliminate A b, and now the reduced CFG is

S a

(ii) Removal of unit production

A production of the form non-terminal one non-terminal ie a production of the form A B is called “unit production”. Following algorithm can be used to eliminate the unit production.

Algorithm

While(there exists a unit production, A B)

{

select a unit production A B, such that there exist a production B α,

where α is a terminal.

for(every non-unit production B α)

add a production A α to the grammar

eliminate A B from the grammar

}

eg: Remove unit productions from the

CFG S AB

A a B

C/b C D

D E E a

Solution:

Here unit productions are

B C

C D

D E

Here we cant remove the production B C since C D; ie, D is not a terminal. Similar is the case for C D since D E. But for D E, this can be eliminated since there is a production E a. Therefore this can be changed to the production D a. so the grammar becomes,

S AB A a

B C/b C

D D a E a

Now we can remove C D by D a, it becomes C a Therefore,

S AB A a

B C/b C a

D a E a

B C, C a ==>B a Therefore,

S AB

A a

B a/b

All others are useless. So this is the reduced grammar.

(ii) Removal of ∈ (null) production

We have to eliminate productions of the form A Є, which are called Є-productions. If Є is in L(G), we cannot eliminate all Є-productions from G, but if Є is not in L(G), we can eliminate all Є-productions from G.

In a given CFG, we call a non-terminal N nullable if there is a
production N Є or
there is a derivation that starts at N and leads to Є ie, N==> Є

To eliminate Є-productions from a grammar G we use the following technique.

If A Є is a production to be eliminated then we look for all productions, whose right side contains A, and replace each occurrence of A in each of these productions to obtain the non Є-productions. Now these resultant non Є-productions must be added to the grammar to keep the language generated the same.

eg: S aA A
b/ Є

Solution:

Here A Є is Є-production. Put Є in place of A at right side of productions and add the resulted productions to the grammar. So we get,

S a

Now add this new productions to keep the language generated by this grammar same. Therefore,

S aA

S a A

b

Normal Forms for CFGs

There are two important standard forms for CFGs:

i) Chomsky Normal Form(CNF)

ii) Greibach Normal Form(GNF)

For any CFG, we can construct equivalent grammars in CNF and GNF.

Chomsky normal form

Any non-empty context-free language without ε, has a grammar G all of whose productions are of the form

1. A → BC, or
2. A → a

Furthermore, G has no useless symbols.

Note: If the language has ǫ, then we can get an “almost” Chomsky Normal Form grammar for it, by adding the rule S′ → S|ε to it.

Preliminary Ideas

Given a grammar G = (V, T, S, P), we need to produce a grammar G1 that is in Chomsky Normal Form.

Let G′ = (V ′, T, S, P′) be the grammar obtained after eliminating ε -productions, unit productions, and useless symbols from G.

If A → x is a rule of G′, where x Є V ′ U T, then x Є T, because G′ has no unit productions. So A → x is in correct form.

All remaining productions are of form A → X1X2 ⋅ ⋅ ⋅Xn where Xi Є V ′ U T, n ≥ 2. We will put these rules in the right form by applying the following two transformations:

1. Make all the bodies of these rules to consist only of variables
2. Make all the right hand sides of length 2.

Removing Terminals from Long Bodies

Let A → X1X2 ⋅ ⋅ ⋅Xn with Xi being either a variable or a terminal. We want have rules where all the Xi are variables.

Consider A → BbCdefG. How do you remove the terminals?

Solution: For each a, b, c . . . Є T add variables X_a,X_b,X_c, . . . with productions $X_a \rightarrow a$, $X_b \rightarrow b$, ... Then replace the production A → BbCdefG by A → $BX_bCX_dX_eX_fG$

Reducing right-hand sides to length 2

Now all productions are of the form A → a or A → B1B2 ⋅ ⋅ ⋅Bn, where n ≥ 2 and B_i is a variable.

How do you eliminate rules of the form A → B_1B_2 . . .B_n where n > 2? Replace the rule by the following set of rules

A → $B_1B_{(2,n)}$

$B_{(2,n)} \rightarrow B_2B_{(3,n)}$

$B_{(3,n)} \rightarrow B_3B_{(4,n)}$

...

$B_{(n-1,n)} \rightarrow B_{n-1}B_n$

where $B_{(i,n)}$ are “new” variables.

Example 1:

Convert: S → aA|bB|b, A → Baa|ba, B → bAAb|ab, into Chomsky Normal Form.

Step 1: Eliminate ǫ-productions, unit productions, and useless symbols.

This grammar is already in the right form.

Step 2: Remove terminals from the bodies of long rules. New grammar is:

Xa → a, Xb → b, S → XaA|XbB|b, A → BXaXa|XbXa, and B → XbAAXb|XaXb

Step 3: Reduce the right-hand side of rules to be of length at most 2.

New grammar replaces A → BXaXa by rules A → BXaa, Xaa → XaXa,

and B → XbAAXb by rules B → XbXAAb, XAAb → AXAb, XAb → AXb

Example 2:

$S \to aB|bA$

$A \to a|aS|bAA$

$B \to b|bS|aBB$

$$
\begin{array}{llll}
A \to a & S \to aB & : & S \to A_1B \\
B \to b & & & A_1 \to a \\
 & S \to bA & : & S \to B_1A \\
 & & & B_1 \to b \\
 & A \to aS & : & A \to A_2a \\
 & & & A_2 \to a \\
 & A \to bAA & : & A \to B_2A_3 \\
 & & & B_2 \to b \\
 & & & A_3 \to AA \\
 & B \to bS & : & B \to B_3S \\
 & & & B_3 \to b \\
 & B \to aBB & : & B \to A_4b_4 \\
 & & & A_4 \to a \\
 & & & B_4 \to BB
\end{array}
$$

Example 3: Consider the *CFG*: $S \to aSb|\in$ generating the language $L=\{a_nb_n \mid n \geq 0\}$. we will construct a *CNF* to generate the language $L-\{\in\}$ i.e. $\{a_nb_n \mid n \geq 1\}$.

- **Solutions :** We first eliminate $\in$ -productions (generating the language $\{a_nb_n \mid n \geq 1\}$) using the procedure already described to get $S \to aSb|ab$.
- ***Step* 1 :** Introduce nonterminals *A*, *B* and replace these productions with $S \to ASB|\ AB,\ A \to a,\ B \to b$
- ***Step* 2 :** Introduce nonterminal *C* and replace the only production $S \to ASB$ (which is not allowable form in *CNF*) with $S \to AC$ and $C \to SB$
- The final grammar in CNF is now

- $S \rightarrow AC|AB\ C$
  $\rightarrow SB$
  $A \rightarrow a$
  $B \rightarrow b$

Greibach normal form

In computer science, to say that a context-free grammar is in **Greibach normal form** (GNF) means that all production rules are of the form:

or

where A is a nonterminal symbol, α is a terminal symbol, X is a (possibly empty) sequence of nonterminal symbols not including the start symbol, S is the start symbol, and λ is the null string.

Observe that the grammar must be without left recursions.

Every context-free grammar can be transformed into an equivalent grammar in Greibach normal form. (Some definitions do not consider the second form of rule to be permitted, in which case a context-free grammar that can generate the null string cannot be so transformed.) This can be used to prove that every context-free language can be accepted by a non-deterministic pushdown automaton.

Given a grammar in GNF and a derivable string in the grammar with length n, any top-down parser will halt at depth n.

Greibach normal form is named after Sheila Greibach.

Example : $A \rightarrow BB\ B \rightarrow AC|\ a\ C \rightarrow AB / BA / a$. We will construct an equivalent CFG in GNF.

Step 1: Renaming the nonterminal, we get

$A_1 \rightarrow A_2 A_2$
$A_2 \rightarrow A_1 A_3 \mid a$
$A_3 \rightarrow A_1 A_2 \mid A_2 A_1 \mid a$

Step 2 : A_1-productions already satisfy INP. Process A_2- and A_3-productions to enforce the INP. First consider A_2-productions:

Apply lemma 2 to $A_2 \rightarrow A_1 A_3$ obtaining $A_2 \rightarrow A_2 A_2 A_3 \mid a$. Now apply lemma 1 to eliminate left-recursion

We get

$A_2 \rightarrow a \mid A_{-2}$
$A_{-2} \rightarrow A_2 A_3 \mid A_2 A_3 A_{-2}$

which satisfy the INP property.

The resulting grammar is

$A_1 \rightarrow A2\,A2$
$A_2 \rightarrow a \mid aA_{-2}$
$A_{-2} \rightarrow A_2 A_3 \mid A_2 A_3 A_{-2}$
$A_3 \rightarrow A_1 A_2 \mid A_2 A_1 \mid a$

Next consider A_3-productions. Applying lemma 2 to $A_3 \rightarrow A_1 A_2$ we get

$A_3 \rightarrow A_2 A_2 A_2 \mid A_2 A_1 \mid a$

Applying lemma 2 again on the first two A_3-productions above we get

$A_3 \rightarrow aA_2 A_2 \mid aA_{-2} A_2 A_2 \mid aA_1 \mid aA_{-2} A_1 \mid a$

Now, all productions satisfy the INP.

The resulting grammar is:

$A_1 \to A_2A_2 \mid$
$A_2 \to a \mid aA_{-2}$
$A_{-2} \to A_2A_3 \mid A_2A_3A_{-2}$
$A_3 \to aA_2A_2 \mid aA_{-2}A_2A_2 \mid aA_1 \mid aA_{-2}A_1 \mid a$

Step 3 : All A_3 -productions and A_2 -productions are already in GNF. Apply lemma 2 to A_1 -productions, to get $A_1 \to aA_2 \mid aA_{-2}A_2$.

Similarly, applying lemma 2 to A_{-2} -production we get

$A_{-2} \to aA_3 \mid aA_{-2}A_3 \mid aA_3A_{-2} \mid aA_{-2}A_3A_{-2}$

All the productions are in GNF now. So, the resulting equivalent grammar in GNF is

$A_1 \to aA_2 \mid aA_{-2}A_2$
$A_{-2} \to aA_3 \mid aA_{-2}A_3 \mid aA_3A_{-2} \mid aA_{-2}A_3A_{-2}$
$A_2 \to a \mid aA_{-2}$
$A_3 \to aA_2A_2 \mid aA_{-2}A_2A_2 \mid aA_1 \mid aA_{-2}A_1 \mid a$

Pumping Lemma for Context free languages

The pumping lemma for context-free languages gives a method of generating an infinite number of strings from a given sufficiently long string in a context-free language L. It is used to prove that certain languages are not context-free. The construction we make use of in proving pumping lemma yields some decesion algorithms regarding context-free languages.

Lemma 6.3 Let G be a context-free grammar in CNF and T be a derivation tree in G. If the length of the longest path in T is less than or equal to k, then the yield of T is of length less than or equal to 2^{k-1}.

Proof We prove the result by induction on k, the length of the longest path for all A-trees (Recall an A-tree is a derivation tree whose root has label A).

When the longest path in an A-tree is of length 1, the root has only one son whose label is a terminal (when the root has two sons, the labels are variables). So the yield is of length 1. Thus, there is basis for induction.

Assume the result for $k - 1$ ($k > 1$). Let T be an A-tree with a longest path of length less than or equal to k. As $k > 1$, the root of T has exactly two sons with labels A_1 and A_2. The two subtrees with the two sons as roots have the longest paths of length less than or equal to $k - 1$ (see Fig. 6.12).

If w_1 and w_2 are their yields, then by induction hypothesis, $|w_1| \leq 2^{k-2}$, $|w_2| \leq 2^{k-2}$. So the yield of $T = w_1w_2$, $|w_1w_2| \leq 2^{k-2} + 2^{k-2} = 2^{k-1}$. By the principle of induction, the result is true for all A-trees, and hence for all derivation trees.

Theorem 6.10 (Pumping lemma for context-free languages). Let L be a context-free language. Then we can find a natural number n such that:

(i) Every $z \in L$ with $|z| \geq n$ can be written as $uvwxy$ for some strings u, v, w, x, y.

(ii) $|vx| \geq 1$.

(iii) $|vwx| \leq n$.

(iv) $uv^k wx^k y \in L$ for all $k \geq 0$.

Proof By Corollary 1 of Theorem 6.6, we can decide whether or not $\Lambda \in L$. When $\Lambda \in L$, we consider $L - \{\Lambda\}$ and construct a grammar $G = (V_N, \Sigma, P, S)$ in CNF generating $L - \{\Lambda\}$ (when $\Lambda \notin L$, we construct G in CNF generating L).

Let $|V_N| = m$ and $n = 2^m$. To prove that n is the required number, we start
with $z \in L$, $|z| \geq 2^m$, and construct a derivation tree T (parse tree) of z. If the length of a longest path in T is at most m, by Lemma 6.3, $|z| \leq 2^{m-1}$ (since z is the yield of T). But $|z| \geq 2^m > 2^{n-1}$. So T has a path, say Γ, of length greater than or equal to $m + 1$. Γ has at least $m + 2$ vertices and only the last vertex is a leaf. Thus in Γ all the labels except the last one are variables. As $|V_N| = m$, some label is repeated.

We choose a repeated label as follows: We start with the leaf of Γ and travel along Γ upwards. We stop when some label, say B, is repeated. (Among several repeated labels, B is the first.) Let v_1 and v_2 be the vertices with label B, v_1 being nearer the root. In Γ, the portion of the path from v_1 to the leaf has only one label, namely B, which is repeated, and so its length is at most $m + 1$.

Let T_1 and T_2 be the subtrees with v_1, v_2 as roots and z_1, w as yields, respectively. As Γ is a longest path in T, the portion of Γ from v_1 to the leaf is a longest path in T_1 and of length at most $m + 1$. By Lemma 6.3, $|z_1| \leq 2^m$ (since z_1 is the yield of T_1).

For better understanding, we illustrate the construction for the grammar whose productions are $S \to AB$, $A \to aB \mid a$, $B \to bA \mid b$, as in Fig. 6.13. In the figure,

$$\Gamma = S \to A \to B \to A \to B \to b$$
$$z = ababb, \quad z_1 = bab, \quad w = b$$
$$v = ba, \quad x = \Lambda, \quad u = a, \quad y = b$$

Proof (i) We have to prove the 'only if' part. If $z \in L$ with $|z| \geq n$, we apply the pumping lemma to write $z = uvwxy$, where $1 \leq |vx| \leq n$. Also, $uwy \in L$ and $|uwy| < |z|$. Applying the pumping lemma repeatedly, we can get $z' \in L$ such that $|z'| < n$. Thus (i) is proved.

(ii) If $z \in L$ such that $n \leq |z| < 2n$, by pumping lemma we can write $z = uvwxy$. Also, $uv^k wx^k y \in L$ for all $k \geq 0$. Thus we get an infinite number of elements in L. Conversely, if L is infinite, we can find $z \in L$ with $|z| \geq n$. If $|z| < 2n$, there is nothing to prove. Otherwise, we can apply the pumping lemma to write $z = uvwxy$ and get $uwy \in L$. Every time we apply the pumping lemma we get a smaller string and the decrease in length is at most n (being equal to $|vx|$). So, we ultimately get a string z' in L such that $n \leq |z'| < 2n$. This proves (ii). ∎

Note: As the proof of the corollary depends only on the length of vx, we can apply the corollary to regular sets as well (refer to pumping lemma for regular sets).

The corollary given above provides us algorithms to test whether a given context-free language is empty or infinite. But these algorithms are not efficient. We shall give some other algorithms in Section 6.6.

We use the pumping lemma to show that a language L is not a context-free language. We assume that L is context-free. By applying the pumping lemma we get a contradiction.

The procedure can be carried out by using the following steps:

Step 1 Assume L is context-free. Let n be the natural number obtained by using the pumping lemma.

Step 2 Choose $z \in L$ so that $|z| \geq n$. Write $z = uvwxy$ using the pumping lemma.

Step 3 Find a suitable k so that $uv^k wx^k y \notin L$. This is a contradiction, and so L is not context-free.

Example 1

Show that $L = \{a^n b^n c^n \mid n \geq 1\}$ is not context-free but context-sensitive.

Solution

We have already constructed a context-sensitive grammar G generating L (see Example 4.11). We note that in every string of L, any symbol appears the same number of times as any other symbol. Also a cannot appear after b, and c cannot appear before b, and so on.

Step 1 Assume L is context-free. Let n be the natural number obtained by using the pumping lemma.

Step 2 Let $z = a^n b^n c^n$. Then $|z| = 3n > n$. Write $z = uvwxy$, where $|vx| \geq 1$, i.e. at least one of v or x is not Λ.

(iii) *Algorithm for deciding whether a regular language L is empty.* Construct a deterministic finite automaton M accepting L. We construct the set of all states reachable from the initial state q_0. We find the states which are reachable from q_0 by applying a single input symbol. These states are arranged as a row under columns corresponding to every input symbol. The construction is repeated for every state appearing in an earlier row. The construction terminates in a finite number of steps. If a final state appears in this tabular column, then L is nonempty. (Actually, we can terminate the construction as soon as some final state is obtained in the tabular column.) Otherwise, L is empty.

(iv) *Algorithm for deciding whether a regular language L is infinite.* Construct a deterministic finite automaton M accepting L. L is infinite if and only if M has a cycle.

Applications of PDA – Parsing

In a natural language, parsing is the process of splitting a sentence into words. There are two types of parsing, namely the top-down parsing and the bottom-up parsing. Suppose we want to parse the sentence "Ram ate a mango." If NP, VP, N, V, ART denote noun predicate, verb predicate, noun, verb and article, then the top-down parsing can be done as follows:

$$
\begin{aligned}
S &\rightarrow \text{NPVP} \\
&\rightarrow \text{Name VP} \\
&\rightarrow \text{Ram V NP} \\
&\rightarrow \text{Ram ate ART N} \\
&\rightarrow \text{Ram ate a N} \\
&\rightarrow \text{Ram ate a mango}
\end{aligned}
$$

The bottom-up parsing for the same sentence is

$$
\begin{aligned}
\text{Ram ate a mango} &\rightarrow \text{Name ate a mango} \\
&\rightarrow \text{Name verb a mango} \\
&\rightarrow \text{Name V ART N} \\
&\rightarrow \text{NP VN P} \\
&\rightarrow \text{NP VP} \\
&\rightarrow S
\end{aligned}
$$

In the case of formal languages, we derive a terminal string in $L(G)$ by applying the productions of G. If we know that $w \in \Sigma^*$ in $L(G)$, then $S \overset{*}{\Rightarrow} w$. The process of the reconstruction of the derivation of w is called parsing. Parsing is possible in the case of some context-free languages.

Parsing becomes important in the case of programming languages. If a statement in a programming language is given, only the derivation of the statement can give the meaning of the statement. (This is termed *semantics*.)

As mentioned earlier, there are two types of parsing: top-down parsing and bottom-up parsing.

In top-down parsing, we attempt to construct the derivation (or the corresponding parse tree) of the input string, starting from the root (with label S) and ending in the given input string. This is equivalent to finding a leftmost derivation. On the other hand, in bottom-up parsing we build the derivation from the given input string to the top (root with label S).

In this section we present certain techniques for top-down parsing which can be applied to a certain subclass of context-free languages. We illustrate them by means of some examples. We discuss LL(l) parsing, LL(k) parsing, left factoring and the technique to remove left recursion.

EXAMPLE 7.10

Let $G = (\{S, A, B\}, \{a, b\}, P, S)$ where P consists of $S \to aAB$, $S \to bBA$, $A \to bS$, $A \to a$, $B \to as$, $B \to b$. $w = abbbab$ is in $L(G)$. Let us try to get a leftmost derivation of w. When we start with S we have two choices: $S \to aAB$ and $S \to bBA$. By looking at the first symbol of w, we see that $S \to bBA$ will not yield w. So we choose $S \to aAB$ as the production to be applied in step 1 and we get $S \Rightarrow aAB$. Now consider the leftmost variable A in the sentential form aAB. We have to apply an A-production among the productions $A \to bS$ and $A \to a$. $A \to a$ will not yield w subsequently since the second symbol in w is b. So, we choose $A \to bS$ and get $S \Rightarrow aAB \Rightarrow abSB$. Also, the substring ab of w is a substring of the sentential form $abSB$. By looking ahead for one symbol, namely the symbol b, we decide to apply $S \to bBA$ in the third step. This leads to $S \Rightarrow aAB \Rightarrow abSB \Rightarrow abbBAB$. The leftmost variable in the sentential form $abbBAB$ is B. By looking ahead for one symbol which is b, we apply the B-production $B \to b$ in the fourth step. On similar considerations, we apply $A \to a$ and $B \to b$ in the last two steps to get the leftmost derivation.

$$S \Rightarrow aAB \Rightarrow abSB \Rightarrow abbBAB \Rightarrow abbbAB \Rightarrow abbbaB \Rightarrow abbbab$$

Thus in the case of the given grammar, we are able to construct a leftmost derivation of w by looking ahead for one symbol in the input string. In order to do top-down parsing for a general string in $L(G)$, we prepare a table called the *parsing table*. The table provides the production to be applied for a given variable with a particular look ahead for one symbol.

For convenience, we denote the productions S $\to aAB$, $S \to bBA$, $A \to bS$, $A \to a$, $B \to aS$ and $B \to b$ by P_1, P_2, . . ., P_6. Let E denote an error. It indicates that the given input string is not in $L(G)$. The table for the given grammar is given in Table 7.1.

TABLE 7.1 Parsing Table for Example 7.10

	Λ	a	b
S	E	P_1	P_2
A	E	P_4	P_3
B	E	P_5	P_6

For example, if A is the leftmost variable in a sentential form and the first symbol in unprocessed substring of the given input string is b, then we have to apply P_3.

A grammar possessing this property (by looking ahead for one symbol in the input string we can decide the production to be applied in the next step) is called an LL(1) grammar.

EXAMPLE 7.11

Let G be a context-free grammar having the productions $S \to F + S$, $S \to F * S$, $S \to F$ and $F \to a$. Consider $w = a + a * a$. This is a string in $L(G)$. Let us try to get the top-down parsing for w.

Looking ahead for one symbol will not help us. For the string $a + a * a$, we can apply $F \to a$ on seeing a. But if a is followed by $+$ or $*$, we cannot apply a. So in this case it is necessary to look ahead for two symbols.

When we start with S we have three productions $S \to F + S$, $S \to F * S$ and $S \to F$. The first two symbols in $a + a * a$ are $a +$. This forces us to apply only $S \to F + S$ and not other S-productions. So, $S \to F + S$. We can apply $F \to a$ now to get $S \Rightarrow F + S \Rightarrow a + S$. Now the remaining part of w is $a * a$. The first two symbols $a *$ suggest that we apply $S \to F * S$ in the third step. So, $S \overset{*}{\Rightarrow} a + S \Rightarrow a + F * S$. As the third symbol in w is a, we apply $F \to a$, yielding $S \overset{*}{\Rightarrow} a + F * S \Rightarrow a + a * S$. The remaining part of the input string w is a. So, we have to apply $S \to F$ and $F \to a$. Thus the leftmost derivation of $a + a * a$ is $S \Rightarrow F + S \Rightarrow a + S \Rightarrow a + F * S \Rightarrow a + a * S \Rightarrow a + a * F \Rightarrow a + a * a$.

As in Example 7.10, we can prepare a table (Table 7.2) which enables us to get a leftmost derivation for any input string. P_1, P_2, P_3 and P_4 denote the productions $S \to F + S$, $S \to F * S$, $S \to F$ and $F \to a$. E denotes an error.

TABLE 7.2 Parsing Table for Example 7.11

	Λ	a	$+$	$*$	aa	$a+$	$a*$
S	E	P_3	E	E	E	P_1	P_2
F	E	P_4	E	E	E	P_4	P_4
	$+a$	$++$	$+*$	$*a$	$*+$	$**$	
S	E	E	E	E	E	E	
F	E	E	E	E	E	E	

For example, if the leftmost variable in a sentential form is F and the next two symbols to be processed are $a *$, then we apply P_4, i.e. $F \to a$. When we encounter $* a$ as the next two symbols, an error is indicated in the table and so the input string is not in $L(G)$.

A grammar G having the property (by looking ahead for k symbols we derive a given input string in $L(G)$), is called an LL(k) grammar. The grammar given in Example 7.11 is an LL(2) grammar.

In Examples 7.10 and 7.11 for getting a leftmost derivation, one production among several choices was obtained by look ahead for k symbols. This kind of nondeterminism cannot be resolved in some grammars even by looking ahead.

This is the case when a grammar has two A-productions of the form $A \to \alpha\beta$ and $A \to \alpha\gamma$. By a technique called 'left factoring', we resolve this nondeterminism. Another troublesome phenomenon in a context-free grammar which creates a problem is called left recursion. A variable A is called left recursive if there is an A-production of the form $A \to A\alpha$. Such a production can cause a top-down parser into an infinite loop. Left factoring and technique for avoiding left recursion are provided in Theorems 7.6 and 7.7.

Theorem 7.6 Let G be a context-free grammar having two A-productions of the form $A \to \alpha\beta$ and $A \to \alpha\gamma$. If $A \to \alpha\beta$ and $A \to \alpha\gamma$ are replaced by $A \to \alpha A'$. $A' \to \beta$ and $A' \to \gamma$, where A' is a new variable then the resulting grammar is equivalent to G.

Proof The equivalence can be proved by showing that the effect of applying $A \to a\beta$ and $A \to \alpha\gamma$ in a derivation can be realised by applying $A \to \alpha A'$, $A' \to \beta$ and $A' \to \gamma$ and vice versa.

Note: The technique of avoiding nondeterminism using Theorem 7.6 is called left factoring.

Theorem 7.7 Let G be a context-free grammar. Let the set of all A-productions be $\{A \to A\alpha_1, \ldots, A \to A\alpha_n, A \to \beta_1, \ldots, A \to \beta_m\}$. Then the grammar G' obtained by introducing a new variable A' and replacing all A-productions in G by $A \to \beta_1 A', \ldots, A \to \beta_m A', A' \to \alpha_1 A', \ldots, A' \to \alpha_n A'$ and $A' \to \Lambda$ is equivalent to G.

Proof Similar to proof of Lemma 6.3.

Theorems 7.6 and 7.7 are useful to construct a top-down parser only for certain context-free grammars and not for all context-free grammars. We summarize our discussion as follows:

Construction of Top-Down Parser

Step 1 Eliminate left recursion in G by repeatedly applying Theorem 7.7 to all left recursive variables.

Step 2 Apply Theorem 7.6 to get left factoring wherever necessary.

Step 3 If the resulting grammar is LL(k) for some natural number k, apply top-down parsing using the techniques explained in Examples 7.10 and 7.11.

EXAMPLE 7.12

Consider the language consisting of all arithmetic expressions involving +, *, (and) over the variables $x1$ and $x2$. This language is generated by a grammar

$G = (\{T, F, E\}\ \Sigma, P, E)$, where $\Sigma = \{x, 1, 2, +, *, (,)\}$ and P consists of

$$E \to E + T \qquad F \to (E)$$
$$E \to T \qquad F \to x1$$
$$T \to T * F \qquad F \to x2$$
$$T \to F$$

Let us construct a top-down parser for $L(G)$.

Step 1 We eliminate left recursion by applying Theorem 7.7 to the left recursive variables E and T. We replace $E \to E + T$ and $E \to T$ by $E \to TE'$, $E' \to +TE'$ and $E' \to \Lambda$ (E' is a new variable). Similarly, $T \to T * F$ and $T \to F$ are replaced by $T \to FT'$, $T' \to *FT'$ and $T' \to \Lambda$. The resulting equivalent grammar is

$G_1 = (\{T, F, E, T', E'\}, \Sigma, P', E)$, where P' consists of

$$E \to TE' \qquad T \to \Lambda$$
$$E' \to +TE' \qquad F \to (E)$$
$$E' \to \Lambda \qquad F \to x1$$
$$T \to FT' \qquad F \to x2$$
$$T' \to *FT'$$

Step 2 We apply Theorem 7.6 for left factoring to $F \to x1$ and $F \to x2$ to get new productions $F \to xN \to N \to 1$ and $N \to 2$.

The resulting equivalent grammar is

$G_2 = (\{T, F, E, T', E'\}, \Sigma, P'', E)$ where P'' consists of

$$P_1: E \to TE' \qquad P_6: T' \to \Lambda$$
$$P_2: E' \to +TE' \qquad P_7: F \to (E)$$
$$P_3: E' \to \Lambda \qquad P_8: F \to xN$$
$$P_4: T \to FT' \qquad P_9: N \to 1$$
$$P_5: T'' \to *FT' \qquad P_{10}: N \to 2$$

Step 3 The grammar G_2 obtained in step 2 is an LL(1) grammar. The parsing table is given in Table 7.3.

TABLE 7.3 Parsing Table for Example 7.12

	Λ	x	1	2	+	*	(	)
E	E	P_1	E	E	E	E	P_1	E
T	E	P_4	E	E	E	E	P_4	E
F	E	P_8	E	E	E	E	P_7	E
T'	P_6	E	E	E	E	P_5	E	P_6
E'	P_3	E	E	E	P_2	E	E	P_3
N	E	E	P_9	P_{10}	E	E	E	E

We have seen that pda's are the accepting devices for context-free languages. Theorem 7.3 gives us a method of constructing a pda accepting a given context-free language by empty store. In certain cases the construction can be modified in such a way that a leftmost derivation of a given input string can be obtained while testing to know whether the given string is accepted by the pda. This is the case when the given grammar is LL(1). We illustrate this by constructing a (deterministic) pda accepting the language given in Example 7.10 and a leftmost derivation of a given input string using the pda.

EXAMPLE 7.13

For the grammar given in Example 7.10, construct a deterministic pda accepting $L(G)$ and a leftmost derivaiton of *abbab*.

Solution

We construct a pda accepting $L(G)\$$ ($\$ is a symbol indicating the end of the input string). This is done by using Theorem 7.3. The transitions are

$$\delta(q, \Lambda, A) = \{(q, \alpha) \mid A \to \alpha \text{ is in } P\}$$

$$\delta(q, t, t) = \{(q, \Lambda)\} \quad \text{for every } t \text{ in } \Sigma$$

This pda is not deterministic as we have two S-productions, two A-productions, etc. In Example 7.10 we resolved the nondeterminism by looking ahead for one more symbol in the input string to be processed. In the construction of pda this can be achieved by changing the state from q to q_a on reading a. When the pda is in state q_a and the current symbol is S we choose the transition resulting in (q, aAB). Now the deterministic pda accepting $L(G)\$$ by null store is

$$A = (\{p, q, q_a, q_b\}, \{a, b, \$\}, \{S, A, B, a, b, Z_0\}, \delta, p, Z_0, \emptyset)$$

where δ is defined by the following rules:

$$R_1 : \delta(p, \Lambda, Z_0) = (q, s)$$

$$R_2 : \delta(q, a, \Lambda) = (q_a, \Lambda)$$

$$R_3 : \delta(q_a, \Lambda, a) = (q, e)$$

$$R_4 : \delta(q, b, \Lambda) = (q_b, \Lambda)$$

$$R_5 : \delta(q_a, \Lambda, b) = (q, e)$$

$$R_6 : \delta(q_a, \Lambda, S) = (q_a, aAB)$$

$$R_7 : \delta(q_b, \Lambda, S) = (q_b, bBA)$$

$$R_8 : \delta(q_a, \Lambda, A) = (q_a, a)$$

$$R_9 : \delta(q_b, \Lambda, A) = (q_b, bS)$$

R_{10} : $\delta(q_a, \Lambda, B) = (q_a, aS)$

R_{11} : $\delta(q_b, \Lambda, B) = (q_b, b)$

R_{12} : $\delta(q, \$, Z_0) = (q, \Lambda)$

Here R_1 changes the initial ID (p, w, Z) into (q, w, SZ). R_2 and R_4 are for remembering the next symbol. R_6–R_{11} are simulating the productions. R_3 and R_5 are for matching the current input symbol and the topmost symbol on PDS and for erasing it (in PDS). Finally, R_{12} is a move for erasing Z and making the PDS empty when the last symbol \$ of the input string is read.

To get a leftmost derivation for an input string w, apply the unique transition given by R_1 to R_{12}. When we apply R_6 to R_{11}, we are using a corresponding production. By recording these productions we can test whether $w \in L(G)$ and get a leftmost derivation. The parsing for the input string *abbbab* is given m Table 7.4.

The last column of Table 7.4 gives us a leftmost derivation of *abbbab*. It is $S \Rightarrow aAB \Rightarrow abSB \Rightarrow abbBAB \Rightarrow abbbAB \Rightarrow abbbaB \Rightarrow abbbab$.

TABLE 7.4 Top-down Parsing for *w* of Example 7.13

Step	State	Unread input	Pushdown stack	Transition used	Production applied
1	p	abbbab\$	Z_0	—	—
2	q	abbbab\$	SZ_0	R_1	
3	q_a	bbbab\$	SZ_0	R_2	
4	q_a	bbbab\$	$aABZ_0$	R_6	$S \rightarrow aAB$
5	q	bbbab\$	ABZ_0	R_3	
6	q_b	bbab\$	ABZ_0	R_4	
7	q_b	bbab\$	$bSBZ_0$	R_9	$A \rightarrow bS$
8	q	bbab\$	SBZ_0	R_5	
9	q_b	bab\$	SBZ_0	R_4	
10	q_b	bab\$	$bBABZ_0$	R_7	$S \rightarrow bBA$
11	q	bab\$	$BABZ_0$	R_5	
12	q_b	ab\$	$BABZ_0$	R_4	
13	q_b	ab\$	$bABZ_0$	R_{11}	$B \rightarrow b$
14	q	ab\$	ABZ_0	R_5	
15	q_a	b\$	ABZ_0	R_2	
16	q_a	b\$	aBZ_0	R_8	$A \rightarrow a$
17	q	b\$	BZ_0	R_3	
18	q_b	\$	BZ_0	R_4	
19	q_b	\$	bZ_0	R_{11}	$B \rightarrow b$
20	q	\$	Z_0	R_5	
21	q	Λ	Λ	R_{12}	

Bottom-up Parsing

In bottom-up parsing we build the derivation tree from the given input string to the top (the root with label S). For certain classes of grammars, called weak precedence grammars, we can construct a deterministic pda which acts as a bottom-up parser. We illustrate the method by constructing the parser for the grammar given in Example 7.12.

In bottom-up parsing we have to reverse the productions to get S finally. This suggests the following moves for a pda acting as bottom-up parser.

(i) $\delta(p, \Lambda, \alpha^T) = \{(P, A) | \text{there exists a production } A \to \alpha\}$

(ii) $\delta(p, \sigma, \Lambda) = \{(p, \sigma)\}$ for all σ in Σ.

Using (i) we replace α^T on the basis by A when $A \to \alpha$ is a production. The input symbol σ is moved onto the stack using (ii). For acceptability, we require some moves when the PDS has S or Z_0 on the top.

As in top-down parsing we construct the pda accepting $L(G)\$$. Here we will have two types of operations, namely shifting and reducing. By shifting we mean pushing the input symbol onto the stack (moves given by (ii)). By reducing we mean replacing α^T by A when $A \to \alpha$ is a production in G (moves given by (i)).

At every step we have (i) to decide whether to shift or to reduce (ii) to choose the prefix of the string on PDS for reducing, once we have decided to reduce. For (i) we use a relation P called a precedence relation. If $(a, b) \in P$ where a is the topmost symbol on PDS and b is the input symbol then we reduce. Otherwise we shift b onto the stack. Regarding (ii), we choose the longest prefix of the string on the PDS of the form α^T to be reduced to A (when $A \to \alpha$ is a production).

We illustrate the method using the grammar given in Example 7.12.

Introduction to Turing Machine

Neither Finite Automata(FA) nor Pushdown Automata(PDA) can be regarded as truly general models for computers, since they are not capable of recognizing some type of languages, such as $\{a^n b^n c^n / n>=0\}$. So the next model of automata is Turing Machine.

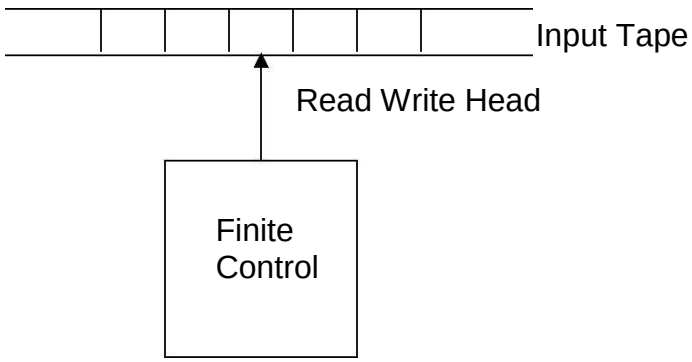
In 1930's several mathematicians began to think about what it means to be able to compute a 'function'. Alan M. Turing and Alenzo Church arrived at same conclusion: "a function is computable if it can be computed by a Turing machine".

Alan Turing proposed the Turing Machine as a model of 'any possible computation'. Turing Machines are more powerful than PDA. This can do general purpose computations. Church –Turing thesis that claims that there exists no model of computation which is more powerful than the Turing Machine.

Turing Machine

Basic model of a Turing machine consists of

ix) a two way infinite tape,
x) a read/write head and
xi) a finite control.



At any time, action of a Turing machine depends on the current state and the input symbol and involves (i) change of state (ii) writing a symbol in the cell scanned (iii) head movement to the left or right and (iv) Turing machine halts or not halts. A Turing machine may utilize the tape cells beyond the input limits and 'Blank' cell plays a significant role in the working of a Turing machine. Turing machine halts in any situation for which a transition is not defined. Unlike the previously dealt automata, it is possible that a Turing machine may not halt. At any state a Turing machine can halt or not halt. ie, it ends in accepting state if it successfully halts(accept halt). Otherwise it halts in any non accepting state (reject halt).

Formal Definition

A Turing machine can be formally defined as M=(Q,Σ,Γ,δ,q0,B,F)

All symbols are same except that Γ,δ,B. Here Γ is a finite non empty set of tape symbols or tape alphabets, B is the special blank symbol and B ∈ Γ and δ is the transition function defined as,

$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R\}$

{L,R} represents the movement of the head.

Language Acceptance by Turing Machine

A string w is accepted by a Turing machine if $q_0 w \vdash^* \alpha_1 q \alpha_2$ for some 'q' in F.
and $\alpha_1, \alpha_2 \in \Gamma^*$, and Turing machine has no further move.

M does not accept 'w' if the machine M either halts in non accepting state or does not halt. Language accepted by a Turing machine is defined as

$L(M)=\{w q_0 w \vdash^* \alpha_1 q \alpha_2$ for some q in F and there is no further move$\}$

Representation of a Turing Machine

We can describe a Turing machine using

(iii) instantaneous descriptions using move relations

Instantaneous Description (ID) in PDA was in terms of current state, input string to be processed, and topmost symbol of PDS. But in Turing machine the R/W head can move to the left also. So ID of a Turing machine is defined in terms of the entire input string and current state. During a specific execution, configuration of the Turing machine decides further behavior consists of (i) state (ii) non-blank portion of the tape content to the left of the head (iii) non-blank portion of the tape content to the right of the head. This
is called instantaneous description (ID) and is represented by $\alpha_1 q \alpha_2$. The head reads the leftmost symbol of α_2. If α_2 is empty, the head scans 'B'. Each move changes the Turing machine from one ID to another. The symbol |-- is used to represent the move. |--*-- represents 'sequence of moves' or 'reflexive transitive closure' of the relation |---

Let $\delta(q,x_i)=(p,y,R)$ then

$x_1 x_2 \ldots\ldots x_{i-1}\, q\, x_i \ldots\ldots x_n \vdash$

$x_1 x_2 \ldots\ldots x_{i-1}\, y\, p\, x_{i+1} \ldots\ldots x_n$

(iv) transition table

We give the definition of δ in the form of a table called the **transition table**. If $\delta(q,a)=(\gamma,\alpha,\beta)$, we write $\alpha\beta\gamma$ under a-column and q-row. So if we get $\alpha\beta\gamma$ in the table,

means that α is written in the current cell, β gives the movement of the head (L/R), and γ denotes the new state into which Turing machine enters.

Eg:

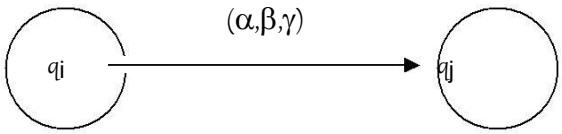
Present state	Tape symbols		
	0	**1**	**b**
q_1	0Rq_1		1Lq_2
q_2	0Lq_2	1Lq_2	bRq_3
q_3	bRq_4	bRq_5	
q_4	0Rq_4	1Rq_4	0Rq_5
*q_5			0Lq_2

(iii) transition diagram

In the transition diagram the labels are triples of the form (α,β,γ) where $\alpha,\beta \in \Gamma$ and $\gamma \in \{L,R\}$. When there is a directed edge from q_i to q_j with label (α,β,γ), it means that $\delta(q_i, \alpha)=(q_j, \beta, \gamma)$.

During the processing of an input string, suppose the Turing machine enters q_i and R/W head scans the present symbol α. As a result, the symbol β is written in the cell under R/W head. The R/W head moves to the left or right, depending on γ, and the new state is q_j.

Ie,



eg:

Design a Turing machine to recognize all strings consisting of even number of 1's. Solution: (i) q_1 is the initial state. M enters state q_2 on scanning 1 and writes b.

3) If M is in state q_2 and scans 1, it enters q_1 and writes b.

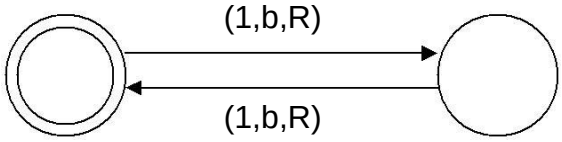
q_1 is the only accepting state.

So M accepts a string if it exhausts all input symbols and finally in state q_1. Symbolically,

$M=(\{q_1,q_2\},\{1\},\{1,b\}, \delta, q_1,b,\{q_1\})$ Where δ is defined by

Present state	Input symbols	
	1	**B**
*q_1	XRq_2	BLq_1
q_2	XRq_1	

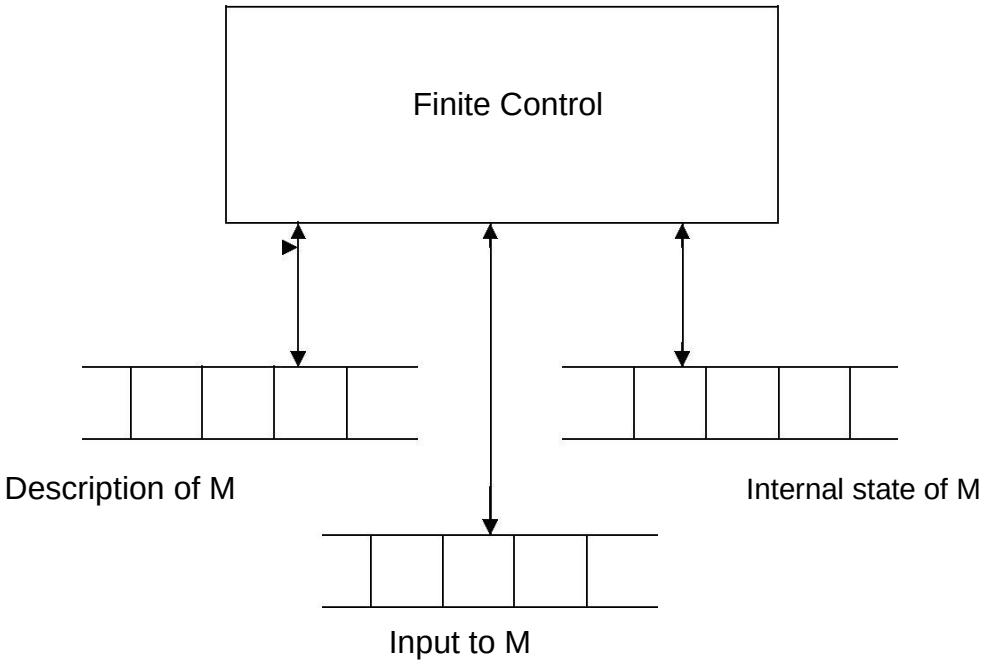
Let us obtain the computation sequence of 1111. Thus
$q_1$1111B |-- X q_2 111B |-- XX $q_1$11B |-- XXX $q_2$1B |-- XXXXq1B |--
XXXq1XB As q1 is accepting state. 1111 is accepted.



Universal Turing Machine

The Turing machine that was discussed for the design till now is special-purpose computers. Designing general purpose Turing machine is a more complex task. We must design a machine that can accept 2 inputs, (1) input data (2) description of computation (algorithm or program). This is precisely what a general-purpose computer does. It accepts data and program.

A general purpose Turing machine is called 'Universal Turing Machine (UTM)' when it is powerful enough to simulate the behavior of any computer including Turing machine itself. ie, a UTM can simulate the behavior of an arbitrary Turing machine over any Σ. So a UTM is analogous to general-purpose computer which can execute any given program.



Modifications of basic model of TM

The TM till now discussed are not the most efficient, but it is evident that even with very will-designed TMs, it will take a large number of states for simulating even a simple behavior. Thus we can modify our basic model by

(a) increasing the number of R/W heads
(b) making the tape 2D or 3D
(c) adding special purpose memory (stack/special purpose registers)

All these modifications will at most speed up the operation of the machine, but do not increase the computing power.

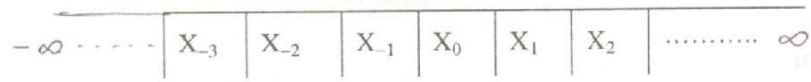
Variants of Turing Machine

1. Non-deterministic TM
2. One-way infinite tape TM
3. TM with ‘stay’ option
4. Multi tape TM
5. Multidimensional TM
6. Multi stack TM

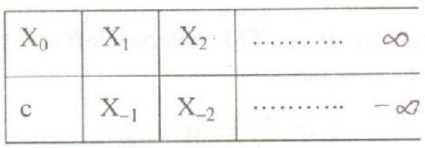
1. One-Way Infinite Tape TM

In one-way infinite tape TM, the left end is fixed. Further left movement is forbidden. However, as we show later, this restriction does not diminish the capability of a Turing machine. That is why, in some literature on this subject, one-way infinite TM is taken as the basic model.

Given a TM $M_1 = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, we can construct a one-way infinite TM M_2 to simulate the behaviour of M_1. Outline of construction is given below. Cells of a two-way infinite tape can be indexed by the set of integers $(-\infty, \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots, \infty)$ and content of i-th cell can be represented by X_i (as shown in the figure).



If we fold the tape at position 0, the resulting tape can be considered as a 2-track one-way infinite tape. Based on this idea, we will construct a one-way infinite TM M_2 to simulate M_1. M_2 has a two-track tape. The upper track is used to simulate the right portion of M_1 and lower track simulates the left portion. Lower track of the 0-th cell contains a marker c (which is used to switch over the action from one track to another). The arrangement is shown in the figure.



Finite control of M_2 has two parts. The first part is the actual control part to simulate M_1 and the second part indicates whether M_1 is on the right side (U) or left side (L).

Formally,

$M_2 = (Q', \Sigma, \Gamma', \delta, q_0', B, F')$

where,

$Q' = \{q_0'\} \cup Q \times \{L, U\}$,

$\Gamma' = \Gamma \times \Gamma \cup \{c\}$

δ' as defined by the following set of rules,

q_0' is the start state,

B is the blank symbol.

$F' = \{[p, X] | p \in F \text{ and } X = L \text{ or } U\}$.

2. Turing Machine with 'Stay' Option

In this model, head can move right or left or stay in the same position. A formal definition of a TM with stay option is similar to the standard model. The only change is in the transition function which is defined as $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, U, S\}$. However, this additional facility does not enhance the capability of the Turing machine. Given a TM M_1 (with stay option) $= (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ we can construct a standard TM M_2 to simulate the behaviour of M_1. For each transition with 'Stay', M_2 goes right and goes to a temporary state. From that temporary state, it gets back to its original state. Formally $M_2 = (Q', \Sigma, \Gamma, \delta', q_0, B, F)$ where,

1. $Q' = Q \cup \{q_i' | q_i \in Q\}$
   [Each state has a corresponding temporary state]
2. $\delta'(q, A) = \delta(q, A)$ if $\delta(q, A) = (q', X, D)$ where D = L or R.
3. If $\delta(q_i, A) = (q_j, Y, S)$ then δ' includes

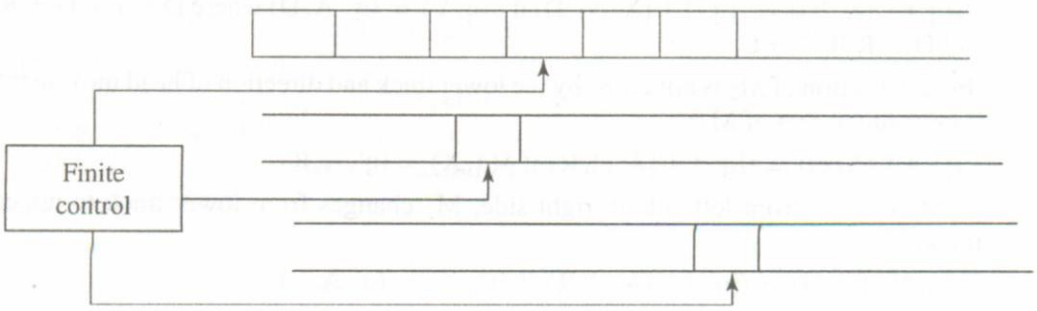
$$\delta'(q_i, A) = (q_j', Y, R) \text{ and}$$

$$\delta'(q_j', X) = (q_j, X, L) \text{ for all } X \in \Gamma.$$

   [In temporary state q_j', irrespective of the symbol scanned, M_2 moves left and goes to q_j].

3. Multi tape Turing Machine

We may add extra tapes to the basic model of Turing Machine. A multitape TM has k tapes each having its own read/write head (see the figure below). Action of the TM depends on the contents of the cells scanned and the current state and involves (i) change of state and (ii) writing on the cells scanned and moving the head (L,R or S) of each tape. Formally, transition function of a k-tape TM is $\delta = Q \times \Gamma^k \rightarrow Q \times (\Gamma \times D)^k$ where $D = \{L, R, S\}$.



Remark: Unlike multitrack TM, head movements are independent in multitape TM.

4. Non-Deterministic Turing Machine

In non-deterministic Turing machine, at each step, there is a choice of next move. Hence, the transition function is defined as

$$\delta: Q \times \Gamma \rightarrow 2^{Q \times \Gamma \times \{L,R\}}.$$

But, nondeterminism does not enhance the capability of a TM. Given any NDTM M_1, we can construct an equivalent DTM M_2. An outline of construction is given below.

Any nondeterministic computation can be considered as a tree with start-Id as the root. Branches at each node correspond to the choices at that stage.

Each path from the root corresponds to a specific sequence of choices. NDTM M_1 accepts the input if there is some path in the tree from root to an accepting ID. Simulating DTM M_2, makes a systematic search for the accepting ID in the computation tree. We use breadth first search for this purpose. Let h be the depth of the accepting ID. When we systematically enumerate ID's using BFS, ultimately we arrive at depth h and successfully reach the accepting ID. Outline of M_2 for this BFS is given below.

If, at any step, M_1 has k choices, each choice can be given an index i. Similarly, any path in the tree can be identified by a sequence number of the form $i_1 i_2 \ldots i_k$. Simulating DTM M_2 has 3 tapes. Tape 1 holds the input w. Tape 2 is used to generate a sequence of integers $i_1 i_2 \ldots i_k$. Tape 3 is used for actual simulation using the path as given by the sequence in tape 2. M_2 halts if the accepting ID is reached.

5. Multidimensional Turing Machine

As a model of computation, a tape is equivalent to memory. A tape in the standard model is linear. In actual computers, memory is organized as a multidimensional array. Correspondingly, we have multidimensional tape TMs. However, it can be shown that multidimensional TM is equivalent to the linear tape TM. We will show how a linear tape TM can simulate a 2-demensional TM.

Contents of a 2-dimensional TM can be considered as a sequence of lines and the symbol * is used to separate successive lines. The symbol ** is used to mark the ends. The arrangement is shown in the figure.

** line 1 * line 2 * line 3 ... line k **

Simulating TM has two tapes. Tape 1 holds the linear representation of 2-dimensional TM. Tape 2 holds an index to indicate the position of the head with respect to the line in which head is present. 2-dimensional TM has left, right, up and down movements. Simulation of left and right movements is straight forward. If it involves extending the current line, we can shift the tape contents by one cell to the right. To simulate the 'up' movement, we move the head to the beginning of the previous line (by crossing the separator *) and use the content of tape 2 to shift the head to the corresponding position in the previous line. Simulation of 'down' movement is similar.

6. Multi stack Machines

A multistack TM is an automata with a read-only input tape and k-stacks. Action of the machine depends on TOS symbol in each stack, input symbol, and the current state and involves, (i) change of state (ii) pushing a symbol/pop on each stack. It can be shown that 2-stack machine is equivalent to the Turing machine. It is trivial to show that any 3-tape TM can simulate a 2-stack machine. (Additional 2 tapes can simulate 2 stacks). Now, we will show how a 2-stack machine M_2 can simulate a standard Turing machine M_1.

M_2 first copies the input to the stack 1 and then copies from stack 1 to stack 2. Then, the configuration will be such that stack 1 contains the tape content to the left of the head and stack 2 contains the tape content to the right. TOS symbol of the stack 2 is the current input symbol. If $\delta(q, a) = q', A, R)$, M_2 pops off 'a' from stack 2 and pushes A onto stack 1. If $\delta(q, a) = (q', A, L)$, M_2 pops off a, pushes A onto stack 2 and shifts TOS in stack 1 to stack 2. Set of states Q and accepting states F are same in M_1 and M_2. Hence, if M_1 halts in an accepting state, M_2 also halts in an accepting state.

CONCEPTUAL TOOLS FOR CONSTRUCTION OF TURING MACHINES

Though, the basic model is found to be powerful enough to compute any computable function, some conceptual tools are useful to design TM's to solve complex problems. In this section, we shall discuss many such tools. But, it is to be noted that, these concepts are theoretically equivalent to the basic model. In Section 9.5, we illustrate the use of such tools in the design of TMs to solve complex problems.

1. Memory in Finite Control

Facility of some limited memory in finite control may help Turing machine to 'remember' one or more (up to k) symbols for later action. Conceptually, the state is considered as an ordered pair $(q, [X_1 X_2 \ldots X_k])$ where q is the control part and $[X_1 X_2 \ldots X_k]$ represents the contents of the buffer. As the set of states $Q' = Q \times \Gamma^k$ is finite, the resulting TM is theoretically same as the basic model.

This facility is useful in solving problems involving pattern matching. For example, consider the recognition of $L = \{wcw\}$. To recognize a typical string abbcabb, we have to match each symbol with the corresponding symbol. If the symbol to be matched (a or b) is stored in finite control, it will help in searching the corresponding symbol after the separator c. In the next section, we will describe specific examples to illustrate this concept.

2. Multitrack Tape

In single track, each cell may hold a single symbol. In k-track TM, each cell can hold k symbols, and the head can read all the k symbols. Action depends on all the symbols read. Conceptually, multitrack TM is not different from the basic model. Only the tape alphabet is considered as a k-tuple. If the basic tape alphabet is Γ, alphabet Γ' for k-track TM is Γ^k which is finite. In the next section, we illustrate this concept by designing a 3-track TM to add binary numbers.

3. Subroutines

When the solution involves a repeated solution of a subtask, it is useful to design a TM for the subtask. This TM will be a subroutine of the main TM. To call the subroutine, the main routine gives control to the start state of the subroutine. On return, subroutine gives control to a prespecified state of the main routine. The concept is illustrated by the TM for multiplying two unary numbers described in the next section.

Halting Problem of Turing Machine

According to Church's thesis, a TM can be treated as the most general computing system.

Theorm:

The Halting problem of TM over Σ={0,1} is unsolvable. ie, the problem of determining whether or not an arbitrary TM M over {0,1} halts for an arbitrary input x in Σ^* is unsolvable.

Proof:

Proof is by contradiction. Let M be an arbitrary TM. Let d(M) be the encoded binary string representing M. Then the machine string pair will have d(M)*x as its encoded description. According to our assumption HP is solvable. Hence there exists an algorithm P which decides HP. ie,

= if M halts for input x, then P reaches an accept halt.

= if M does not halt for input x, then P reaches a reject halt.

Let us construct a new algorithm Q based on P as follows:

= it takes d(M) as input and copies it to obtain d(M)*d(M) and then applies algorithm P to this input(ie, d(M)*d(M)),

= Q loops for ever if P reaches an accept halt and Q halts if P reaches a reject halt.

By Church's thesis, there exists a Turing machine say M', which can execute the algorithm Q. Since the algorithm P, as also Q, works for an arbitrary machine M, Q also works for M', so we take M=M'. From (d) and (a) we can conclude that M' loops for ever if M' halts. From (d) and (b) we conclude that M' halts if M' loops for ever. Thus, we obtain the conclusion " M' halts if and only if M' loops for ever". This is a contradiction and HP is unsolvable.