VCSIJ 2018-19-04

MODULE - 1 ATC

K.S. Sampada
Asst. professor
CSE, RNSIT

90+20=110

1. Why study the theory of computation:-

Learning the science of computing address those mathematical properties for solving problems which does not depend on either today's technological details or programming fashions. These theory of computation is still useful in 2 ways:

* It provides a set of abstract structures that are useful for solving certain classes of problems. These abstract structures can be implemented on whatever hardware/software platform is available
* It defines provable limits to what can be computed, regardless of processor speed or memory size. Identification of these limits helps to focus the design effort.

Applications of the theory of computation:

1. Today's applications of computing which enables communi-cation among machines and person are based on theory of computation.
2. Design and implementation of modern programming languages rely heavily on the theory of context-free languages [CFL], CFL are used to document the language's syntax which forms the basis for the parsing tec
3. Programs that translate from one language to another, Natural Language processor, that check the grammar also rely on context-free languages.
4. Diverse systems as communication protocols, Automatic vending machine, security devices are all described as Finite state machines.
5. Interactive video games are finite state machines.
6. Finite state machines & context-free grammars can also be extended to computational biology.

7. The undecidability & tractability that can of automata theory can be extended fromm to design a description language for semantic web.

Languages and strings :-

Alphabet : denoted by Σ is a finile set. The members of Σ symbols or characters.

Eg: $\Sigma = \{a, b\}$, $\Sigma = \{0, 1\}$

Here a, b or 0, 1 are characters / symbols.

Strings : A string is a finile sequence [even empty] of symbols drawn from some alphabet Σ.

Eg: If $\Sigma = \{a, b\}$.

strings drawn from Σ includes ε, a, aaa, ab, ...

shortest string that can be formed from Σ is empty string ε.

Set of all string over an alphabet is represented as Σ^* [Kleene star operator].

Eg: $\Sigma = \{a, b, c, \ldots z\}$ strings : ε, aabbcg, aaaaa ...

$\Sigma = \{0, 1\}$ strings : ε, 0, 00, 11, 010, ...

Functions on strings.

Length of string: $(s) \Rightarrow$ represented as $|s|$ is the number of symbols in s

$$|0110010| = 7$$

$$|\varepsilon| = 0.$$

For any symbol c and string s, the function $\#_c(s)$ is defined to be the number of times that the symbol c occurs in s.

Eg $\#_a(aaaa) = 4$

concatenation:- consider two strings s & t; concatenation of 2 strings s & t represented as $s||t$ or $s.t$ is ~~xrepresented~~ the string formed by appending t to s.

Eg $s =$ good and $t =$ bye then

$s.t =$ "goodbye"

$|st| = |s| + |t| =$

concatenation of strings is associative.

$\forall s, t, w \quad ((st)w = s(tw))$

Empty string ε is the identity for concatenation.

$\forall x \quad (x\varepsilon = \varepsilon.x = x)$

Replication:- For each string w and each natural number i, the string w^i is defined as

$$w^0 = \varepsilon$$

$$w^{i+1} = w^i.w.$$

Eg $a^3 = aaa$

$(bye)^2 = byebye.$

$a^0b^3 = bbb$

Reversal: For each string w, the reverse of w $[w^R]$ is defined as:

If $|w| = 0$ then $w^R = w = \varepsilon$

If $|w| \geq 1$ then $\exists a \in \Sigma, \exists u \in \Sigma^* \ (w = ua)$

$w^R = au^R$.

Theorem 2.1

If w and x are strings then $(wx)^R = x^R w^R$.

Proof: Eg w = name x = tag

$(wx)^R = (\text{nametag})^R$.

$\Rightarrow (x)^R (w)^R = (\text{tag})^R (\text{name})^R$

$= \text{gat eman}$.

proof is by induction on $|x|$:

Base case: $|x| = 0$ Then $x = \varepsilon$.

$(wx)^R = x^R . w^R \Rightarrow (w\varepsilon)^R = w^R$

$= \varepsilon w^R$

$= \varepsilon^R w^R$.

$= x^R . w^R$.

prove: $\forall n \geq 0 (((|x| = n) \rightarrow ((wx)^R = x^R w^R))$

$\rightarrow ((|x| = n+1)$

$\rightarrow ((wx)^R = x^R w^R)))$

consider any string x where $|x| = n+1$

Then $x = ua$ for some character a & $|u| = n$.

so: $(wx)^R = (w(ua))^R$ where $x = ua$

$= ((wu)a)^R$ associativity of concalēnation.

$= a(wu)^R$ def^ of reversal

$= a(u)^R (w)^R$ induction hypothesis

$= (au)^R w^R$. associativity of concalēnation.

$= (ua)^R w^R$ definition of reveral.

$= x^R w^R$. where $ua = x$.

Relations on strings :-

Substring: A string s is a substring of a string t iff s occurs contiguously as part of t.

Eg: aaa is a substring of aaabbbaaa

aaaa is not a substring of aaabbbaaa.

Proper substring: A string s is a proper substring of string t iff s is a substring of t and $s \neq t$. Every string is a substring of itself. The empty string ε is a substring of every string.

Prefix: A string s is a prefix of t iff $\exists x \in \Sigma^*$ ($t = sx$). A string s is a proper prefix of a string t iff s is a prefix of t and $s \neq t$. Every string is a prefix of itself. Empty string ε is a prefix of every string.

Eg string t = abba ; prefixes are.

ε, a, ab, abb, abba.

Suffix: A string s is a suffix of t iff $\exists x \in \Sigma^*$ ($t = xs$)

A string s is a proper suffix of string t iff s is a suffix of t and $s \neq t$. Every string is a suffix of itself. The empty string ε is a suffix of every string.

Eg: t = abba ; suffixes are.

ε, a, ba, bba, abba.

Languages:

A language is a set of strings over finite set of alphabet Σ.

Σ_L => represents the alphabets from which the strings in the language L are formed.

Defining languages given an alphabet:

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

$\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$.

Some of the languages over Σ are:

$\phi, \{\varepsilon\}, \{a, b\}, \{\varepsilon, a, aa, aaa, aaaa, \ldots\}$,

$\{\varepsilon, b, bb, bbb, bbbb, \ldots\}$.

Techniques for defining languages:-

Since languages are sets, they can be defined using any of the set-defining techniques.

$L = \{w \in \{a, b\}^* : \text{all a's precede all b's in } w\}$.

strings $\varepsilon, a, aa, aabbb, ab, bb$ in L.

strings $aba, ba, abc \notin L$

$L = \{x : \exists y \in \{a, b\}^* \ (x = ya)\}$ strings that end in a

strings $a, aa, aaa, bbaa, ba \in L$.

strings $\varepsilon, bab, bca \notin L$.

Empty language

Language that contains no strings.

$L = \{\} = \phi$.

$L = \{\varepsilon\}$ => is a language that has 1 string.

$L = \{\varepsilon\}$ and $L = \phi$ are different.

1) Language containing single 'a'
{a}.

2) 2 a's.
{aa}.

3) atleast 1 a.
{a} {aa} {aaa}.

Halting Problem Language :-

$L = \{w : w$ is a C program that halts on all inputs$\}$

using Prefix relation:

$L_1 = \{w \in \{a,b\}^* :$ no prefix of w contains $b\}$.

$= \{\varepsilon, a, aa, aaa, aaaa, \ldots\}$.

$L_2 = \{w \in \{a,b\}^* :$ no prefix of w starts $b\}$.

$= \{w \in \{a,b\}^* :$ first character of w is $a\} \cup \{\varepsilon\}$.

$L_3 = \{w \in \{a,b\}^* :$ every prefix of w starts with $b\}$

$= \phi$.

L_3 is empty language because ε is the prefix of every string. since ε does not start with b no strings meet L_3's requirement.

using Replication to define a language:

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

$= \{\varepsilon, a, aa, aaa, aaaa, \ldots\}$.

Lexicographic Enumeration:

$L = \{x \in \{a,b\}^* :$ all a's precede all b's$\}$.

Lexicographic enumeration of L is

$= \{\varepsilon, a, b, aa, ab, bb, aaa, aab, abb, bbb,$
$aaaa, aaab, aabb, abbb, bbbb, \ldots\}$.

Cardinality of a Language :-

Smallest language over any alphabet is ϕ, & its cardinality is 0.

Largest language over any alphabet is Σ is Σ^*

Theorem 2.2.

If $\Sigma \neq \phi$ then Σ^* is countably infinite.

Proof :- The elements of Σ^* can be lexicographically enumerated by following procedure:

* Enumerates all strings of length 0, then length 1, length 2 and so on.
* within the strings of a given length, enumerates them in dictionary order.

This enumeration is infinite since there is no longest string in Σ^*. As there exists an infinite enumeration of Σ^*, it is countably infinite.

since any language over Σ is a subset of Σ^*, the cardinality of every language is at least 0 and atmost N_0. Therefore all languages are finite or countably infinite.

Theorem 2.3.

If $\Sigma \neq \phi$ then the set of languages over Σ is uncountably infinite.

Proof :- Set of languages defined on Σ is $P(\Sigma^*)$.

By theorem 2.2, Σ^* is countably infinite.

[By set theory ; if S is countably infinite set, $P(S)$ is uncountably infinite].

Therefore $P(\Sigma^*)$ is uncountably infinite.

P = power set

Functions on Languages:-

Since languages are sets, all of the standard set operations are well-defined on languages.

Set functions applied to Languages:-

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

$L_1 = \{$ strings with even number of a's $\}$.

$= \{\varepsilon, aa, aaaa, aaaaaa, \ldots\}$.

$L_2 = \{$ strings with no b's $\}$.

$= \{\varepsilon, a, aa, aaa, aaaa, \ldots\}$.

union $\cup$

$L_1 \cup L_2 = \{$ strings with even number of a's plus strings with no b's $\}$.

$= \{\varepsilon, a, aa, aaa, aaaa, \ldots\}$.

Intersection $\cap$

$L_1 \cap L_2 = \{$ only strings with even no of a's $\}$.

$= \{\varepsilon, aa, aaaa, aaaaaa, \ldots\}$.

Difference -

$L_2 - L_1 = \{$ strings that are in L_2 and not in $L_1\}$.

$\therefore$ only strings of a's with odd no of a's.

$= \{a, aaa, aaaaa, \ldots\}$.

Complement $\neg$

$\neg(L_2 - L_1) = \{$strings with atleast one b$\}$ $\cup$ $\{$strings with an even number of a's$\}$.

Three operations that are defined on languages are:

* concatenation
* Kleene star
* Reverse.

concatenation of languages:

Let L_1 and L_2 be two languages defined over some alphabet Σ. Then their concatenation is written as $L_1 L_2$

$$L_1 L_2 = \{ w \in \Sigma^* : \exists s \in L_1 (\exists t \in L_2 (w = st)) \}.$$

Eg:- Let $L_1 = \{$ cat, dog, mouse, bird$\}$.

$L_2 = \{$ bone, food$\}$.

$L_1 L_2 = \{$ catbone, cat food, dogbone, dog food, mousebone, mousefood, bird bone, bird food$\}$

* Language $\{\varepsilon\}$ is the identity for concatenation of languages.

$$L\{\varepsilon\} = \{\varepsilon\} L = L$$

* Language ϕ is a zero for concatenation.

$$L \cdot \phi = \phi \cdot L = \phi$$

* concatenation, as a function defined on languages is associative.

$$((L_1 \cdot L_2) L_3) = (L_1 (L_2 L_3))$$

Kleene star:-

Let L be a language defined over some alphabet Σ. Then the kleene star of L is

$$L^* = \{\varepsilon\} \cup \{w \in \Sigma^* : \exists k \geq 1 \ (\exists w_1, w_2, \ldots w_k \in L \ (w = w_1 w_2 \ldots w_k))\}.$$

i.e. L^* is the set of strings that can be formed by concatenating together zero or more strings from L.

Eg:- $L = \{dog, cat, fish\}$ then

$L^* = \{\varepsilon, dog, cat, fish, dogdog, dogcat, \ldots fishdog, fishcatfish, \ldots\}$.

Eg:- Let $L = \{w \in \{a,b\}^* : \#_a(w)$ is odd & $\#_b(w)$ is even$\}$

Then $L^* = \{w \in \{a,b\}^* : \#_b(w)$ is even$\}$.

The constraint on a disappears in the L^* as. Strings in L^* are formed by concatenating together any number of strings from L.

* L^* always contain an infinite number of strings as long as L is not equal to ϕ or $\{\varepsilon\}$.
  - If $L = \phi$ then $L^* = \{\varepsilon\}$.
  - If $L = \{\varepsilon\}$ then $L^* = \{\varepsilon\}$.

$L^+ = L.L^*$ i.e atleast one element of L is selected.

$L^+ = L^* - \{\varepsilon\}$ iff $\varepsilon \notin L$

Let $L = \{0,1\}^+$ be set of binary strings. then the Language $\notin L$ does not include ε.

Reverse :-

Let L be a language defined over some alphabet Σ_1, then the reverse of L is

$$L^R = \{w \in \Sigma_1^* : w = x^R \text{ for some } x \in L\}.$$

i.e L^R is set of strings that can be formed by taking some string in L and reversing it.

Reverse of a language is defined in terms of definition of reverse applied to strings.

Theorem 2.4

If L_1 and L_2 are languages then $(L_1 L_2)^R = L_2^R L_1^R$

proof :- If x and y are strings then

$\forall x (\forall y ((xy)^R = y^R x^R))$ - By theorem 2.1

$(L_1 L_2)^R = \{(xy)^R : x \in L_1 \& y \in L_2\}$ - Def^n of concatenation.

$= \{y^R x^R : x \in L_1 \& y \in L_2\}$. - Def^n of reverse

$= L_2^R L_1^R$

Excercise :-

2.1 consider the language $L = \{1^n 2^n : n > 0\}$.

Is the string 122 in L ?

string in L must have equal number of 1's & 2's.

So 122 is not in L.

2.2. Let $L_1 = \{a^n b^n : n > 0\}$ & $L_2 = \{c^n : n > 0\}$. for each of the following string, stale whether or not it is an element of $L_1 L_2$.

(a) ε NO, Both in L_1 & L_2 $n > 0$

(b) aabbcc. YES

(c) abbcc NO L_1 = Equal no of a's & b's

(d) aabbcccc YES

2.3. Let $L_1 = \{$Peach, apple, cherry$\}$ and $L_2 = \{$pie, cobbler, $\varepsilon\}$ List the elements of $L_1 L_2$ in lexicographic order.

$L_1 L_2 = \{$apple, peach, cherry, apple pie, peach pie, cherry pie, apple cobbler, peach cobbler, cherry cobbler$\}$.

2.4. Let $L = \{w \in \{a, b\}^* : |w| \equiv_3 0\}$. List the 1st six elements in a lexicographic enumeration of L.

$L = \{\varepsilon, aaa, aab, aba, abb, baa \ldots\}$.

2.5.

2.6. For each of the following Languages L, give an English description. Give 2 strings that are in L and that are not in L.

(a) $L = \{w \in \{a,b\}^* ;$ exactly one prefix of w ends in a $\}$.

L is a set of strings composed of zero or more b's and 'a', strings ending with prefix 'a'.

a, bba, bbab $\in L$. and bbb, aaa $\notin L$.

prefix ending with a.

A Language Hierarchy:-

To categorise the problems that are easy to solve and those that are not, a unifying framework is to be defined to which any computational problem can be cast.

That frame work is Language recognition. Let L be the definition of language and w be the string. Then "Is w in L?" question is answered with either yes/NO; which is defined as decision problems.

Problems can be divided into 2 major categories.

* Problems that are already stated as decision problems.

  To solve these kind of problems encode the input as strings, define a language that contains exactly the set of inputs for which the answer is yes for "is w in L?".

* Problems that are not stated as decision problems.

  First reformulate the problem as a decision problem and then encode it as a language recognition task.

constructing an appropriate string encoding:

1. Given a search string w and a web document d do they match?.
   (or) should a search engine on input w, consider returning d?

   $L = \{ \langle w, d \rangle : d$ is a candidate match for the query $w \}$.

2. To decide whether sum of 2 integer belongs to the integer.

$L = \{ w \mid w = \langle i_1 \rangle + \langle i_2 \rangle = \langle i_3 \rangle ; i \in \Sigma \{0 \ldots 9\}^* \}$.

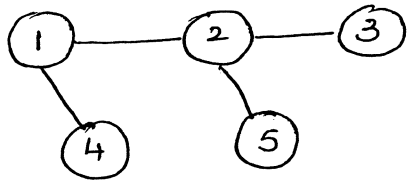
Eg: $w = 2 + 4 = 6$; $23 + 47 = 70 \in L$.

$w = 2 + 4 = 10 \notin L$.

3. Graph connectivity:

Given an undirected graph, is it connected?

Let G be the graph represented as follows: is G connected.



Let V be the vertices represented as binary number and E be the edge represented as pair of vertices joining 2 vertices.

Eg: $V = \{(1), (10), (11), (100), (101)\}$.

$E(1,2) = \{ \langle 1 | 10 \rangle \}$

for the above graph. there are 5 vertices & the edges are represented as.

$101 | \langle 1|10 \rangle | \langle 10|11 \rangle | \langle 1|100 \rangle | \langle 10|101 \rangle$

Language

$L = \{ w \in \{0,1,|\}^* ; w = n_1 | n_2 | \ldots n_i$ where $n \in \{0,1\}^*$ and w encodes directed graph$\}$.

4. Given a protein fragment f and a protein molecule p, could f be a fragment of p.?.

$L = \{ \langle f, p \rangle$ f could be fragment from p$\}$.

Represent each protien molecule as a sequence of amino acids residue. Each of the amino acid is represented as a character, Now compare whether f is a substring of P.

casting Problems as Decision question.

Problems that are not already stated as decision questions, can be transformed into decision questions. i.e. they can be reformulated so that they become Language recognition problems. The main idea is to encode into a single string, both the inputs and outputs of the original problem. P.

Eg:- Addition as decision.
Given two non-negative integers, compute their sum.

Encoding:- Transform the problem of adding 2 numbers into the problem of checking whether the third number is the sum of 1st two.

Integersum = $\{$w of the form: $\langle i_1 \rangle + \langle i_2 \rangle = \langle i_3 \rangle$, where $\langle i_1 \rangle, \langle i_2 \rangle, \langle i_3 \rangle \in \{0,1,2,3,4,5,6,7,8,9\}^+$ & $\langle i_3 \rangle$ is $\langle i_1 \rangle + \langle i_2 \rangle\}$.

Eg:- sorting as decision.
Given list of integers, sort it.

Encoding:- Transform the problem of sorting a list into problem of examining a pair of list & deciding whether the second corresponds to the sorted version of the 1st.

$L = \{ w_1 \# w_2 : \exists n \geq 1 (w_1$ is of the form $i_1, i_2, \ldots i_n$, w_2 is of the form $i_1, i_2, \ldots i_n$ and w_2 contains same objects as w_1 and w_2 is sorted) $\}$.

Eg : $w_1 = 1, 5, 3, 9, 6$.
$w_2 = 1, 3, 5, 6, 9$
∴ the string is in L.

$w_1 = 1, 5, 3, 9, 6$
$w_2 = 1, 2, 3, 4, 5, 6, 7$
String is not in L.

Eg: Database querying as Decision.
Given a database and a query, execute the query against the database.

Encoding: Transform the task of executing the query into the problem of evaluating a reply to see if it is correct.

$L = \{ d \# q \# a :$ d is an encoding of a database,
q is a string representing a query
a is the correct result of applying q to d $\}$.

d = (name, age, phone), (J, 23, 567), (M, 24, 234)
q = (select name age = 23)
a = J.

Machine Based Hierarchy of Language :-

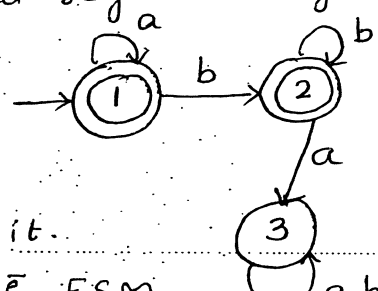
The computational models allows to write the programs in such a way that it accepts some language L. Hierarchy of computational model is defined as follows:

1. 1st model is very simple, programs written for it is easy to understand, they run in linear time and algorithms exists to answer almost any question asked about such programs.
   - finite state machine.
2. 2nd model is more powerful, but still limited.
   - Push down Automata
3. 3rd model is powerful enough to describe anything that can be computed by any sort of real computer.
   - Turing Machine.

Regular Languages

The class of languages that can be accepted by finite state machine is called regular language.

Finite state machine is as shown in figure. FSM has a start state, with an unlabelled arrow leading to it.



FSM starts in its start state. FSM changes state based on the transitions as it reads each character. Machine accepts string s if it reaches accepting state as marked with double circle after reading last character from s. otherwise it rejects it.

Context-Free Languages:

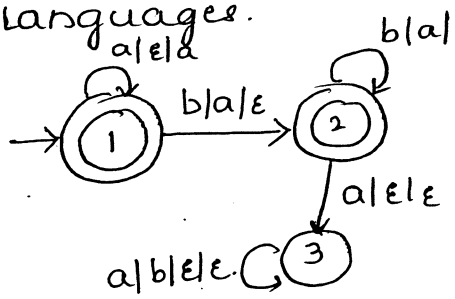
Language accepted by Push down Automata [PDA] is context-free languages.

[FSM] finite state machine has the drawback of counting the no of characters it reads as it has the memory in the states.

eg $L = \{a^n b^n \mid n \geq 0\}$ in this the language accepted by L is all a's come first, # number of a's = number of b's.

Trying to build FSM for the above language is not possible as ~~it cannot~~ there is no bound on no of a's.

Machine that consists of FSM and a single stack called PDA are designed to implement the languages.



simple PDA that accepts $a^n b^n$ is as shown in figure.

As it reads the character it pushes it on to the stack.

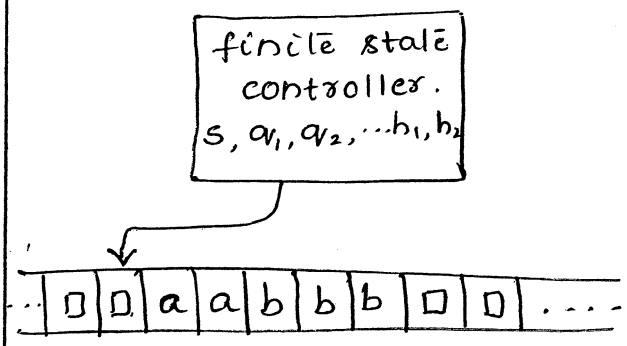
Transitions on PDA is based on next character to be read, top of the stack & the current state.

Most of the programming languages, query language and markup languages are context free.

Decidable and semidecidable languages.

Language of English sentences in which word occurs more than once are not context-free. Thus building FSM or PDA for those languages is not possible.

The resulting machine which accepts the language is Turing Machine.



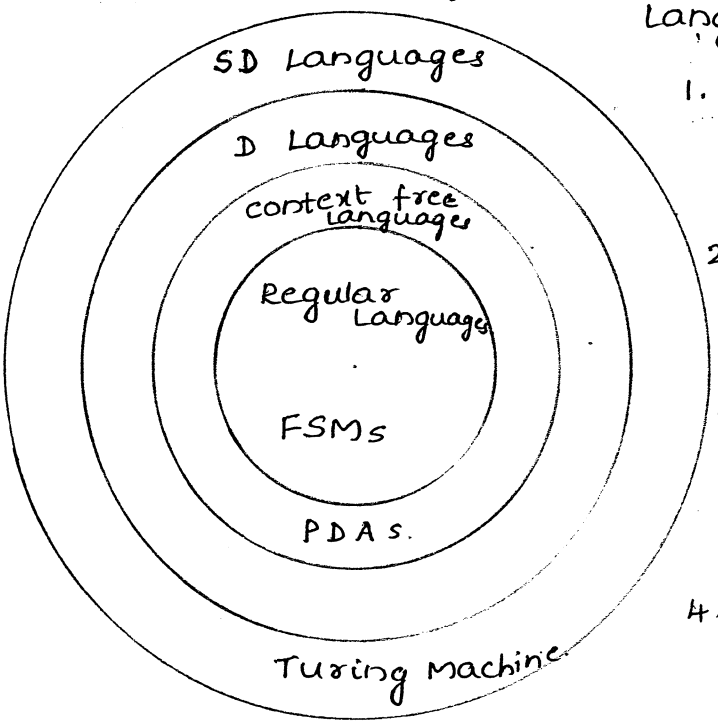
The structure of Turing Machine [TM] is as shown in figure. write the input string onto the tape and then start the machine with read/write head pointing at beginning location of the i/p string. At each step TM M considers its current state and the character that is on the tape pointing by read/write head. Read/write head can move in one direction or it can move back and forth many times/forever.

Turing Machine can be used to define two new classes of languages.

* Language L is **decidable** iff there exists a Turing Machine M that halts on all inputs, accepts all strings that are in L and rejects the strings that are not in L.
  M always says YES or NO as appropriate.

* Language L is **semidecidable** iff there exists a Turing Machine M that accepts all strings that are in L and fails to accept every string that is not in L. (given a string that is not in L, M may reject or it may loop forever).
  M can say YES but may fail to say NO.

Computational Hierarchy:- Four classes of Languages defined are



1. Regular language - accepted by FSM
2. Contex-free Language. - accepted by PDAs
3. Decidable Lang (D- Language) - accepted by TM
4. Semi Decidable lang (SD - Lang) - accepted by TM.

Each of these classes are proper subset of next class as shown in figure.

Language Hierarchy have access to tools with greater expressive power as they move outwards. The expressiveness can be considered with the following payoff's:

* computational effeciency:- computational effeciency increases with language hierarchy. The run-time of various automata are: FSM - Linear time with length of i/p string. PDA's has cube of the length of i/p string whereas TM may require time that grows exponentially.

* Decidability:-

* clarity:- Given a particular problem, it is described in its simplest form using tools by the designer.

Eg:- Regular expression / Regular language - FSM.

context free grammar / context-free language - PDA.

No corresponding tools exists for broader classes of decidable & semidecidable languages.

* Rule of least power : use the least powerful language suitable for expressing information, constraints or programs on WWW.

Tractability Hierarchy of language classes:

Decidable languages can further be classified based on the resources [time, space] required by M.

* P, which contains those languages that can be decided in time that grows as some polynomial function of the length of the input.
* NP, contains those languages that can be decided by a non-deterministic machine with the property that the amount of time required to explore one sequence of guesses grows as some polynomial function of the length of the input.
* PSPACE; which contain those languages that can be decided by a machine whose space requirement grows as some polynomial function of the length of the input.

$$P \subseteq NP \subseteq PSPACE.$$

COMPUTATION:-

Key ideas for computations are:

* Decision procedures
* Nondeterminism
* Functions on Languages.

Decision procedures:-

A decision procedure is an algorithm to solve decision problem. Decision procedure must be guaranteed to halt on all inputs. It is a program whose result is a Boolean value. Decision procedures must posses two correctness properties:

* program must be guaranteed to halt on all inputs
* when the program halts and returns an answer, it must be the correct answer for given input.

Example-1: check for even numbers.

```
even (x: integer) =
        If (x/2) * 2 = x then return True.
                                Else FALSE.
```

This procedure is decidable as it answers YES/NO for given input x.

Example-2: check for prime numbers:
(TO prove loop terminates).

```
prime (x: positive integer)
    For (i = 2 to ceiling (sqrt(x))) do.
        if (x/i) * i = x then return FALSE
    Return True.
```

where ceiling(sqrt(x)) = $\lceil \sqrt{x} \rceil$

procedure checks whether the given number x is prime or not. It is decidable.

Fermat number is a sequence defined as.

$$F_n = 2^{2^n} + 1, \quad n \geq 0$$

$F_0 = 3$, $F_1 = 5$, $F_2 = 17$, $F_3 = 257$, ...

Example-3 : check for small prime Fermat numbers.

```
fermatSmall() =
    i = 0
    Repeat:
        candidate = (2 ** (2 ** i)) + 1
        If candidate is prime then return True.
        i = i + 1
    until candidate >= 1,000,000
    Return FALSE
```

procedure is guaranteed to halt as the value of the candidate increases each time i is increased and the loop terminates when candidate value exceeds 1,000,000.

Example-4 : Program that halt on a particular input.

Given a program P that takes a string w as an i/p parameter, does P halt on some value of w?

```
halts_on_w (P: program, w: string) =
    1. Simulate the execution of P on w
    2. If simulation halts return True
                    else return False.
```

halts_on_w is not a decision procedure, because it never return the value False.

This kind of procedure is called semidecision procedure.

Determinism and Nondeterminism: consider function.

* choose (action 1;
  action 2;
  :
  action n)

The function choose is presented with a finite set of alternatives, each of which will return either a success value or the value False. choose( ) will.

* Return some success value, if there is one
* Else choose ( ) will
  - ✓ halt and return False
  - ✓ fail to halt if any of the actions fails to halt.

* choose (x from S : P(x))

The choose ( ) is presented with set of S values, where S may be finite or infinite. choose ( ) will.

* Return some element x of S such that P(x) halts with a value other than False, if there is one.
* Else choose ( ) will.
  - ✓ Halt and return False if it can be determined that all elements of x of S, P(x) is not satisfied.
  - ✓ Fail to halt if there is no mechanism for determining that no elements of S that satisfy P exists.

In both of the cases of choose ( ), the function is non-deterministic.

Example-1:- choosing a travel plan.

Finite state Machines: [FSM]

Finite state machine is a computational device whose input is a string and whose output is one of two values that is Accept or Reject. FSM's are also called finite state automata o[FSA].

Deterministic FSM [DFSM]

DfSM M is a quintuple $(K, \Sigma, \delta, s, A)$, where

* K is a finite set of states
* Σ is the input alphabet.
* $s \in K$ is the start state.
* $A \subseteq K$ is the set of accepting state
* δ is the transition function

$$K \times \Sigma \rightarrow K.$$

(state) (i/p alphabet) → (state)

A configuration of a DFSM M is an element of $K * \Sigma^*$. It captures 2 things important to M's future behaviour:

* Its current state
* i/p that is still left to read.

The initial configuration of DFSM M, on i/p w is (s_M, w) where s_M is start state of M.

The transition function δ defines the operation of DFSM M one step at a time. It can be used to define the sequence of configuration that M will enter.

Let 'c' be any element of Σ_1 and w be string of Σ_1^*, Then

$(q_1, cw) \vdash_M (q_2, w)$ iff $((q_1, c), q_2) \in \delta$

$c_1 \vdash_M^* c_2$ → configuration c_1 yields configuration c_2 iff M can go from c_1 to c_2 in zero or more steps.

computation by M is a finite sequence of configu-rations. $c_0, c_1, c_2, \ldots c_n$ for some $n \geq 0$ such that:

* c_0 is an initial configuration
* c_n is of the form (q, ε) for some state $q \in A_M$. and
* $c_0 \vdash_M c_1 \vdash_M c_2 \vdash_M \ldots \vdash_M c_n$.

Let w be an element of Σ_1^*. Then

* M accepts w iff $(s, w) \vdash_M^* (q, \varepsilon)$ for some $q \in A_M$ where $q \in A_M$ is called Accepting state.
* M rejects w iff $(s, w) \vdash_M^* (q, \varepsilon)$ for some $q \notin A_M$ where $q \notin A_M$ is rejecting configuration

M halts whenever it enters Accepting or a rejecting configuration.

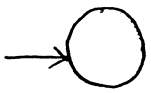
Language Accepted by M denoted by L(M) is the set of all strings accepted by M.

A simple Language of a's & b's.

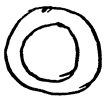
Let $L = \{w \in (a,b)^* \}$: where a is immediately followed by b }

Design :-

1. Start state will be indicated with the unlabelled arrow pointing to it



2. Accepting state will be indicated with double circles.



3. Transitions are defined as :

q_0 is initial state.

$\delta = \{((q_0, a), q_1)$ → accepting a.

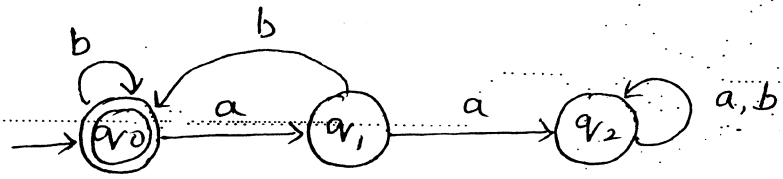
$((q_0, b), q_0)$, → any no of b's are accepted.

$((q_1, a), q_2)$ → a should be followed by b (By def^ of L)

$((q_1, b), q_0)$ → q_0 is accepting configuration

$((q_2, a), q_2)$ } → q_2 is rejecting state.

$((q_2, b), q_2)$ } as 'aa' is not accepted.



Eg:- $(q_0, abbab) \vdash (q_1, bbab)$

$\vdash (q_0, bab)$

$\vdash (q_0, ab)$

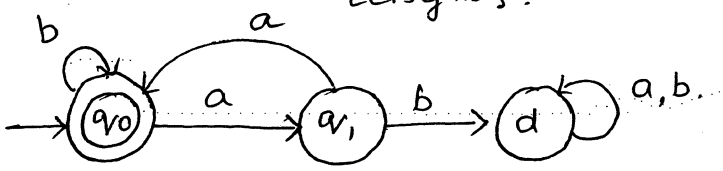
$\vdash (q_1, b)$

$\vdash (q_0, e)$.

Since $q_0 \in A$ (is accepting state) hence M accepts.

Even Length Regions of a's.

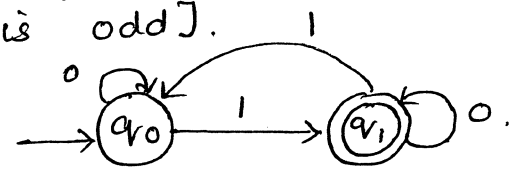
$L = \{w \in \{a,b\}^* :$ every 'a' region in w is of even Length$\}$.



checking for odd parity.

$L = \{w \in \{0,1\}^* :$ w has a odd parity$\}$

[string has odd parity iff the no of 1's in it is odd].

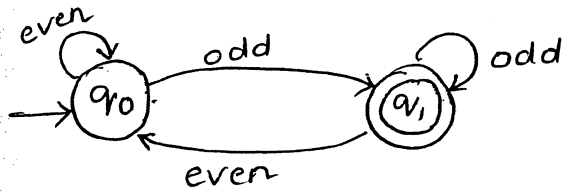


Theorem: 5.1 DFSM Halt.

Every DFSM M on input w, halts after |w| steps.

proof: on input w, M executes some computation $c_0 \vdash_M c_1 \vdash_M c_2 \vdash_M \ldots \vdash_M c_n$ where c_0 is an initial configuration and c_n is of the form (q, ε) for some $q \in K_M$. c_n is either accepting or rejecting configuration. so M will halt when it reaches c_n. Each step in the computation consumes one character of w. so $n = |w|$. Thus M will halt after |w| steps.

construct an FSM to accept odd Integers.



on input 235, the configurations are:

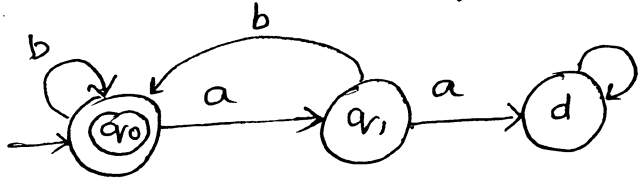
$(q_0, 235) \vdash_M (q_0, 35)$

$\vdash_M (q_1, 5)$

$(q_0, 235) \vdash_M (q_1, \varepsilon)$

A Language is regular iff it is accepted by some FSM.

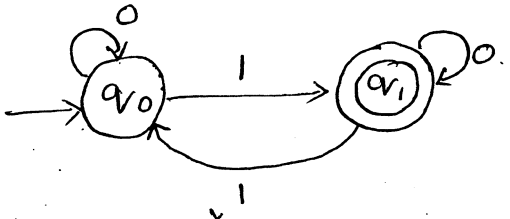
Eg: $L = \{w \in \{a,b\}^*$, every 'a' is followed by b$\}$.



where 'd' is dead state.

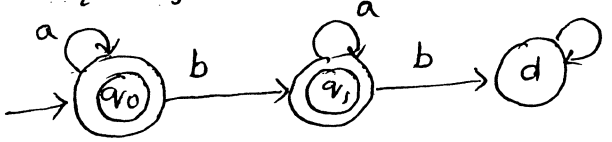
Parity checking:

$L = \{w \in \{0,1\}^*$: w has odd parity$\}$.



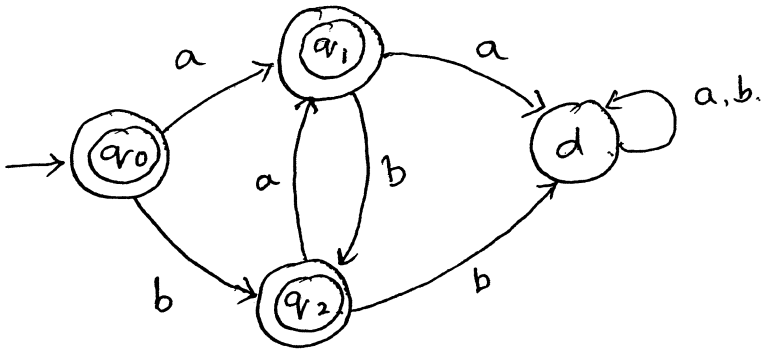
$L = \{w \in \{a,b\}^*$; every 'a' region in w is of even length$\}$

$L = \{w \in \{a,b\}^*$: no more than 1 b is accepted$\}$



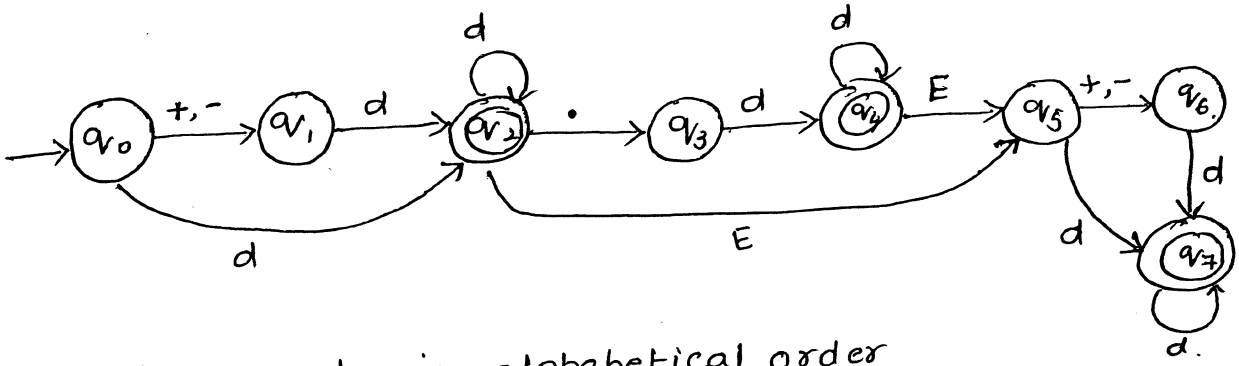
consecutive character.

$L = \{w \in \{a,b\}^*$: no two consecutive characters are same$\}$



Language of Floating point numbers is Regular.

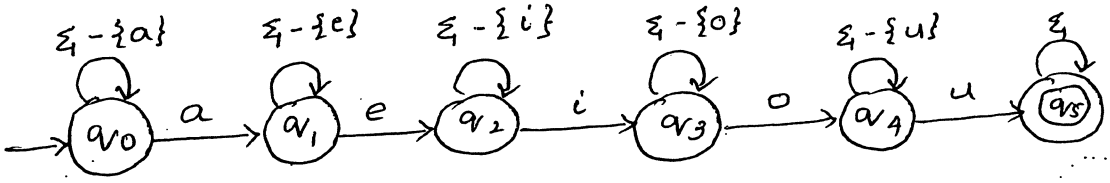
Eg: 3, +3.0, 3.0, 0.3 E1, 0.3 E+1, −0.3 E −1, -3E8



All the vowels in alphabetical order.

$L = \{w \in \{a-z\}^*$; all vowels a, e, i, o, u in w occur in alphabetical order$\}$.

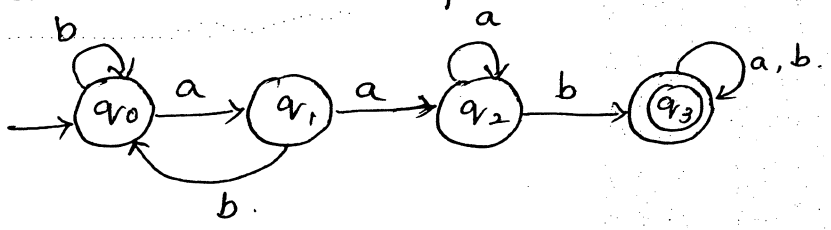
L = {abstemious, facetious, sacrilegious}.



Substrings that doesn't occur:

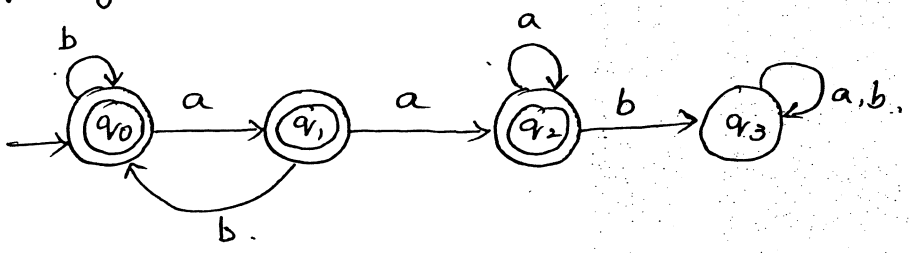
$L = \{ w \in \{a, b\}^* : w \text{ does not contain } aab \}$.

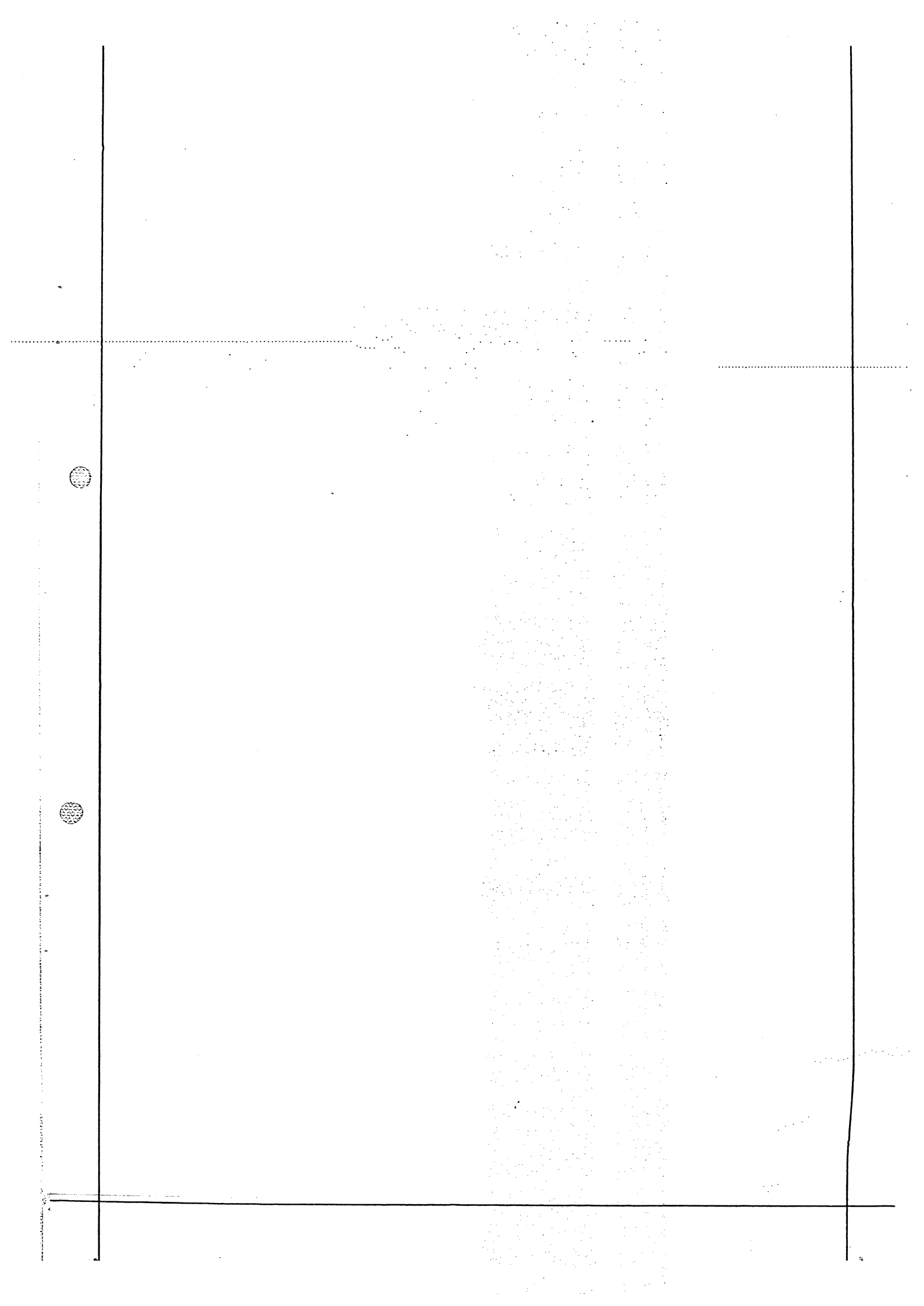
Build DFSM to accept aab.



Not to accept 'aab' = $\neg L$

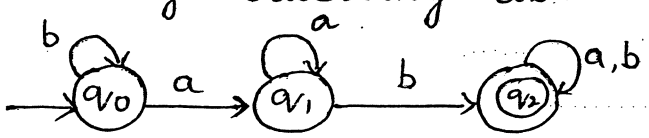
convert the above FSM by making q_0, q_1, q_2 as accepting states and q_3 as non-accepting state



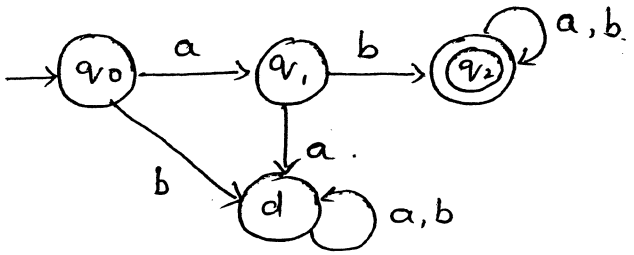


1. Design DFSM for the following; given $\Sigma = \{a, b\}$.

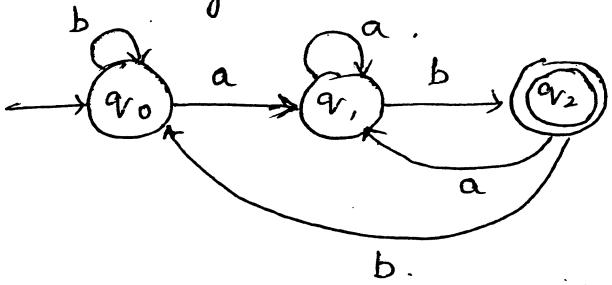
(i) containing substring 'ab'.



(ii) Beginning with 'ab'.

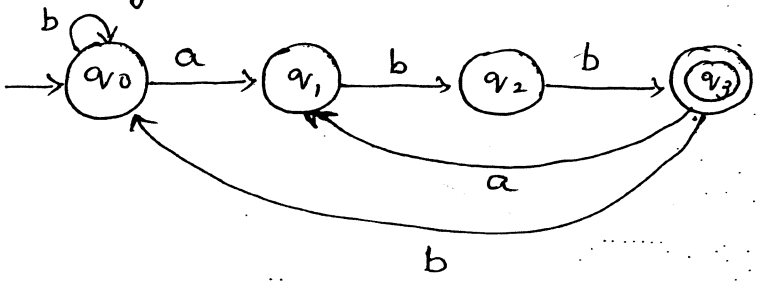


(iii) Ending with 'ab'



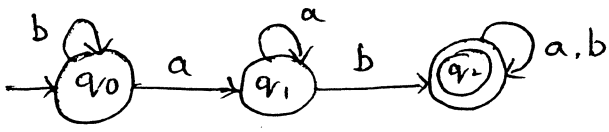
2. Design DFSM to accept the Languages over $\Sigma = \{a, b\}$

(i) Ending with abb.

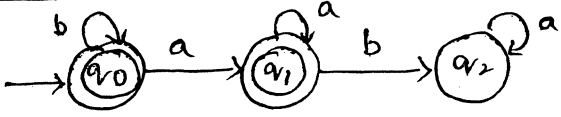


(ii) Not containing substring 'ab'

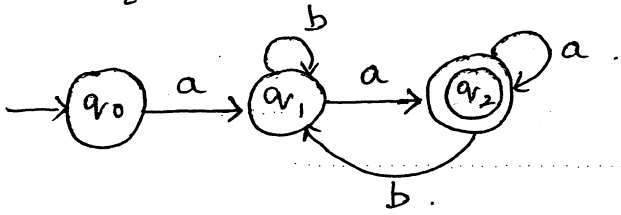
Step-1 construct DFSM containing substring 'ab'



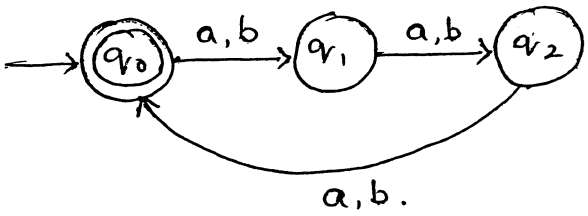
Step-2 :- complement the above DFSM.



(iii) $L = \{ awa \mid w \in (a,b)^* \}$.

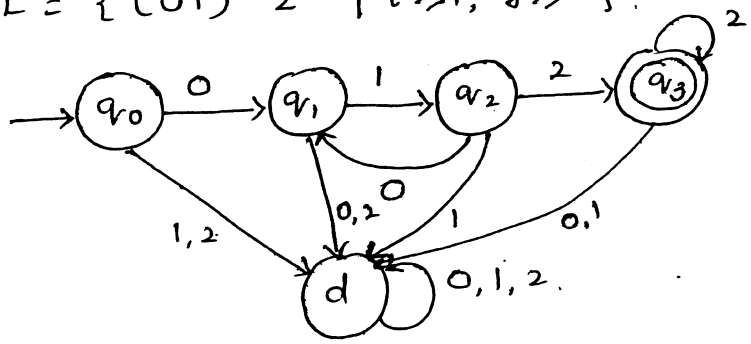


(iv) $L = \{ w \mid |w| \bmod 3 = 0 \}$.



3. Design DFSM to accept the following language over $\Sigma = \{0, 1, 2\}$

(i) $L = \{ (01)^i 2^j \mid i \geq 1, j \geq 1 \}$.



(ii)

4. Design DFSM to check whether the give decimal number is divisible by 3.

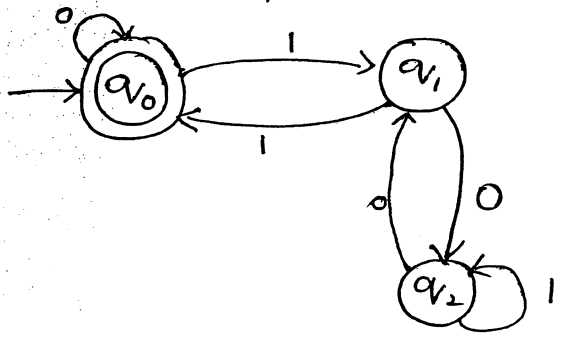
q_0 → where reminder is zero [divisible by 3]

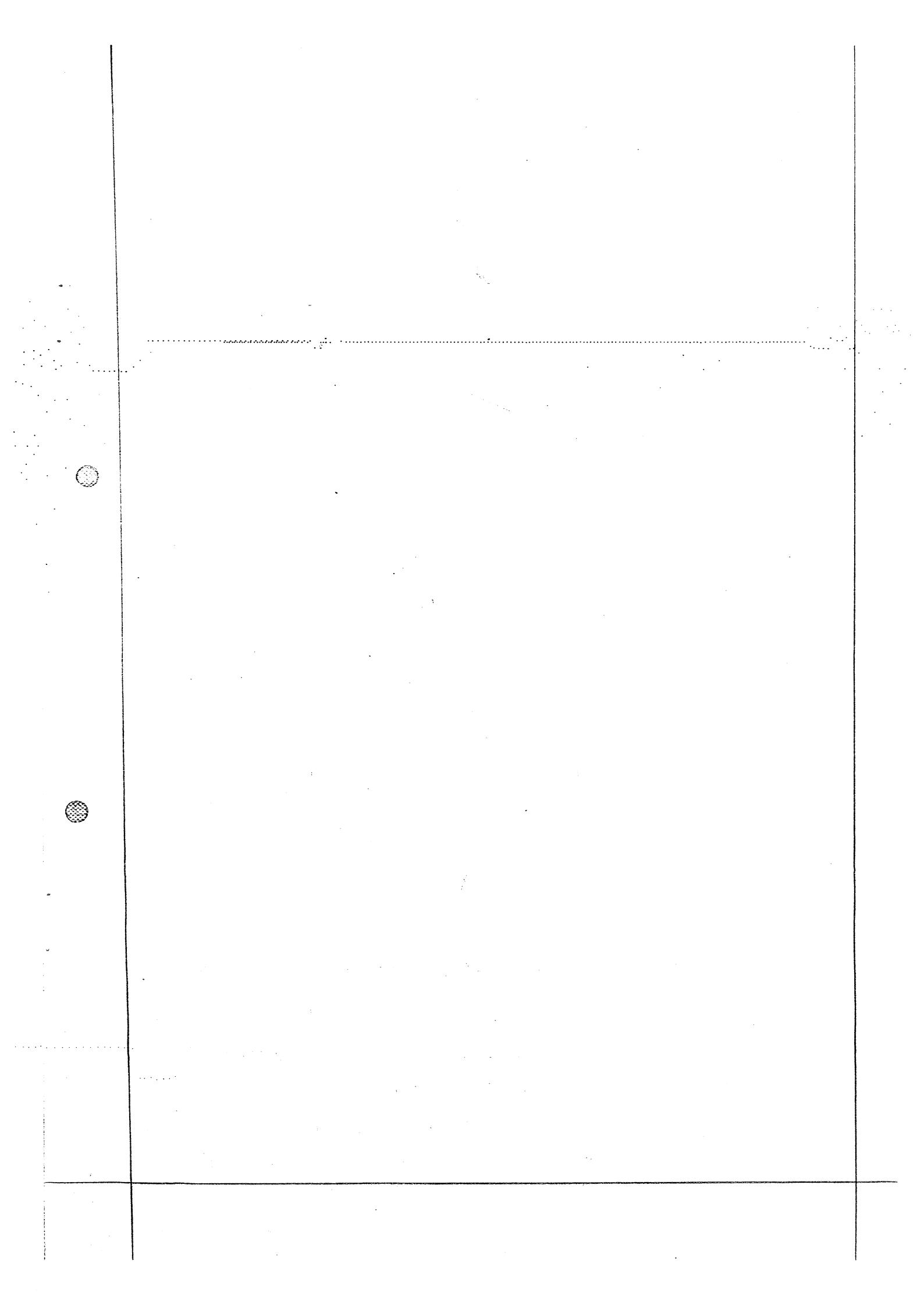
q_1 → Reminder is 1 [1, 4, 7, ...]

q_2 → Reminder is 2 [2, 5, 8, ...].

The numbers where intepreted as binary

000 — q_0
001 — q_1
010 — q_2
011 — q_0
100 — q_1
101 — q_2
110 — q_0
111 — q_1





Nondeterministic FSMs.

A nondeterministic FSM (NDFSM) M is a quintuple $(K, \Sigma, \Delta, S, A)$ where;

* K is a finite set of states.
* Σ is a set of alphabet
* $S \in K$ is the start state
* $A \subseteq$ is the set of final states and
* Δ is the transition relation.

$$(K \times (\Sigma \cup \{\varepsilon\})) \rightarrow K.$$

i.e Δ contains (state, i/p symbol or ε) pair and transition to new state.

Let w be an element of Σ^*, then

* M accepts w iff at least one of its computation accepts.
* M rejects w iff none of its computations accepts

The Language accepted by M is denoted by L(M), is set of all strings accepted by M.

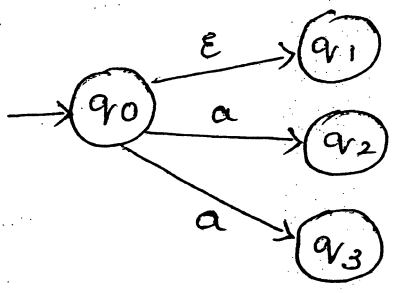
Note: Δ (transition) is may be function and also a relation.

* NDFSM M may enter a configuration in which there are still i/p symbols left to read but from which two no moves are available, hence M cannot reach accepting configuration. M will halt rejecting the string. This is possible because M is not a function.

i.e $\Delta \quad K \times (\Sigma) \rightarrow \{\}$ is not defined.

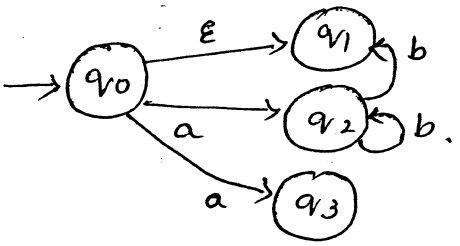
* NDFSM M may enter configuration from which two or more competing moves are possible. This property defines transition is a relation.

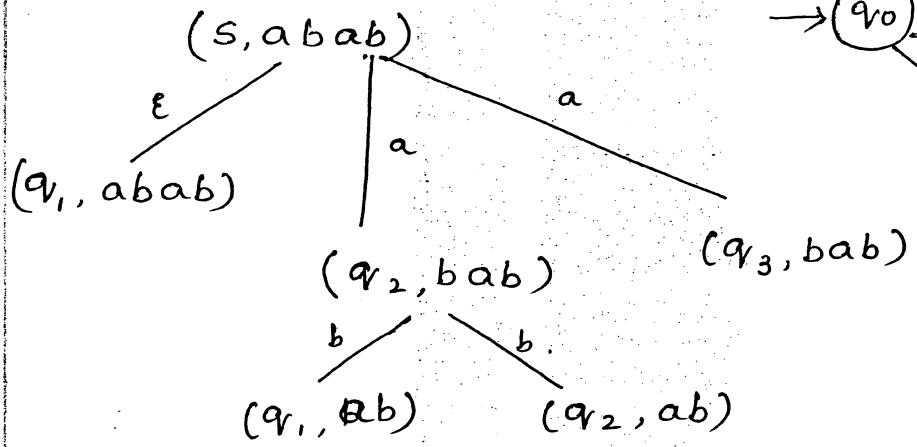
Eg:



* NDFSM M may have ~~one~~ zero, one or more transition with a character from Σ.
* ε-transition may or may not be followed. It is useful as it enables M to guess the correct path.

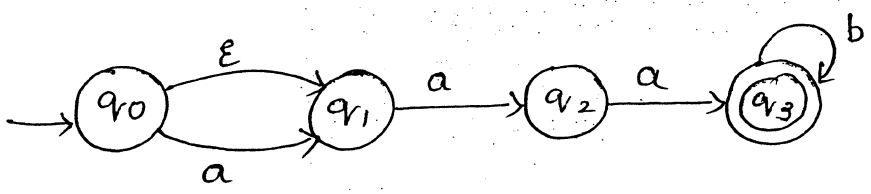
Example: consider the ~~above~~ figure of an NDFSM M. Search for string abab.



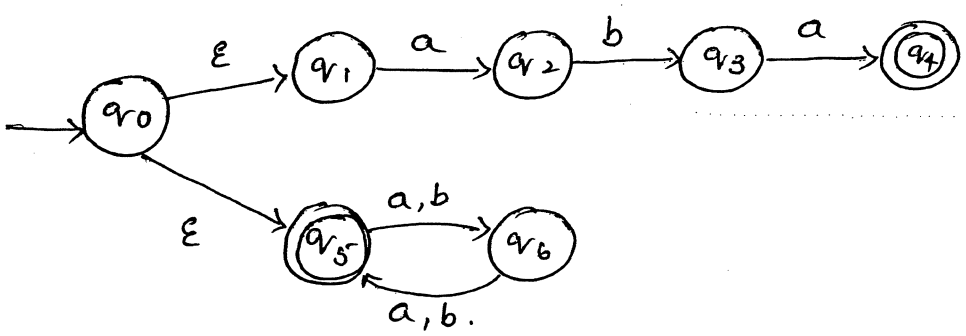


Design NDFSM for the Language.

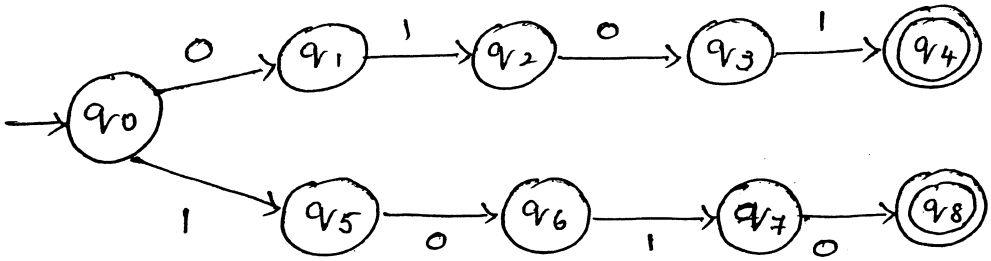
$L = \{w \in \{a,b\}^* : w$ consists of optional a followed by aa followed by zero or more b's$\}$



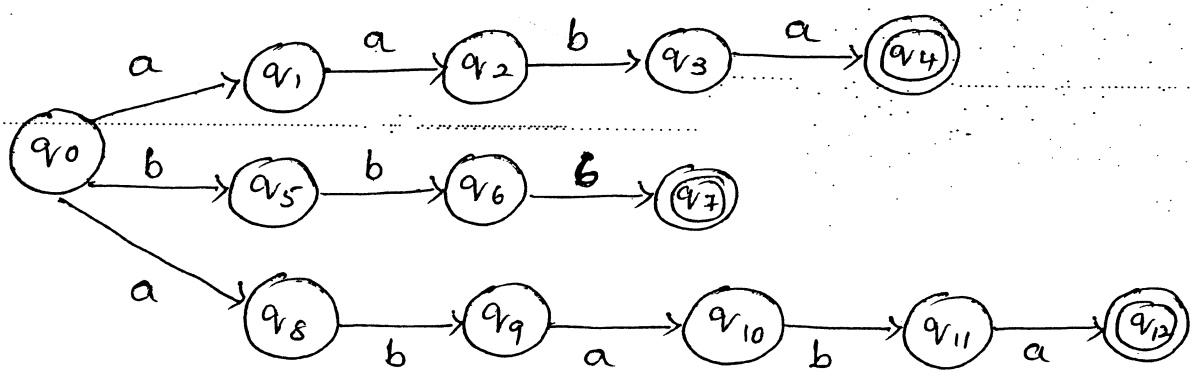
Design NDFSM for the Languages
$L = \{w \in \{a,b\}^* : w = aba \text{ or } |w| \text{ is even}\}$.

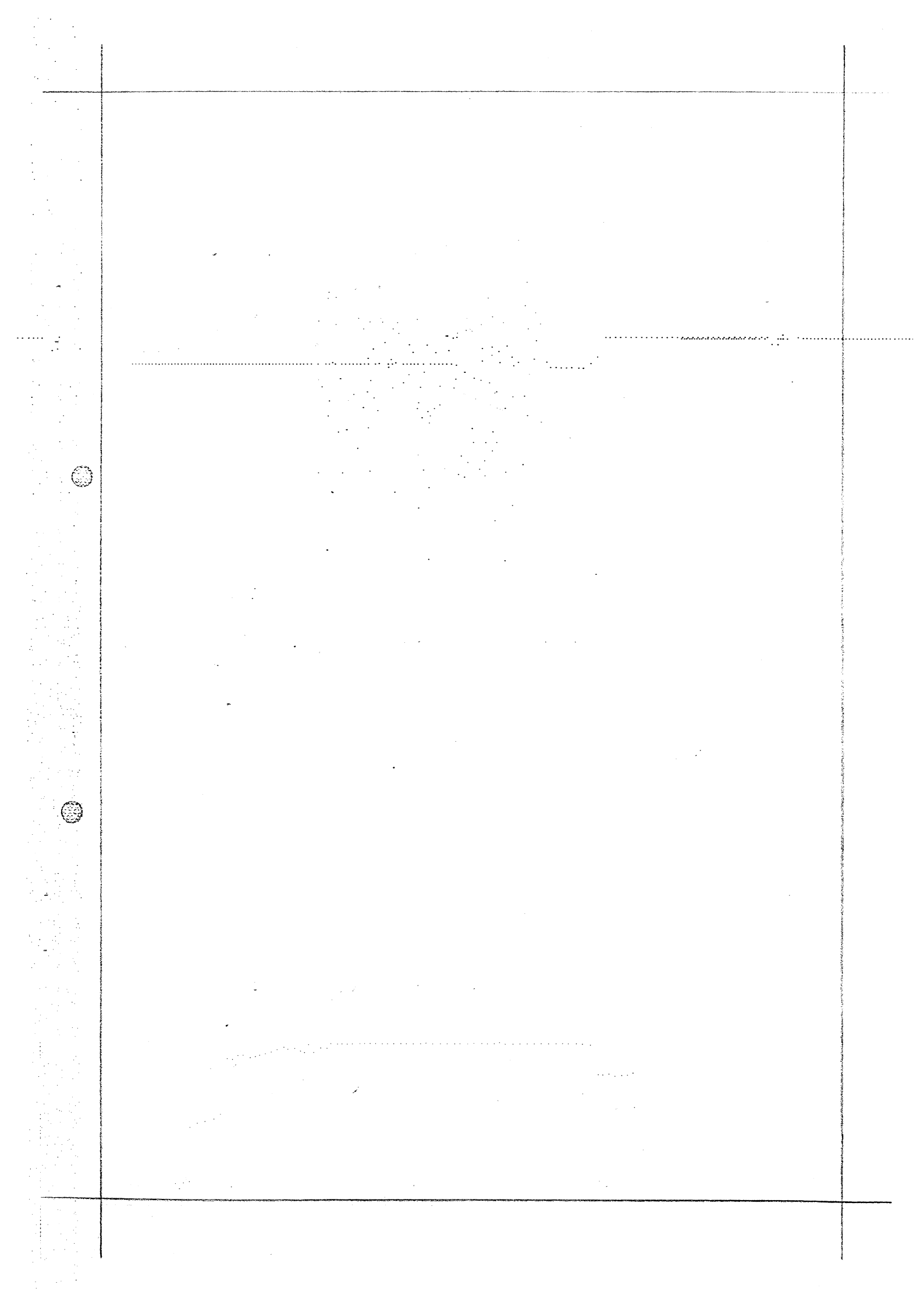


Design NDFSM for the language which contains both 101 and 010 as substrings.



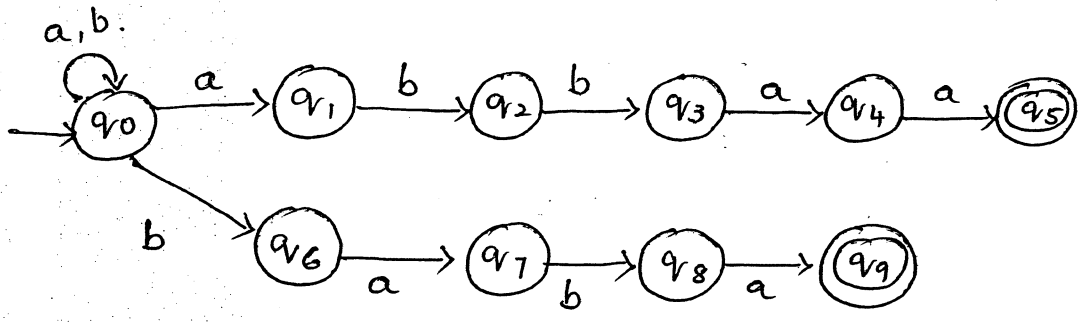
Design NDFSM for the Language containing substring aaba, bbb, ababa.





Analysing Non-deterministic FSMs.

consider the NDFSM for abbaa or baba.



Find whether the strings "ababa" and "abb" are accepted

for string "ababa"

$(q_0, \underline{a}baba) \rightarrow \{q_0, q_1\}$.

$(\{q_0, q_1\}, \underline{b}aba) \rightarrow \{q_0, q_6, q_2\}$.

$(\{q_0, q_6, q_2\}, aba) \rightarrow \{q_0, q_1, q_7\}$ (q_2, a) is not defined

$(\{q_0, q_1, q_7\}, \underline{b}a) \rightarrow \{q_0, q_6, q_2, q_8\}$.

$(\{q_0, q_6, q_2, q_8\}, a) \rightarrow \{q_0, q_1, q_7, q_9\}$. (q_2, a) is not defined.

In this configuration q_9 is accepting state, hence the string is accepted.

for string "abb".

$(q_0, \underline{a}bb) \rightarrow \{q_0, q_1\}$.

$(\{q_0, q_1\}, bb) \rightarrow \{q_0, q_6, q_2\}$.

$(\{q_0, q_6, q_2\}, b) \rightarrow \{q_0, q_6, q_3\}$. (q_6, b) is not defined

$\{q_0, q_6, q_3\}$ none of the configuration is accepting.

Hence string "abb" is rejected.

ε - transitions :-

$eps : K_M \rightarrow \mathcal{P}(K_M)$.

eps (q) is defined as for some state q in M, is the set of states of M that are reachable from q by following zero or more ε-transitions.

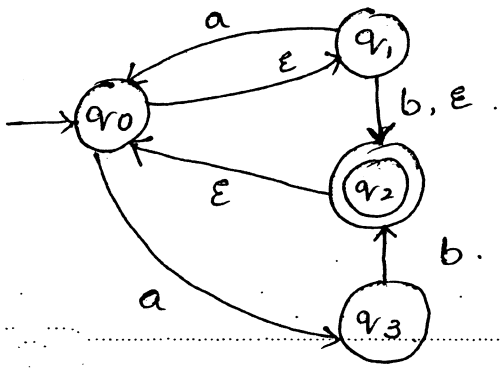
$$eps(q) = \{p \in K : (q, w) \vdash_M^* (p, w)\}.$$

To compute eps (q): [epsilon closure].

eps (q : state)

1. result = {q} (state itself)
2. while there exists some $p \in$ result and some $r \notin$ result and transition $(p, \varepsilon, r) \in \Delta$ do:
   insert r into result
3. return result.

compute eps of each state for the following NDFSM.



$eps(q_0) = \{q_0, q_1, q_2\}$.

$eps(q_1) = \{q_1, q_2, q_0\}$.

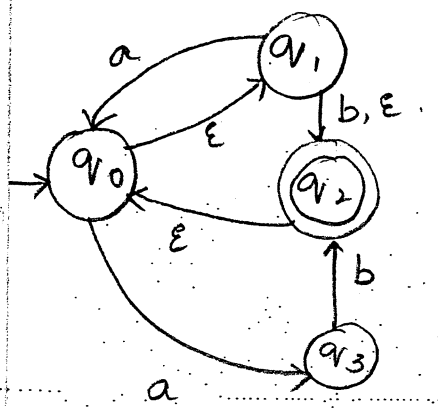
$eps(q_2) = \{q_2, q_0, q_1\}$.

$eps(q_3) = \{q_3\}$.

Simulation Algorithm

Ndfsmsimulate (M: NDFSM, w: string) =

1. current-state = eps(s)
2. while any input symbol in w remain to be read do:
   - 2.1 c = get-next-symbol(w)
   - 2.2 next-state = ϕ
   - 2.3. for each state q in current-state do:
     - for each state p such that $(q, c, p) \in \Delta$ do:
       - next-state = next-state $\cup$ eps(p).
   - 2.4 current-state = next-state.
3. If current-state contains any states in A accept else reject.



Analysing for string abb

current-state = eps(s)

= $\{q_0, q_1, q_2\}$

$(\{q_0, q_1, q_2\}, a\,bb) \rightarrow \{q_3\}$.

$(\{q_3\}, bb) \rightarrow (q_2)$ eps(q_2)

= $\{q_0, q_1, q_2\}$

$(\{q_0, q_1, q_2\}, b) \rightarrow \{q_2\}$ eps(q_2)

= $\{q_0\ q_1\ q_2\}$

Since q_2 is accepting state

abb is accepted.

Equivalence of Nondeterministic and deterministic FSMs.

Theorem 5.2.

FOR EVERY DFSM THERE IS AN EQUIVALENT NDFSM.

Proof:- Let M be a DFSM that accepts some language L. M is also an NDFSM that happens to contain no ε-transitions and whose transition relation happens to be a function. So the NDFSM exists is simply M.

Theorem 5.3.

GIVEN AN NDFSM $M = (K, \Sigma, \Delta, S, A)$ THAT ACCEPTS SOME LANGUAGE L THERE EXISTS AN EQUIVALENT DFSM THAT ACCEPTS L.

Proof:- Proof is by construction of an equivalent DFSM M'. The states of M' will corresponds to set of states in M. So $M' = (K', \Sigma, \delta', s', A')$ where

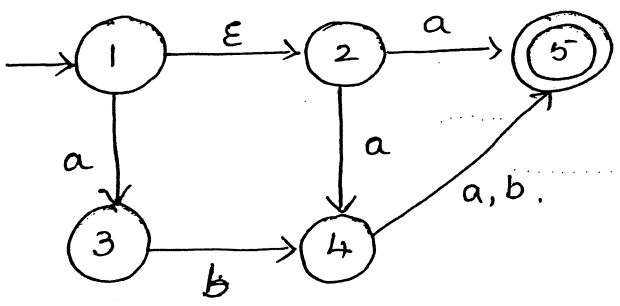
* K' contain one state for each element of $\mathcal{P}(K)$
* $s' = eps(s)$
* $A' = \{Q \subseteq K : Q \cap A \neq \phi\}$
* $\delta'(Q, c) = \bigcup \{eps(p) : \exists p \in Q ((q, c, p) \in \Delta)\}$

Algorithm to compute M' given M [NDFSM to DFSM]

ndfsmtodfsm (M: NDFSM) =

1. For each state q in K do:
   compute eps (q)
2. $s' = eps(s)$
3. compute δ':
   (a) active-states = $\{s'\}$.
   (b) $\delta' = \phi$
   (c) while there exists some element Q of active-states for which δ' has not yet computed do:
      for each character c in Σ do:
         new-state = ϕ.
         For each state q in Q do:
            For each state p such that $(q, c, p) \in \Delta$ do:
               new-state = new-state $\cup$ eps (p)
         Add the transition (Q, c, new-state) to δ'
      If new-state $\notin$ active-states then insert it into active-states.
4. $K' =$ active-states.
5. $A' = \{Q \in K' : Q \cap A \neq \phi\}$.

convert the following NDFSM to DFSM:



Sol^:

1. find eps of each state

eps (1) = {1, 2} eps (3) = {3} eps (5) = {5}.

eps (2) = {2} eps (4) = {4}

2. $s' = eps(s)$

$\therefore s' = eps(q_0') = \{1, 2\}$.

3. Active - states = {1, 2}.

(i) ({1, 2}, a) = {3, 4, 5}.

({1, 2}, b) = {ϕ}.

(ii) ({3, 4, 5}, a) = {5}.

({3, 4, 5}, b) = {4, 5}.

(iii) ({5}, a) = ϕ

({5}, b) = ϕ

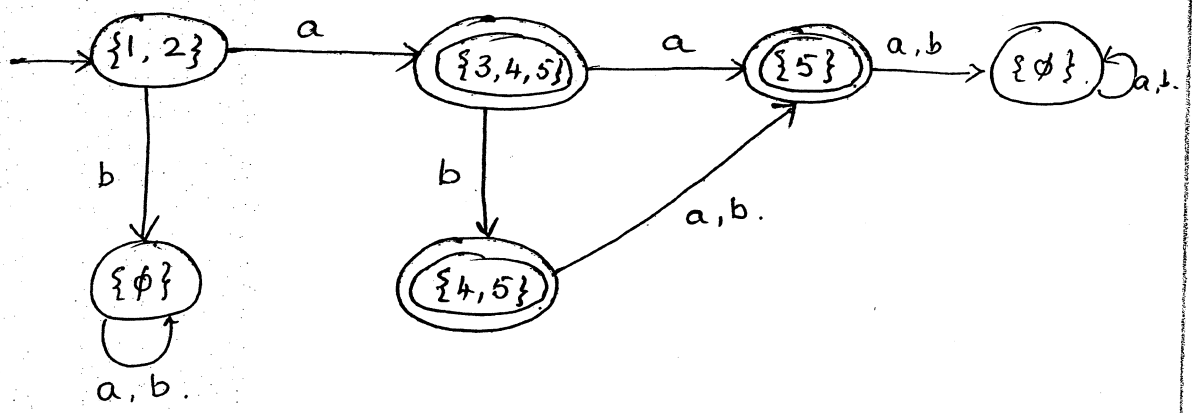
(iv) ({4, 5}, a) = {5}.

({4, 5}, b) = {5}.

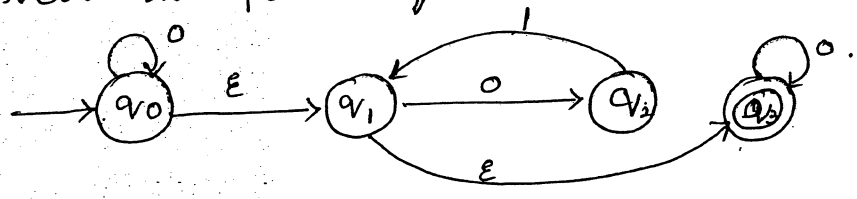
$\therefore$ active - states = {1, 2}, {3, 4, 5}, {4, 5}, {5}, ϕ

4. $K' = \{\{1, 2\}, \{3, 4, 5\}, \{4, 5\}, \{5\}\}$.

5. $A' = \{\{3, 4, 5\}, \{4, 5\}, \{5\}$.



convert the following NDFSM to DFSM.



Soln:-

1. find eps() of each state.

$eps(q_0) = \{q_0, q_1, q_3\}$.

$eps(q_1) = \{q_1, q_3\}$.

$eps(q_2) = \{q_2\}$.

$eps(q_3) = \{q_3\}$.

2. $S' = eps(S)$

$S' = eps(q_0) = \{q_0, q_1, q_3\}$.

3. Active states $\{q_0\ q_1\ q_3\}$.

(i) $(\{q_0\ q_1\ q_3\}, 0) = eps(\delta(q_0,0) \cup \delta(q_1,0) \cup \delta(q_3,0))$

$= \{q_0\ q_1\ q_3\ q_2\}$.

$(\{q_0, q_1, q_3\}, 1) = eps(\delta(q_0,1) \cup \delta(q_1,1) \cup \delta(q_3,1))$

$= eps(\phi \cup \phi \cup \phi)$

$= \phi$.

(ii) $(\{q_0, q_1, q_2, q_3\}, 0) = eps(\delta\{q_0,0\} \cup \delta(q_\phi,0) \cup \delta(q_2,\phi) \cup \delta(q_3,0))$

$= \{q_0\ q_1\ q_3\ q_2\}$.

$(\{q_0 q_1 q_2 q_3\}, 1) = eps(\delta(q_0,1) \cup \delta(q_1,1) \cup \delta(q_2,1) \cup \delta(q_3,1))$

$= eps(\phi \cup \phi \cup \{q_1 q_3\} \cup \phi)$

$= \{q_1, q_3\}$.

(iii) $(\{q_1, q_3\}, 0) = eps(\delta(q_1,0) \cup \delta(q_3,0))$

$= eps(q_2 \cup q_3)$

$= \{q_2 q_3\}$.

$(\{q_1, q_3\}, 1) = eps(\delta(q_3,1) \cup \delta(q_3,1))$

$= eps(\phi \cup \phi)$

$= \phi$.

(IV) $(\{q_2 q_3\}, 0) = eps(\delta\{q_2,0\} \cup \delta(q_3,0))$

$= eps(\phi \cup q_3) = \{q_3\}$.

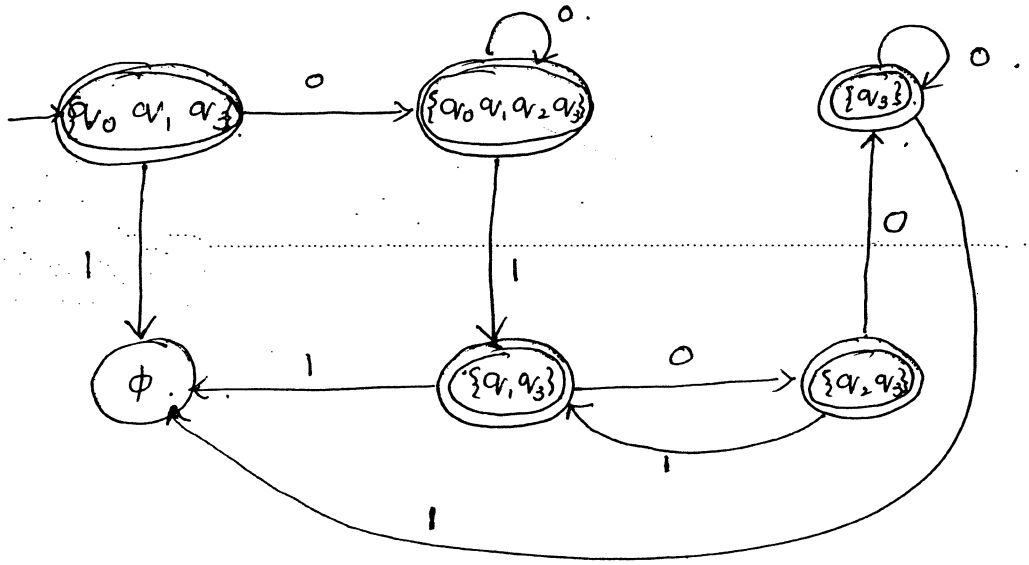
$(\{q_2, q_3\}, 1) = eps(\delta(q_2,1) \cup \delta(q_3,1))$.

$= eps(\{q_1\} \cup \phi)$

$= \{q_1, q_3\}$.

(v) $\{q_3, 0) = \{q_3\}$.

$\{q_3, 1) = \phi$.

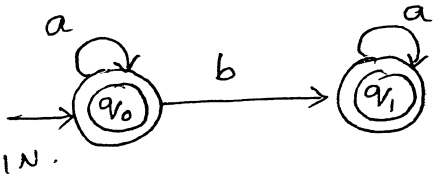


Simulators for FSM:

Once FSM are designed to solve the problem, it can be simulated to execute.

Simulating DFSM: Assume FSM as the specification for simple program, then write the code.

Eg: $L = \{w \in \{a,b\}^* : w$ contains no more than one $b\}$.




```
BEGIN.
S:   q0 = get_next_symbol.
     If q0 = eof then accept.
     Else if q0 = a then go to S
     else if q0 = b then go to T

T:   q1 = get_next_symbol.
     If q1 = eof then accept
     Else if q1 = 'a' then go to T
     else if q1 = 'b' then reject.

END.
```

Simple Interpreter for DFSM $M = (K, \Sigma, \delta, s, A)$:

```
dfsmsimulate (M: DFSM, w: string) =
1. st = s
2. Repeat
     2.1. c = get_next_symbol(w).
     2.2. If c ≠ eof then:
          2.2.1 st = δ(st, c).
   until c = eof.
3. If st ∈ A then accept else reject.
```

Simulating Non-deterministic FSMs.

ndfsmsimulate (M: NDFSM, w: string) =

1. Declare set st /* st will hold current state.
2. Declare set st1 /* st1 will be built to contain next state.
3. st = eps (s)
4. Repeat:
   c = get_next_symbol(w)
   If c ≠ eof then do:
      st1 = ϕ.
      for all q ∈ st do:
         for all r: (q c r) ∈ Δ do:
            st1 = st1 ∪ eps(r)
      st = st1.
      If S = ϕ then exit.
   until c = eof.
5. If st ∩ A ≠ ϕ then accept else reject.

Minimizing FSMs:

DFSM M is minimal iff there is no other DFSM M' such that $L(M) = L(M')$ and M' has fewer states than M does.

1. BUILDING MINIMAL DFSM FOR A LANGUAGE.

x and y are <u>indistinguishable</u> with respect to L is denoted as $x \not\equiv_L y$ iff:

$\forall z \in \Sigma^*$ (either both xz and $yz \in L$ or neither is)

x and y are <u>distinguishable</u> w.r.t L iff they are not indistinguishable. If x and y are distinguishable then there exists at least one string z such that $xz \in L$ and $yz \notin L$ or $xz \notin L$ and $yz \in L$.

Example:

1. If $L = \{a\}^*$, then $a \not\equiv_L aa \not\equiv_L aaa$

That is any number of 'a' generated belongs to L.

2. If $L = \{w \in \{a,b\}^* : |w| \text{ is even}\}$. then.

$a \not\equiv_L aaa$ [yes, they are indistinguishable because both of them does not belong to L]

$a \not\equiv_L aa$ [No, they are distinguishable because $a \notin L$ whereas $aa \in L$].

$\not\equiv_L$ is an equivalence relation because it is

* REFLEXIVE: $\forall x \in \Sigma^* (x \not\equiv_L x)$, $\because \forall x, z \in \Sigma^* (xz \in L \leftrightarrow xz \in L)$

* SYMMETRIC: $\forall x, y \in \Sigma^* (x \not\equiv_L y \rightarrow y \not\equiv_L x)$, because

$\forall x, y, z \in \Sigma^* ((xz \in L \leftrightarrow yz \in L) \leftrightarrow (yz \in L \leftrightarrow xz \in L))$

* TRANSITIVE: $\forall x, y, z \in \Sigma^* (((x \not\equiv_L y) \wedge (y \not\equiv_L w)) \rightarrow (x \not\equiv_L w))$

because:

$\forall x, y, z \in \Sigma^* (((xz \in L \leftrightarrow yz \in L) \wedge (yz \in L \leftrightarrow wz \in L)) \rightarrow$

$(xz \in L \leftrightarrow wz \in L))$.

* NO equivalence class of $\approx_L$ is empty
* Every string in Σ^* is in exactly one equivalence class of $\approx_L$.

Determining $\approx_L$

1. $L = \{w \in \Sigma^* :$ every 'a' is immediately followed by b$\}$

To determine equivalence class of $\approx_L$

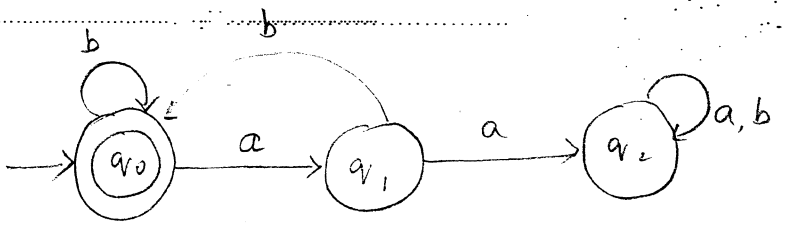
(a) $\{\varepsilon, b, abb, \ldots\}$ /* all strings in L

(b) $\{a, abbba, \ldots\}$. /* strings that end in a & is not followed by b.

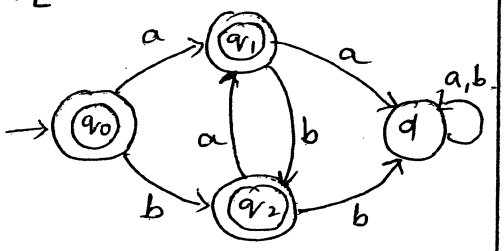
(c) $\{aa, abaa, \ldots\}$. /* that contain atleast one instance of aa.

Three key points about $\approx_L$:

* No equivalence class can contain both strings that are in L and strings that are not.
* If there are strings that would take DFSM for L to the dead state, then there will be one equivalence class of $\approx_L$ that corresponds to dead state.
* Some equivalence class contains ε. It will correspond to the start state of the minimal machine that accepts L.



$L = \{w \in \{a,b\}^* : \text{no two adjacent characters are same}\}$.
Determine equivalence classes of $\approx_L$.



The equivalence classes are:

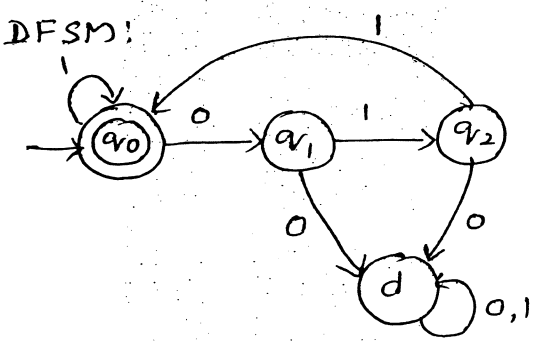
* $\{\varepsilon\}$. $\in L$ (q_0)
* $\{a, aba, ababa, \ldots\}$ $\in L$ (q_1)
* $\{b, bab, abab, \ldots\}$ $\in L$ (q_2)
* $\{aa, abaa, ababb, \ldots\}$ $\notin L$ (d).

There may be more than one equivalence class that contains strings that are in L.

Describe equivalence classes of $\approx_L$ for each of the following:

(a) $L = \{w \in \{0,1\}^* : \text{every 0 in w immediately followed by 11}\}$.

DFSM:

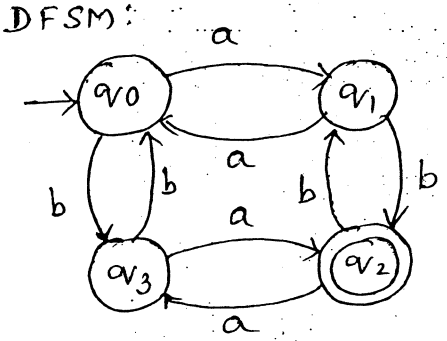


Equivalence class $\approx_L$.

* {Strings in L}.
* {in L but ends in 0}.
* {in L but ends in 01}.
* {ends in dead state}.

(b) $L = \{w \in \{0,1\}^* : w$ has odd number of a's and an odd number of b's$\}$.

DFSM:



Equivalence class $\approx_L$.

* {strings in L}.
* {in L but one more b}.
* {in L but 1 more a}.
* {even number of a's & b's}.

Determine $\eqslantless_L$ for $A^n B^n$
constructing equivalence classes of $\eqslantless_L$.

* $\{\varepsilon\}$
* $\{a\}$
* $\{aa\}$
* $\{aaa\}$.

Each new string of a's has to go in an equivalence class distinct from shorter string because each string requires different continuation string in order to become in L.

So if $L = A^n B^n$, then $\eqslantless_L$ has an infinite number of equivalence classes, therefore $A^n B^n$ is not regular.

Theorem 5.4 $\eqslantless_L$ imposes a lower bound on the minimum number of states of a DFSM for L.

LET L BE A REGULAR LANGUAGE AND LET $M = (K, \Sigma, \delta, s, A)$ BE A DFSM THAT ACCEPTS L. THE NUMBER OF STATES IN M IS GREATER THAN OR EQUAL TO THE NUMBER OF EQUIVALENCE CLASSES OF $\eqslantless_L$.

Proof: suppose that the number of states in M were less than the number of equivalence classes of $\eqslantless_L$. Then by pigeonhole principle, there must be atleast one state q that contains strings from atleast two equivalence classes of $\eqslantless_L$. Then M's future behaviour on those strings will be identical, which is not consistent with the fact that they are in different equivalence classes of $\eqslantless_L$.

Pigeonhole principle states that if 'p' items are put into 'h' pigeonholes with $p > h$ then atleast one pigeonhole must contain more than 1 item.

2. THERE EXISTS A UNIQUE MINIMAL DFSM FOR EVERY REGULAR LANGUAGE.

Theorem:

LET L BE A REGULAR LANGUAGE OVER SOME ALPHABET Σ. THEN THERE IS A DFSM M THAT ACCEPTS L AND THAT HAS PRECISELY 'n' STATES WHERE 'n' is THE NUMBER OF EQUIVALE-NCE CLASSES OF $\approx_L$. ANY OTHER DFSM THAT ACCEPTS L MUST EITHER HAVE MORE STATES THAN M OR IT MUST BE EQUIVALENT TO M EXCEPT FOR STATE NAMES.

Proof: proof is by construction of $M = (K, \Sigma, \delta, s, A)$ where:

* K contains 'n' states, one for each equivalence classes of $\approx_L$.
* $s = [\varepsilon]$, equivalence class of ε under $\approx_L$.
* $A = \{[x] : x \in L\}$.
* $\delta([x], a) = [xa]$ [i.e. M/c is in state x and after reading character a, it moves to state $[xa]$.]

For this construction to prove the theorem, we must show:

* K is finite. Since L is regular, it is accepted by some DFSM M'. M' has some finite number of states m. $n \leq m$ $\therefore$ K is finite.
* δ is a function defined for all (state, input) pairs. and it produces a unique value for each of them. The fact that construction guarantees a unique value follows from definition of $\approx_L$.
* $L = L(M)$ i.e M accepts language L. To prove this show that $\forall s, t((([\varepsilon], st) \vdash_M^* ([s], t))$.

If $|s| = K$ the claim is true. what happens if $|s| = K+1$, let $s = yc$ where $y = \Sigma_1^*$ and $c = \Sigma_1$.

/* M reads first character:

$([\varepsilon], yct) \vdash_M^* ([y], ct)$ Induction hypothesis. $|y| = K$.

/* M reads one more character

$([y], ct) \vdash_M^* ([yc], t)$ definition of δ_M.

/* combining these two after M has read K+1 chia

$([\varepsilon], yct) \vdash_M^* ([yc], t)$

$([\varepsilon], st) \vdash_M^* ([s], t)$ Because $s = yc$.

let $t = \varepsilon$., s be any string Σ_1^*

$([\varepsilon], s.\varepsilon) \vdash_M^* ([s].\varepsilon)$

M will accept iff $[s] \in A$, which by the way in which A was constructed, it will be if the strings in [s] are in L. M accepts precisely strings in M.

* By theorem 5.4 [number of equivalence classes of $\equiv_L$ imposes lower bound on number of states in any DFSM that accepts L], there exists no smaller machine M# that also accepts L.
* There is no different machine M# that also has 'n' states and that accepts L.

Myhill-Nerode Theorem..

Theorem:

A LANGUAGE IS REGULAR IFF THE NUMBER OF EQUIVALENCE CLASSES OF $\equiv_L$ IS FINITE.

Proof:

L regular → the number of equivalence classes of $\equiv_L$ is finite:

If L is regular, then there exists some DFSM M that accepts L. M has some finite number of states m. By theorem 5.4, the number of equivalence classes of $\equiv_L \leq m$. Hence the number of equivalence classes of $\equiv_L$ is finite.

The number of equivalence classes $\equiv_L$ is finite → L regular:

If the number of equivalence classes of $\equiv_L$ is finite, then the construction that was described in proof of theorem 5.5 will build a DFSM that accepts L. Hence L must be regular.

BUILDING A MINIMAL DFSM FROM $\equiv_L$.

Let $\Sigma = \{a, b\}$ and $L = \{w \in \{a,b\}^* : \text{no 2 adjacent characters are same}\}$.

Equivalence classes of $\equiv_L$ are:

$\{\varepsilon\}$ $[q_0]$

$\{a, aba, ababa, \ldots\}$ $[q_1]$

$\{b, ab, bab, abab, \ldots\}$ $[q_2]$

$\{aa, abaa, ababb, \ldots\}$ $[d]$.

Build minimal DFSM M to accept L as follows:

* Equivalence classes of $\equiv_L$ become states of M.
* Start state is $[\varepsilon] = [q_0]$
* Accepting state are all equivalence classes that contain string in L $[q_0]$ $[q_1]$ $[q_2]$.
* $\delta([x], a) = [xa]$

4. MINIMISING AN EXISTING DFSM:

There are two approaches to construct minimal DFSM.

1. Begin with M and collapse redundant states, getting rid of one at a time until resulting machine is minimal.
2. Begin by overclustering the states of L into just two groups, [accepting and non-accepting]. Then iteratively split those groups apart until all the distinctions that L requires have been made.

2nd approach is considered for constructing minimal machine.

TWO states q and p in M are equivalent. [is written as $q \equiv p$] iff for all strings $w \in \Sigma^*$, either w drives M to an accepting state from both p & q or it drives to rejecting state.

The definition can be written as, For all $p, q \in K$.

* $p \equiv^0 q$ iff they are both accepting or both are rejecting states.
* For all $n \geq 1$, $q \equiv^n p$ iff

$$q \equiv^{n-1} p \text{ and}$$

$$\forall a \in \Sigma (\delta(p,a) \equiv^{n-1} \delta(q,a))$$

min DFSM (M: DFSM) =

1. classes = {A, K-A}.

2. Repeat until a pass at which no change to classes has been made:

   2.1 newclasses = ϕ

   2.2 For each equivalence class e in classes, if e contains more than one state, see if it needs to be split:

   For each state q in e do:

   For each character c in Σ do:

   Determine which element of classes q goes to if 'c' is read.

   If there are any two states p and q such that there is any character c such that when 'c' is read, p goes to one element of classes and q goes to another, then p & q must be split. Insert those classes into new classes. If behaviour of 2 states do not differ, no splitting is necessary.

   2.3 classes = new classes.

3. Return $M' = (\text{classes}, \Sigma, \delta, [S_M], \{[q]: \text{the elements of } q \text{ are in } A_M]\})$, where δ_M is $\delta_M([q], c) = p$ then $\delta_M([q], c) = [p]$.

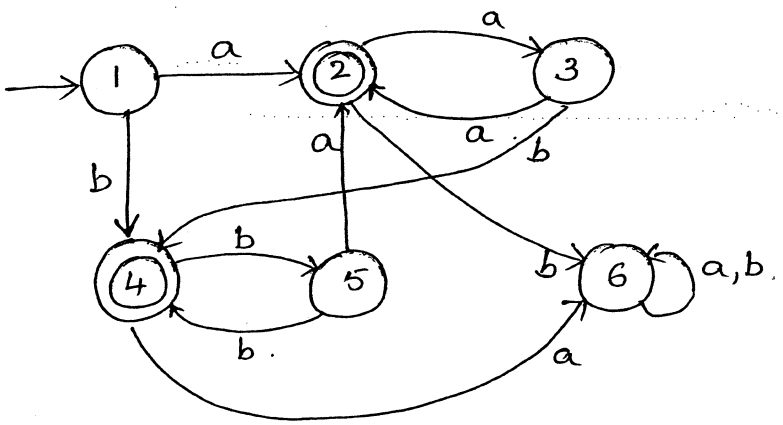
M' is minimal

L(M') = L(M).

Find Minimal machine .

using minDFSM find minimal machine.

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



solution:

1. [2,4] [1,3,5,6]

2. ([2,a], [1,3,5,6]) ([4,a], [1,3,5,6]) NO splitting required.
   ([2,b], [1,3,5,6]) ([4,b], [1,3,5,6])

([1,a], [2,4]) ([3,a], [2,4]), ((5,a), [2 4]) ((6,a) [1,3,5,6])
([1,b], [2,4]) ((3,b), [2,4]) ((5,b), [2,4]) ((6,b) [1,3,5,6])

split the classes as

[1,3,5] [6].

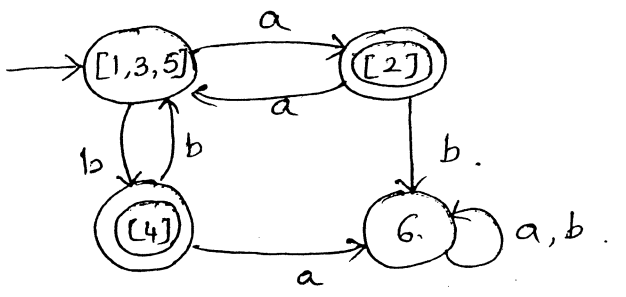
classes = [2,4], [1,3,5], [6].

((2,a) [1,3,5]) ((4,a), [6]) } split.
((2,b) [6]) ((4,b), [1,3,5]) }

classes = [2] [4] [1,3,5], [6].

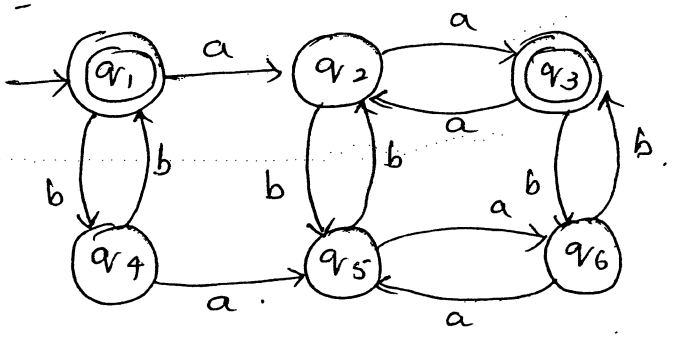
(1,a) [2] (3,a) [2] (5,a) [2] } NO splitting reqd.
(1,b) [4] (3,b) [4] (5,b) [4]. }

M' =



use minDFSM to minimize M.

M =



Solution:-

1. $[q_1\ q_3]\ [q_2\ q_4\ q_5\ q_6]$.

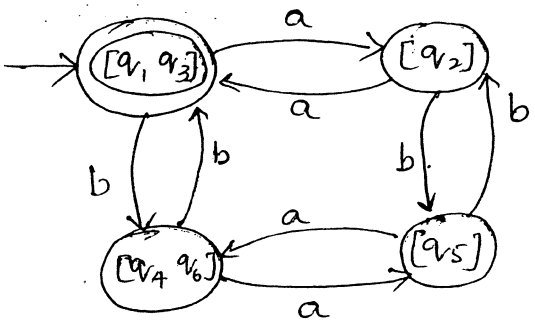
2. $(q_1, a)\ [q_2\ q_4\ q_5\ q_6]$ $(q_3, a)\ [q_2\ q_4\ q_5\ q_6]$ } no split
   $(q_1, b)\ [q_2\ q_4\ q_5\ q_6]$ $(q_3, b)\ [q_2\ q_4\ q_5\ q_6]$

   $(q_2, a)\ [q_1\ q_3]$ $(q_4, a)\ [q_2\ q_4\ q_5\ q_6]$
   $(q_2, b)\ [q_2\ q_4\ q_5\ q_6]$ $(q_4, b)\ [q_1\ q_3]$
   $(q_5, a)\ [q_2\ q_4\ q_5\ q_6]$ $(q_6, a)\ [q_2\ q_4\ q_5\ q_6]$ } split req'd.
   $(q_5, b)\ [q_2\ q_4\ q_5\ q_6]$ $(q_6, b)\ [q_1\ q_3]$.

classes = $[q_1\ q_3]\ [q_2]\ [q_5]\ [q_4\ q_6]$.

$((q_1, a)\ [q_2])$ $((q_3, a)\ [q_2])$ } no split.
$((q_1, b)\ [q_4\ q_6])$ $((q_3, b)\ [q_4\ q_6])$

classes = $[q_1\ q_3]\ [q_2]\ [q_5]\ [q_4\ q_6]$.



CANONICAL FORM OF REGULAR LANGUAGES:

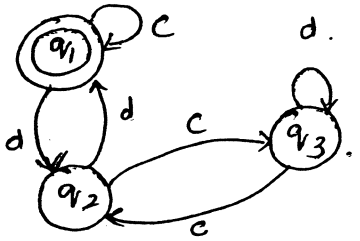
The canonical form for some set of objects C assign exactly one representation to each class of equivalent objects in C. The two objects represented distinctly share same representation iff they are equivalent.

* The two finite state machines are said to be equivalent if both accept the same set of strings, over an input set Σ.
* The two FSM are equivalent, if both of them reach final state after accepting string x over Σ

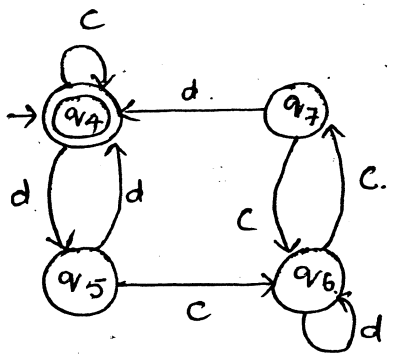
comparing two FA's:-

1. Let M_1 and M_2 be two fSMs.

M_1 =



M_2 =



2. Build MinDFSM. for the above MIc.

M_1' =

$[q_1]$ $[q_2\ q_3]$

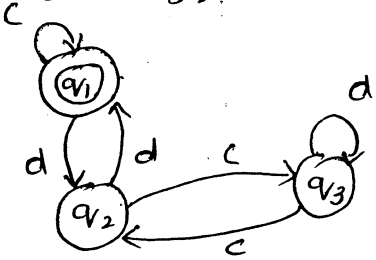
$([q_1, c], [q_1])$ $([q_2, c]$ $[q_2\ q_3])$ $([q_3, c], [q_2\ q_3])$

$([q_1, d], [q_2 q_3]$ $([q_2, d]$ $[q_1])$ $([q_3, d], [q_2\ q_3])$

split read.

$[q_1]$ $[q_2\ q_3]$

$\therefore M_1'$ =



M_2' =

$[q_4]$ $[q_5\ q_6\ q_7]$

$([q_4, c], [q_4])$

$([q_4, d], [q_5\ q_6\ q_7])$

$([q_5, c]\ [q_5\ q_6\ q_7])$, $([q_6, c], [q_5\ q_6\ q_7])$ $([q_7, c], [q_5 q_6 q_7])$

$([q_5, d]\ [q_4])$ $([q_6, d], [q_5\ q_6\ q_7])$ $([q_7, d], [q_4])$

split required.

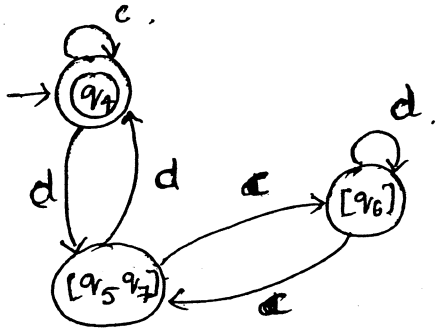
$[q_4]$ $[q_5\ q_7]$ $[q_6]$.

$([q_4, c], [q_4])$

$([q_4, d], [q_5\ q_7])$

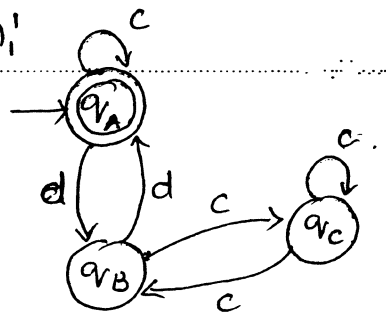
$([q_5, c]\ [q_6])$ $([q_7, c]\ [q_6])$ $([q_6, c], [q_5 q_7])$

$([q_5, d]\ [q_4])$ $([q_7, d]\ [q_4])$ $([q_6, d], [q_6])$



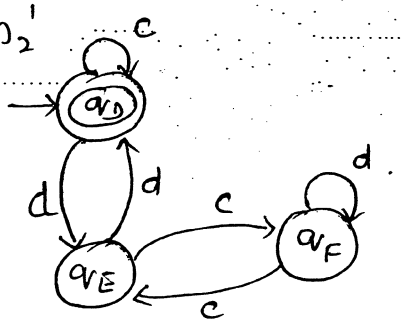
3. Assign different names to the states of M'

Let M_1'



M_1' is canonical form of M_1

M_2'



M_2' is canonical form of M_2

4. construct transition table as pairwise entries (q', q'') where $q' \in M_1'$ and $q'' \in M_2'$ for each input symbol.

	c	d
$(q_A\ q_D)$	$[q_A\ q_D)$	$(q_B\ q_E)$
$(q_B\ q_E)$	$(q_C\ q_F)$	$(q_A\ q_D)$
$(q_C\ q_F)$	$(q_B\ q_E)$	$(q_C\ q_F)$

Build transition table for each i/p symbol Σ_1. In M_1' from q_A on c remains is q_A & M_2' from q_D on c remains is q_D.

From $[q_A\ q_D]$ on c goes to $[q_A\ q_D]$ where q_A & $q_D \in A$

$[q_A\ q_D]$ on d goes to $[q_B\ q_E]$ where q_B & $q_E \in (A-K)$

∴ Next state is

$[q_B\ q_E]$ on c goes to $[q_C\ q_F] \in \{A-K\}$.

$[q_B\ q_E]$ on d goes to $[q_A\ q_D] \in A$.

continues until we donot get new states.

Transition from each of the state also results in either final state or non-final state.

∴ $L(M_1') = L(M_2')$

Hence two FSMs M_1 and M_2 are Equivalent.

build FSM canonical (M: FSM) =

1. M' = ndfsm to DFSM (M)
2. M# = minDFSM (M').
3. create a unique assignment of names to the states of M# as follows:
   - 3.1 call the start state q_0.
   - 3.2. Define an order on the elements of Σ.
   - 3.3 until all states have been named do:
     - select the lowest numbered named state that has not yet been selected call it q.
     - create an ordered list of the transition out of q by the order imposed on their labels.
     - create an ordered list of unnamed states that those transitions enter by doing the following.

       (q, c_1, p_1) then consider p_1 as 1^{st} transition

       $(q\ c_2\ p_2)$ then p_2 is 2^{nd} transition.

       Transitions already exist skip, else continue all transitions have been considered.
     - Name the states as per the ordered list.
4. Return M#.

Given two FSMs M_1 and M_2.

buildFSM canonical (M_1) = buildFSM canonical (M_2) iff $L(M_1) = L(M_2)$.

Finite State Transducers:

Finite state models are augmented for output at each step of machine operation. Finite state transducer are loops that simply run forever, processing inputs.

Finite state transducers generates output whenever M enters associated state and are deterministic are called **Moore Machines.**

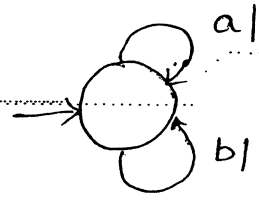
Moore machine M is seven tuple $(K, \Sigma, O, \delta, D, S, A)$ where.

* K is finite set of states
* Σ is input alphabet
* O is output alphabet
* $S \in K$ is start state
* $A \subseteq K$ is set of accepting state.
* δ is transition function defined as $K \times \Sigma \to K$
* D is display or output function.

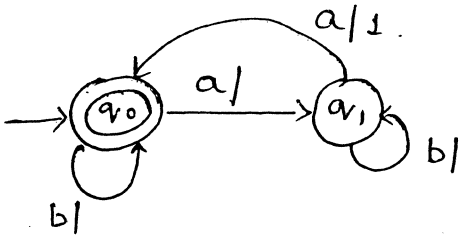
Example:

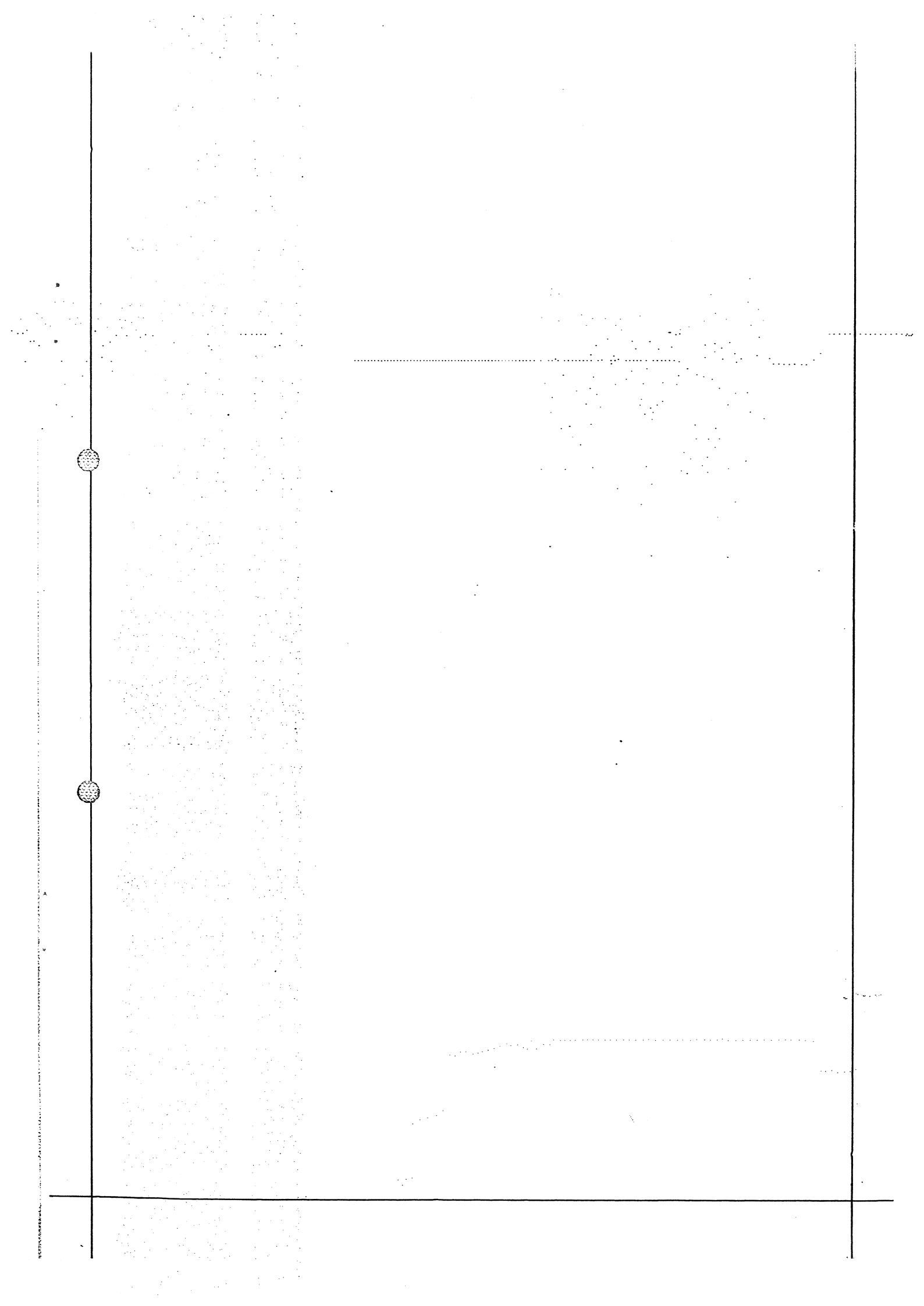
construct a deterministic finite state transducer for the following. $\Sigma = \{a, b\}$.

(a) on i/p w, produce 1^n : where $n = \#_a(w)$.



(b) on i/p w produce 1^n : where $n = \#_a(w)/2$





Mealy machine:

Finite state machine that associates output with transitions are called Mealy machine.

Mealy machine $M = (K, \Sigma_1, O, \delta, S, A)$ where

* K is finite set of states
* Σ_1 is an input alphabet
* O is an output alphabet
* $S \in K$ is start state
* $A \subseteq K$ is Accepting state.
* δ is transition defined as $(K \times \Sigma_1) \rightarrow (K \times O^*)$

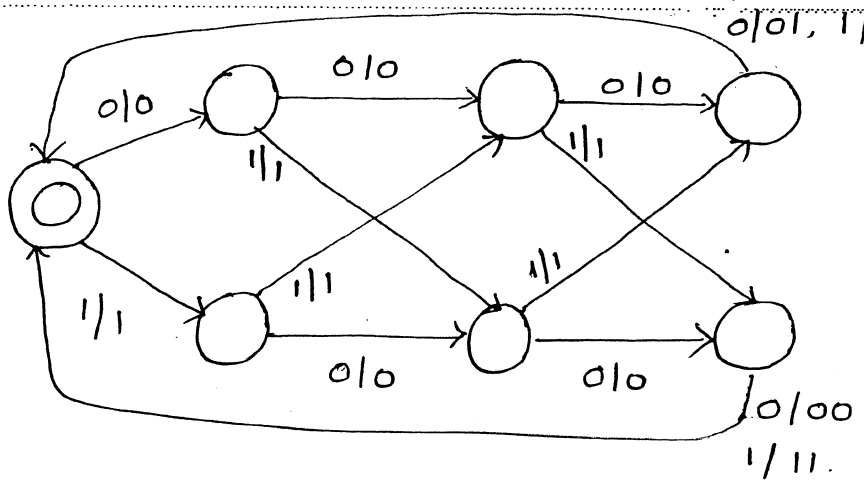
Mealy machine M computes a function $f(w)$ iff, when it reads i/p string w, its o/p sequence is $f(w)$.

Example. An odd parity generator:

After every four bits, output a fifth bit such that each group of four bits has odd parity.

i/p bits	parity bit
0000	1
0001	0
0010	0
0011	1
0100	0
0101	1
0110	1
0111	0

i/p bits	parity bits
1000	0
1001	1
1010	1
1011	0
1100	1
1101	0
1100	0
1111	1



Deterministic Finite state Transducer Interpreter.

Let:

δ_1 (state, symbol) return a single new state &

δ_2 (state, symbol) return an element of O^*

ST := S

Repeat

i := get-next symbol.

If $i \neq eof$ then:

while (δ_2 (ST, i)).

ST := δ_1 (ST, i).

until i = eof

Mealy Machine	Moore machine.
1. output depends both upon the present state and the present input	output depends only upon the present state
2. Less number of states than moore machine	More number of states
3. The value of the o/p fn is a function of the transitions and the changes when the i/p logic on the present state is done.	The value of the o/p fn is a function of the current state at the clock edges whenever state changes occur.
4. Mealy m/c react faster to i/ps.	More logic is required to decode the o/ps resulting in ckt delays.

Bidirectional Transducers:-

A process that reads an input string and constructs a corresponding output string can be described using finite state transducer. [FST].

FST provides a declarative way to describe the relationship between inputs an outputs. Such declarative model can be run in two directions hence the name Bidirectional transducers.

Example:-

(i) Letter substitution:

Language L defined over the alphabet {a,b} is also true for L' that contains exactly the strings in L where every 'a' is replaced by '0' & b is replaced by '1'

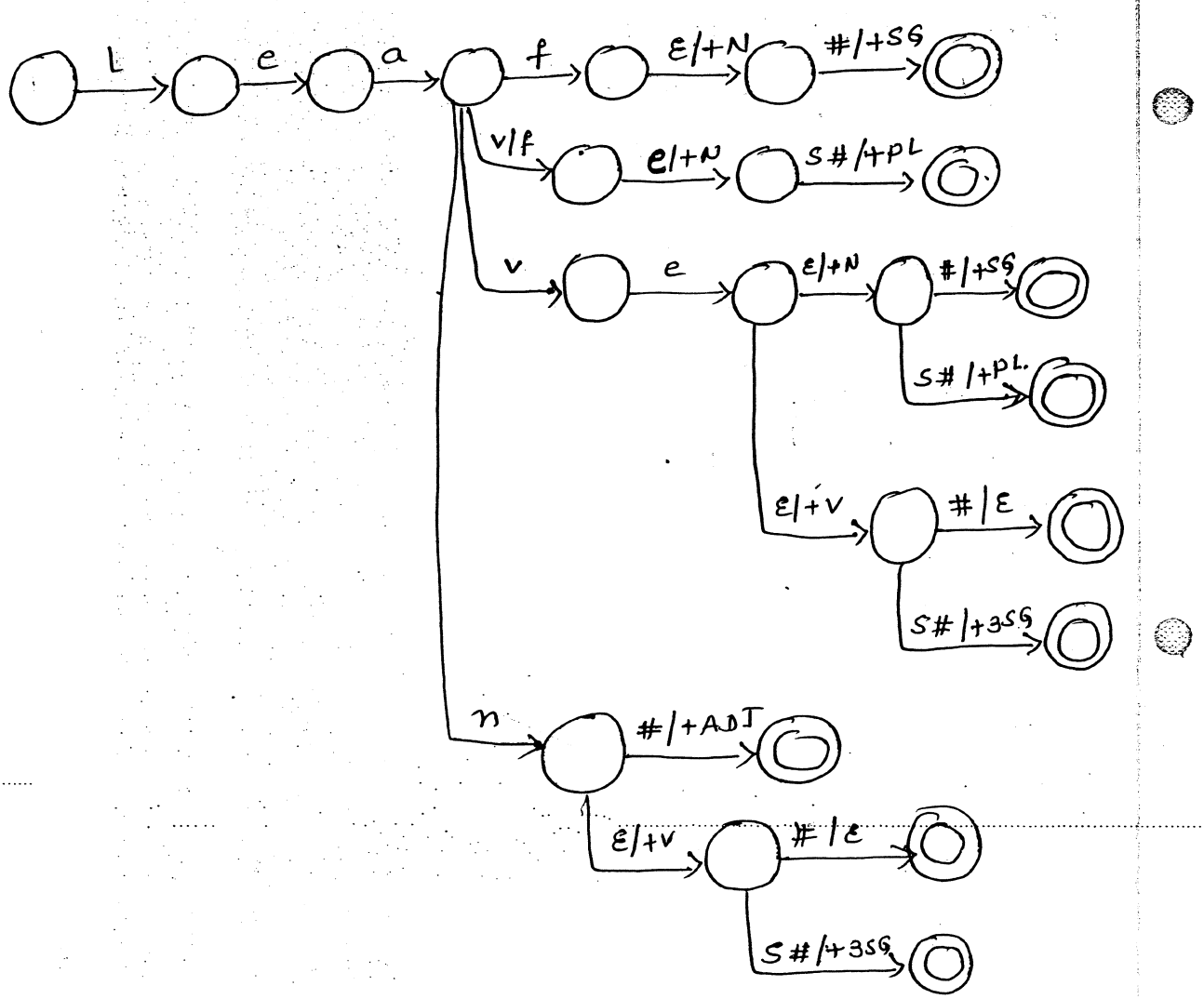
Bidirectional transducer can convert L to strings in L' and vice versa.

(ii) English morphology.

morpheme is the smallest linguistic unit to which meaning can be assigned.

Simple transducer that performs the following mappings.

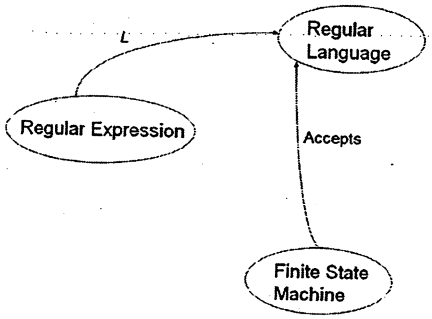
* Leaf → Leaf + Noun [N] + singular [SG]
* Leave → Leave + N + SG
* Leave → Leave + verb [V]
* Leaves → Leaf + N + plural [PL]
* Leaves → Leave + V + 3 SG (3rd person singular).
* Lean → Lean + ADJ
* lean → Lean + V.



K. S. Sampada
Asst. professor
Dept. of CSE
RNSIT

MODULE_2

REGULAR EXPRESSIONS



The regular expressions over an alphabet Σ are all and only the strings that can be obtained as follows:

1. ∅is a regular expression. denoting empty language.
2. εis a regular expression. indicating the language containing empty string.
3. Every element of Σ is a regular expression.
4. If α , β are regular expressions, then so is αβ.
5. If α , β are regular expressions, then so is α∪β.
6. If α is a regular expression, then so is α*.
7. α is a regular expression, then so is α^+.
8. If α is a regular expression, then so is (α).

Examples

If Σ = {a, b}, the following are regular expressions:

∅

ε

a

(a∪ b)*

abba∪ε

Regular Expressions Define Languages

Define *L*, a **semantic interpretation function** for regular expressions:

1. $L(\emptyset) = \emptyset$.language contains no strings
2. $L(\varepsilon) = \{\varepsilon\}$. Language containing empty string
3. $L(c)$, where $c \in \Sigma = \{c\}$. Language containing single string
4. $L(\alpha\beta) = L(\alpha)\ L(\beta)$. Concatenation of 2 languages
5. $L(\alpha \cup \beta) = L(\alpha) \cup L(\beta)$. Union of 2 constituent languages
6. $L(\alpha^*) = (L(\alpha))^*$. * is the Kleene star operator

7. $L(\alpha^+) = L(\alpha\alpha^*) = L(\alpha)\,(L(\alpha))^*$. If $L(\alpha)$ is equal to $\varnothing$, then $L(\alpha^+)$ is also equal to $\varnothing$. Otherwise $L(\alpha^+)$ is the language that is formed by concatenating together one or more strings drawn from $L(\alpha)$.

8. $L((\alpha)) = L(\alpha)$.

The Role of the Rules

- Rules 1, 3, 4, 5, and 6 give the language its power to define sets.
- Rule 8 has as its only role grouping other operators.
- Rules 2 and 7 appear to add functionality to the regular expression language, but they don't.

Write a regular expression to describe each of the following languages:

a) **{$w \in$ {a, b}* : every a in w is immediately preceded and followed by b}.**

(b ∪ bab)*

b) **{$w \in$ {a, b}* : w does not end in ba}.**

ε ∪ a ∪ (a ∪ b)* (b ∪ aa)

c) **{$w \in$ {0, 1}* : $\exists y \in$ {0, 1}* (|xy| is even)}.**

(0 ∪ 1)*

d) **{$w \in$ {0, 1}* : w corresponds to the binary encoding, without leading 0's, of natural numbers that are evenly divisible by 4}.**

(1(0 ∪ 1)* 00) ∪ 0

e) **{$w \in$ {0, 1}* : w corresponds to the binary encoding, without leading 0's, of natural numbers that are powers of 4}.**

1(00)*

f) **{$w \in$ {0-9}* : w corresponds to the decimal encoding, without leading 0's, of an odd natural number}.**

(ε ∪ ((1-9)(0-9)*))(1 ∪ 3 ∪ 5 ∪ 7 ∪ 9)

g) **{$w \in$ {0, 1}* : w has 001 as a substring}.**

(0 ∪ 1)* 001 (0 ∪ 1)*

h) **{$w \in$ {0, 1}* : w does not have 001 as a substring}.**

(1 ∪ 01)* 0*

i) **{$w \in$ {a, b}* : w has bba as a substring}.**

(a ∪ b)* bba (a ∪ b)*

Regular Exp^n	Meaning
a^*	string with any no of a's [0 or more]
a^+	string with atleast 1 a or more.
$(a+b)$	string consisting of either 1 a or 1b
$(a+b)^*$	strings of a's & b's of any length [0 or more].
$(a+b)^*$ abb	strings of a's & b's ending with abb.
ab $(a+b)^*$	strings of a's & b's starting with ab
$(a+b)^*$ $(a+bb)$	strings of a's & b's ending with either a or bb
$(a+b)^*$ aa $(a+b)^*$	strings of a's & b's containing substring 'aa'

obtain Regular expression to accept the language

(i) a's & b's having length 2.

$= (a \cup b)(a \cup b)$

$= (a \cup b)^2$

(ii) strings of a's & b's having even length.

$= ((a \cup b)(a \cup b))^*$

or

$= ((a \cup b)^2)^* \Rightarrow (aa \cup ab \cup ba \cup bb)^*$

(iii) To accept Language containing 'aaa' as substring

$(a \cup b)^* aaa (a \cup b)^*$

K.S. sampada
Asst. prof. CSE
RNSIT

(iv) strings of a's & b's starting with a

$$a.(a \cup b)^*$$

(v) strings of a's & b's whose second symbol from the last is 'a'

1st last symbol $(a \cup b)$

second from last symbol a (given).

3rd from last symbol $(a \cup b)^*$

$$= (a \cup b)^* \; a \; (a \cup b)$$

(vi) $L = \{w \mid w \in \Sigma : |w| \bmod 3 = 0\}$.

$$= \left((a \cup b)^3\right)^*$$

or

$$= \left((a \cup b)\,(a \cup b)\,(a \cup b)\right)^*$$

(vii) strings of 2 or more letters beginning or ending with same letter.

$$a\,(a \cup b)^*\,a \;\cup\; b\,(a \cup b)^*\,b$$

(viii) $L = \{w \mid w \in \Sigma$; no two adjacent characters are same.

$$(a \cup \varepsilon)\,(ba)^*\,(b \cup \varepsilon) \cup (b \cup \varepsilon)\,(ab)^*\,(a \cup \varepsilon)$$

(ix) $L = \{a^n b^m \mid m+n$ is even$\}$.

$$(aa)^*.(bb)^* \cup a(aa)^* \cup b(bb)^*$$

Analyzing a Regular Expression

1. $L((a \cup b)^*b) = L((a \cup b)^*)\ L(b)$
   $= (L((a \cup b)))^*\ L(b)$
   $= (L(a) \cup L(b))^*\ L(b)$
   $= (\{a\} \cup \{b\})^*\ \{b\}$
   $= \{a, b\}^*\ \{b\}.$

   Strings of a,b that ends with b.

Examples

1. $L(\ a^*b^*\ ) = a^*b^*$
2. $L(\ (a \cup b)^*\ ) = (a \cup b)^*$
3. $L(\ (a \cup b)^*a^*b^*\ ) = L\ (a \cup b)^*\ L(a^*)\ L(b)^*$
   $= \{a,b\},\{a\},\{b\}$
4. $L(\ (a \cup b)^*abba(a \cup b)^*\ ) = L(a \cup b)^*\ abba\ L(a \cup b)^*$
   $= \{a,b\}\ abba\ \{a,b\}$

   String consisting of abba as substring
5. $L = \{w \in \{a, b\}^*: |w| \text{ is even}\}$ find RE
   $=(a \cup b)\ (a \cup b))^*$ *or*
   $=(aa \cup ab \cup ba \cup bb)^*$
6. $L = \{w \in \{a, b\}^*: w \text{ contains an odd number of a's}\}$ find RE.
   $=b^*\ (ab^*ab^*)^*\ a\ b^*$ *or*
   $=b^*\ a\ b^*\ (ab^*ab^*)^*$

The regular expression language provides the following 3 operations and its precedence

Highest ↕ Lowest	Regular Expressions	Arithmetic Expressions
	Kleene star	Exponentiation
	concatenation	multiplication
	union	Addition

(a U bb*a) will be interpreted as (a U (b(b*a)))

The Details Matter

1. $a^* \cup b^* \neq (a \cup b)^*$
2. $(ab)^* \neq a^*b^*$

Kleen's Theorem.

Finite state machines and regular expressions define the same class of languages. To prove this,

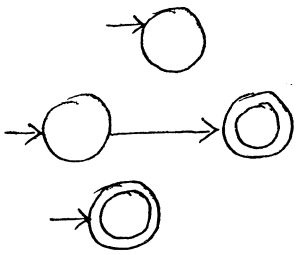
> **Theorem 6.1: FOR EVERY REGULAR EXPRESSION THERE IS A CORRESPONDING FSM.**
>
> Any language that can be defined with a regular expression can be accepted by some FSM & so is regular.

proof: By construction. construct an FSM M such that $L(\alpha) = L(M)$. An FSM for

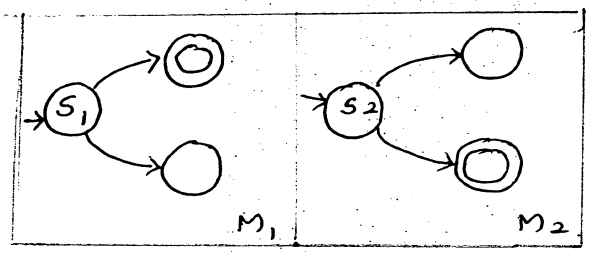
ϕ

single element of Σ:

$\varepsilon(\phi^*)$



UNION: If α is the regular expression $\beta \cup \gamma$ and if both $L(\beta)$ and $L(\gamma)$ are regular, construct



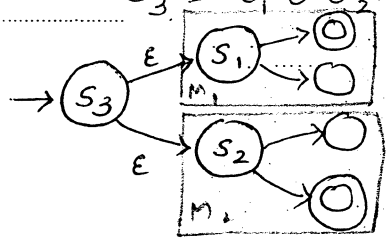
$M_3 = (K_3, \Sigma, \delta_3, S_3, A_3)$ such that $L(M_3) = L(M_1) \cup L(M_2)$.

$\Rightarrow L(\alpha) = L(\beta) \cup L(\gamma)$

create a new start state S_3 & connect it to start state of M_1 & M_2 via ε-transitions so.

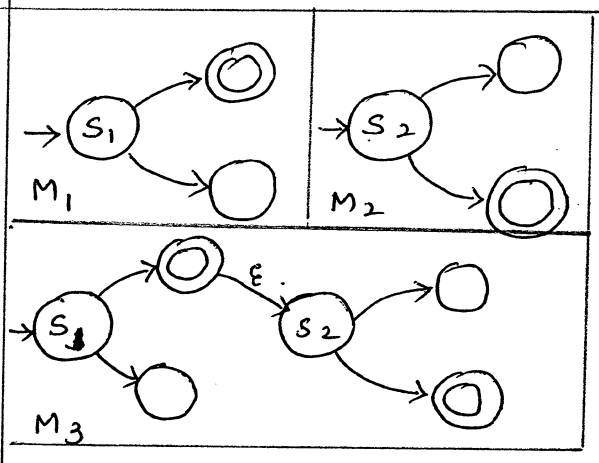
$$M_3 = (\{S_3\} \cup K_1 \cup K_2, \Sigma, \delta_3, S_3, A_1 \cup A_2)$$

where $\delta_3 = \delta_1 \cup \delta_2 \cup \{((S_3, \varepsilon) S_1), ((S_3, \varepsilon) S_2)\}$.



concatenation:- If α is the R.E $\beta\gamma$ and if both $L(\beta)$ & $L(\gamma)$ are regular, then construct. $M_3 = (K_3, \Sigma, \delta, S_3, A_3)$ such that $L(M_3) = L(M_1) . L(M_2)$.

$\Rightarrow L(\alpha) = L(\beta) . L(\gamma)$.

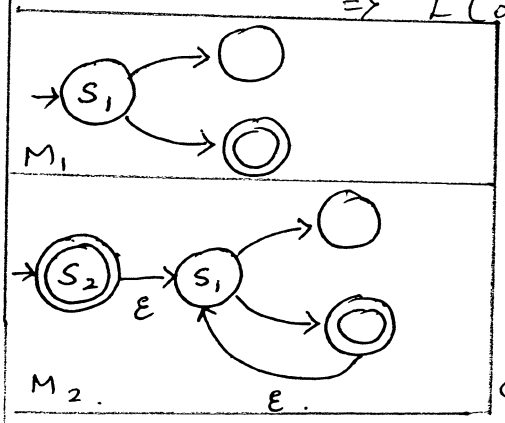


Build M_3 by connecting every accepting state of M_1 to the start state of M_2 via ε-transitions.

$\delta_3 = (\delta_1 \cup \delta_2) \cup ((q, \varepsilon), S_2) : q \in A_1)$

Kleen star: If α is the regular expression β^* and if $L(\beta)$ is regular then construct $M_2 = (K_2, \Sigma_1, \delta, S_2, A_2)$ such that $L(M_3) = L(M_1)^*$

$\Rightarrow L(\alpha) = L(\beta)^*$.



create a new start state S_2 and make it as accepting state.
connect S_2 to S_1 via ε-transition.
from each of M_1's accepting state back to S_1 create ε-transitions.

$\therefore M_2 = (\{S_2\} \cup K_1, \Sigma_1, \delta_2, S_2, \{S_2\})$

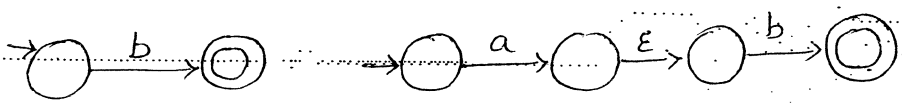
$\delta_2 = \delta_1 \cup \{((S_2, \varepsilon) S_1)\} \cup \{((q, \varepsilon) S_1) : q \in A\}$

Examples:

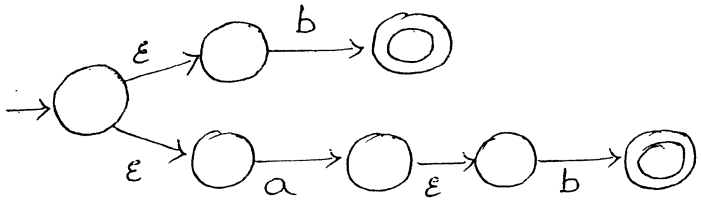
(i) $(b \cup ab)^*$

Solution:

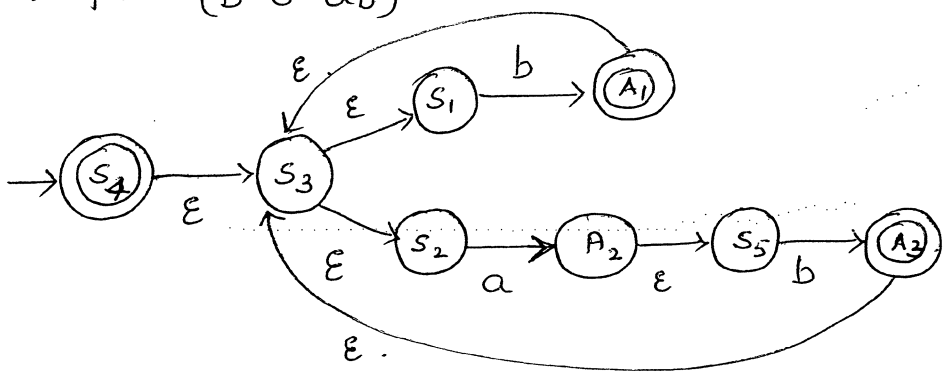
FSM for b. fsm for 'ab



FSM for $(b \cup ab)^*$



FSM for $(b \cup ab)^*$



Algorithm regex to FSM

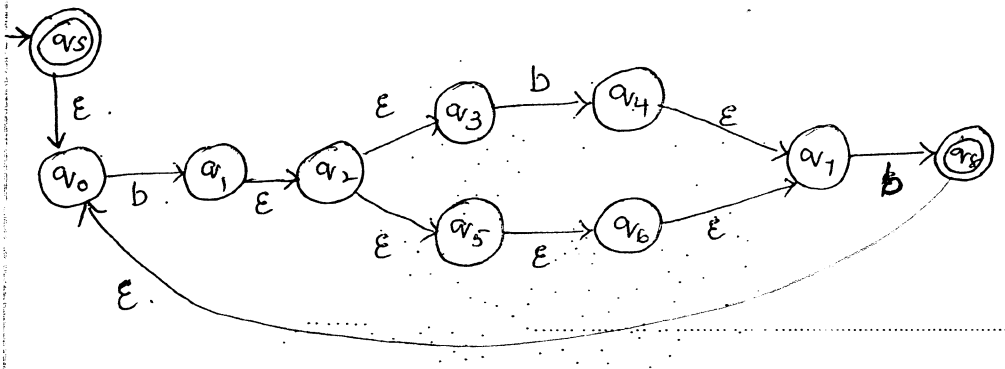
regextofsm (α : regular expression) =

Beginning with primitive subexpressions of α and working outwards until an FSM for all of α has been built do:

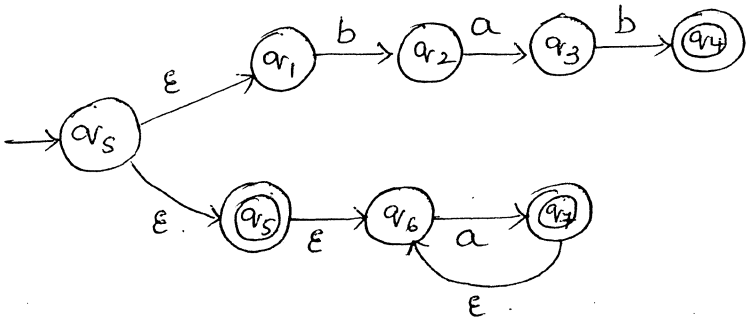
construct an FSM as described above.

Use the algorithm regextofsm to construct FSM to accept the languages generated by following regular expressions:

(a) $(b\ (b \cup \varepsilon)\ b)^*$

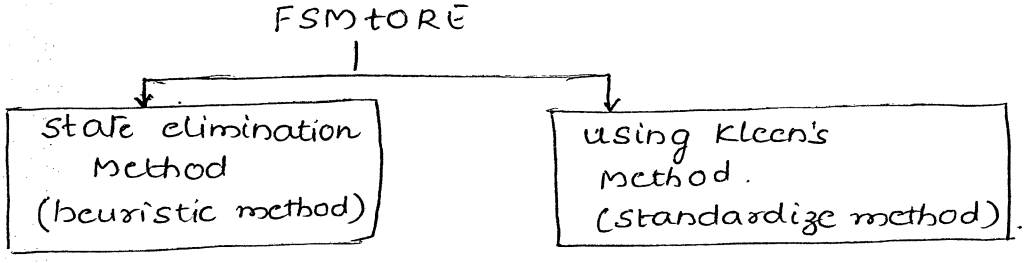


(b) $bab \cup a^*$



Building a Regular Expression from an FSM:

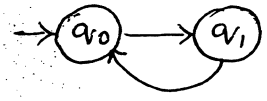
There are two methods to obtain RE from FSM



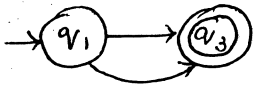
Conversion from FSM to RE. [State elimination method]

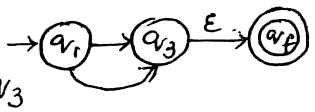
Given arbitrary FSM M, M' will be built by starting with M and then eliminating one state at a time, all the states that lie between start state & accepting state.

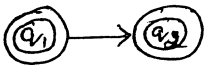
Rules:

(1)  : If any incoming transition to start state:

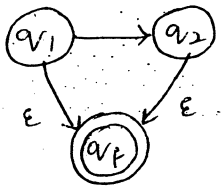
create a new start state & have an ε-transition to start state 

(2)  If final state has outgoing transition

create a new accepting state & have an ε-transition to it from q_3 

(3) If more than one accepting state 

then create a new accepting state and have ε-transitions from each one of them to new final state.



Building a regular Expression from an FSM:

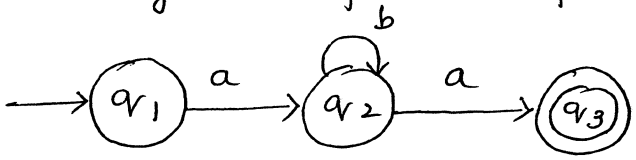
fsmtoregexheuristic (M : FSM) =

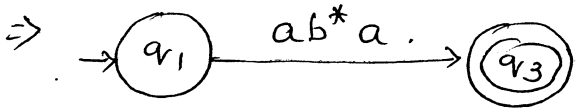
1. Remove from M any states that are unreachable from the start state.
2. If M has no accepting states then halt and return the simple regular expression ϕ.
3. If the start state of M is part of a loop, create a new start state S and connect S to M's start state via an ε-transition. This new start state S will have no transitions into it.
4. If there is more than one accepting state of M or if there is just one but there are any transitions out of it, create a new accepting state and connect each of M's accepting states to it via ε-transition. [Remove old accepting states from set of accepting states].
   * New accepting state will have no-transitions out of it.
5. $L(M) = \{\varepsilon\}$ if M has only one state which is both start state & accepting state & M has no transitions.
6. until only the start state & accepting state remain do:
   - 6.1 Select some state rip of M. Any state other than accepting state & start state can be choosen.
   - 6.2. Remove rip from M.
   - 6.3. Modify the transitions among the remaining states so that M accepts same string.

7. Return regular expression that labels the one remaining transition from the start state to accepting state.

Example - 1.

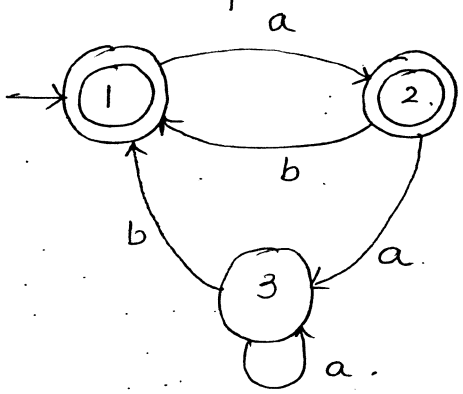
1. find regular expression for M



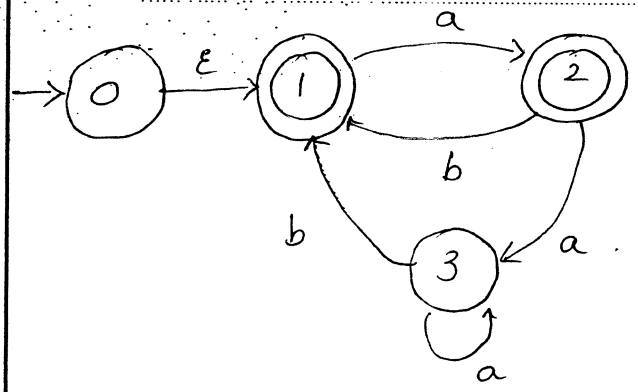


R.E = ab^*a.

2. Build R.E for an FSM.

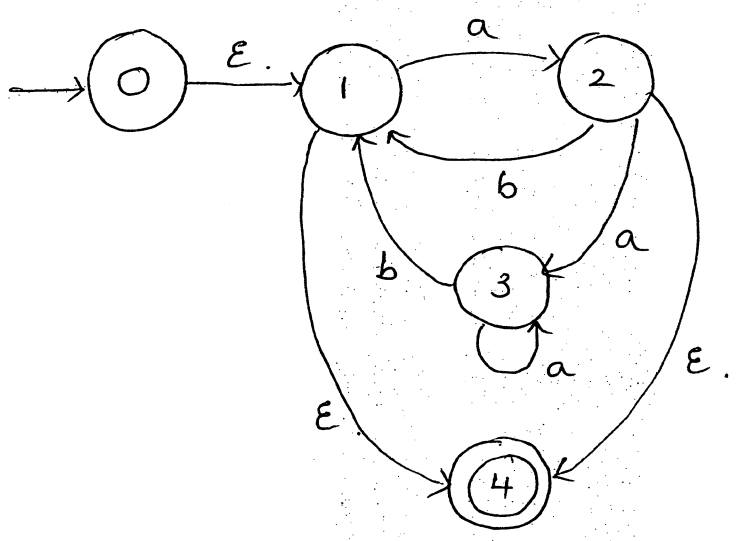


Step-1:- create a new start state.

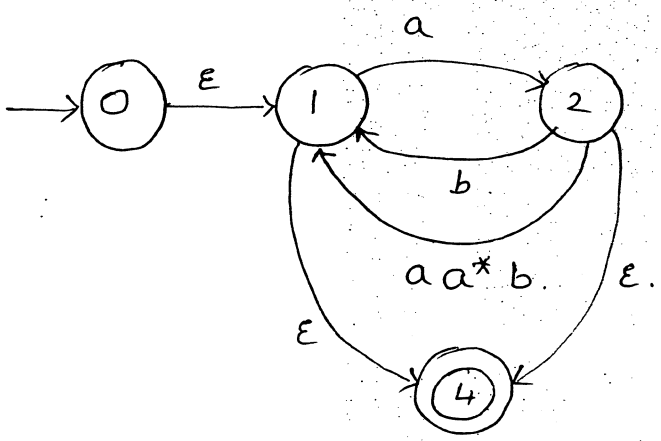


Ctrl.

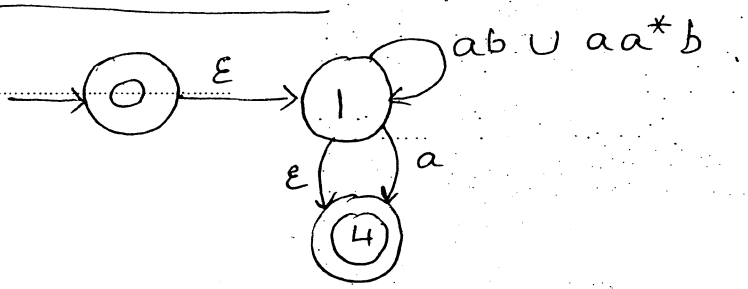
Step-2 :- create a new accepting state.



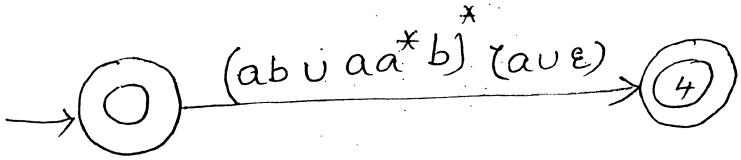
Step-3 :- Remove state-3



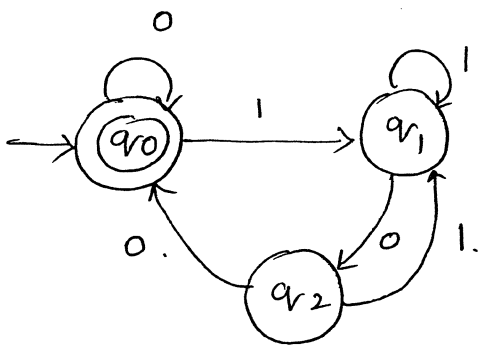
Remove state-2



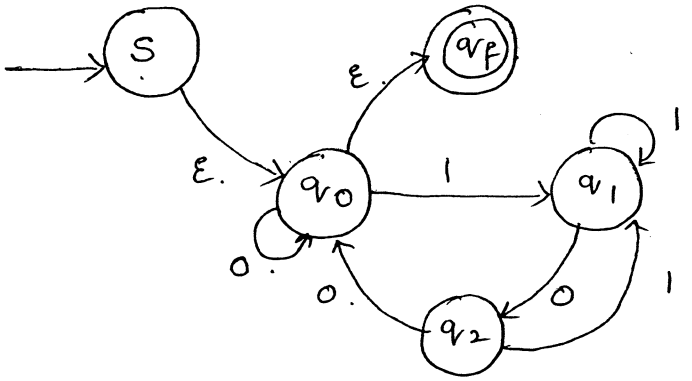
Remove state-1



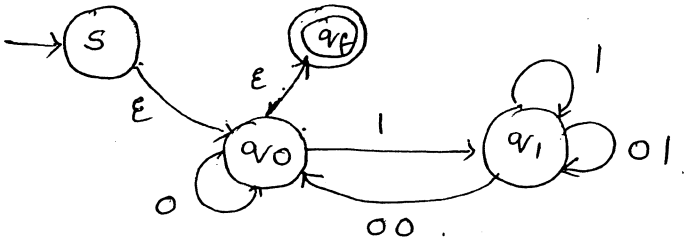
(3)



Step-1 :- Create a new start state & accepting state.

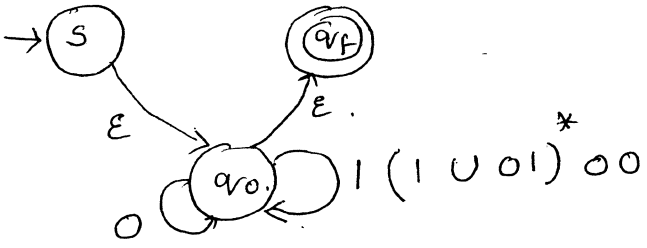


Step-2 :- Remove q_2.

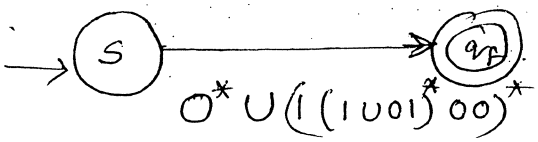


Remove q_1

Remove q_0

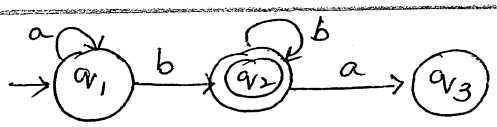


=>

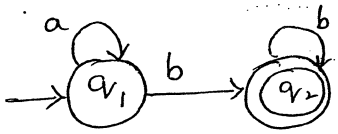


R.E => $\left(0 \cup [1\,(1 \cup 01)^{*}\,00]\right)^{*}$

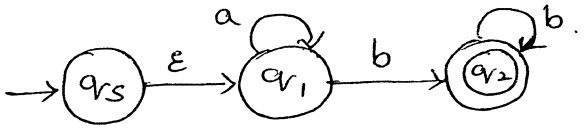
(4)



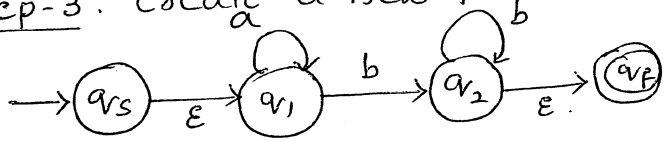
Step-1 :- Remove q_3 as it is unreachable state.



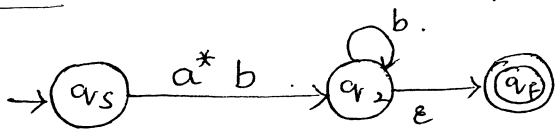
Step-2 : Create a new state q_s



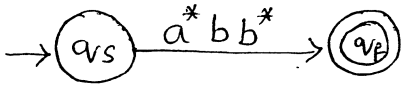
Step-3 : Create a new final state q_f.



Step-4 : Remove state q_1

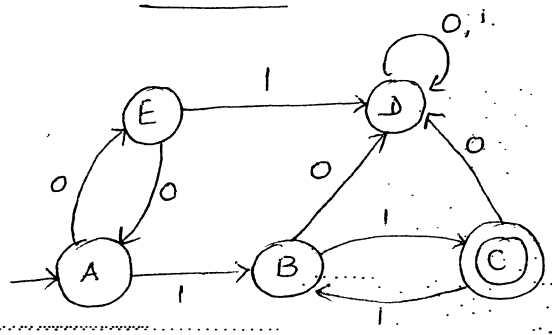


Step-5 : Remove state q_2.

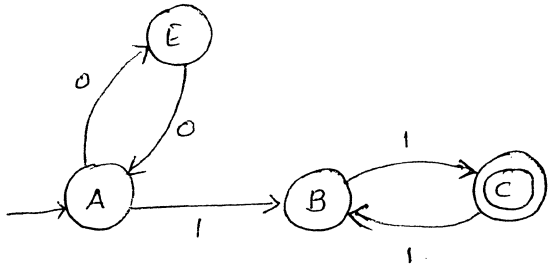


∴ R.E $a^* b b^*$

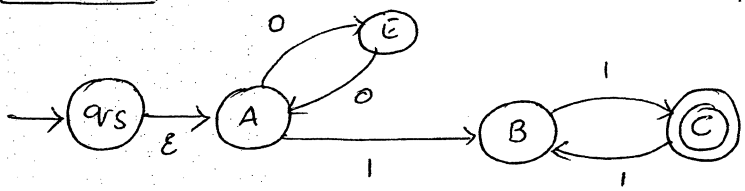
(5)



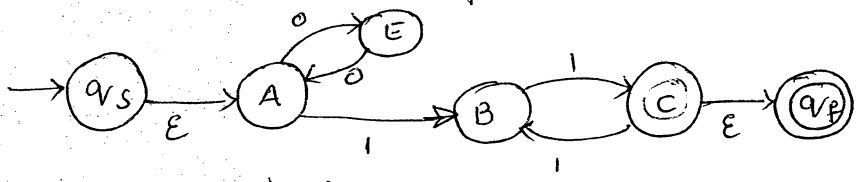
Step-1 : D is unreachabe remove.



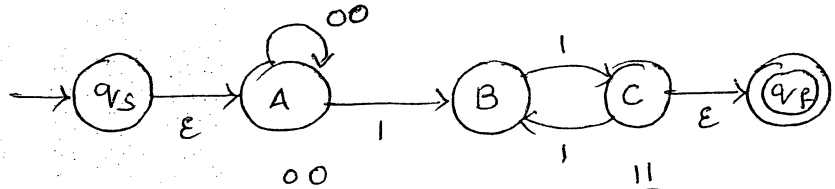
Step-2 :- Create a new start state.

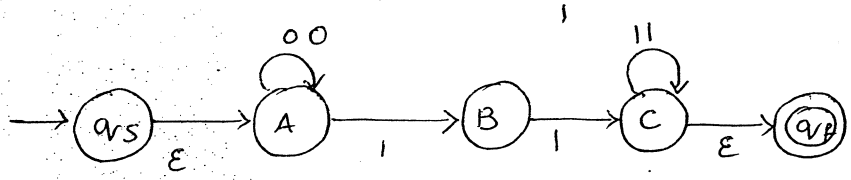


Create a new final state.

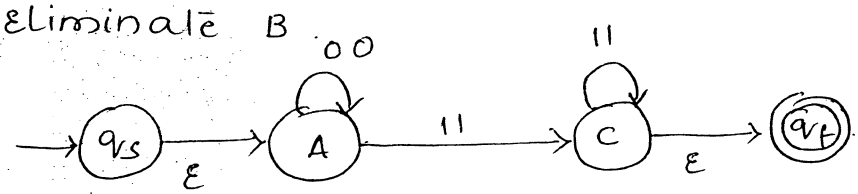


Eliminate 'E'

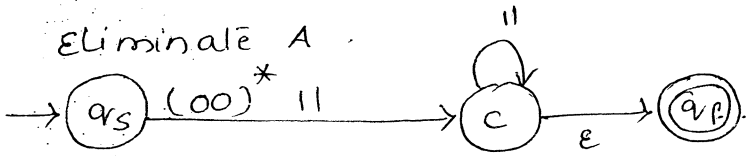




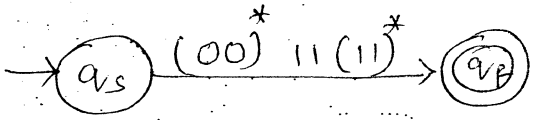
Eliminate B



Eliminate A



Eliminate C.

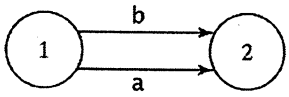


RE $(00)^* 11 (11)^*$

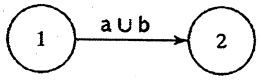
Further Modifications to *M* Before We Start

We require that, from every state other than the accepting state there must be exactly one transition to every state (including itself) except the start state. And into every state other than the start state there must be exactly one transition from every state (including itself) except the accepting state.

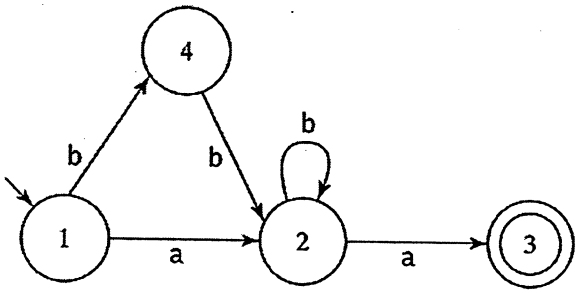
1. *If there is more than one transition between states p and q, collapse them into a single transition*:



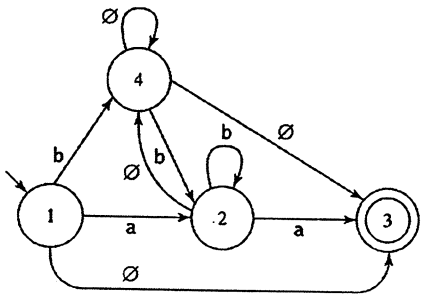
becomes:



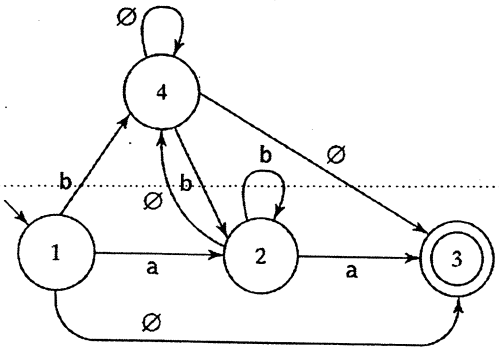
2. *If any of the required transitions are missing, add them:*



becomes



3. *Choose a state. Rip it out. Restore functionality.*



Suppose we rip state 2.

Building RE from FSM using Kleen's method.

Let L be the set of language accepted by FSM. M, where $M = (\{q_1, q_2, \ldots q_n\}, \Sigma, \delta, q_1, F)$ Let.

r_{ij}^{K} denotes the set of strings x such that $\delta(q_i, x) = q_j$. where $q_i \rightarrow$ source state & $q_j \rightarrow$ target state. The inputs are going through the states of automata i.e some i/ps entering into states & some coming out of it. The value of K is always less than i or j.

$$r_{ij}^{K} = r_{ij}^{K-1} \cup r_{ik}^{K-1} \cdot \left(r_{kk}^{K-1}\right)^{*} r_{kj}^{K-1}$$

$$r_{ij}^{0} = \begin{bmatrix} \{\alpha \mid \delta(q_i, \alpha) = q_j\} & \text{if } i \neq j \\ \{\alpha \mid \delta(q_i, \alpha) = q_j\} \cup \{\varepsilon\} & \text{if } i = j \end{bmatrix}$$

Consider any pair of states p and q. Once we remove *rip*, how can M get from p to q?

- It can still take the transition that went directly from p to q, or
- It can take the transition from p to *rip*. Then, it can take the transition from *rip* back to itself zero or more times. Then it can take the transition from *rip* to q.

Defining $R(p, q)$

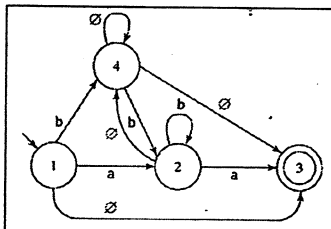
After removing *rip*, the new regular expression that should label the transition from p to q is:

$R(p, q)$	/* Go directly from p to q
$\cup$	/* or
$R(p, rip)$	/* Go from p to *rip*, then
$R(rip, rip)^*$	/* Go from *rip* back to itself any number of times, then
$R(rip, q)$	/* Go from *rip* to q

Without the comments, we have:

$$R' = R(p, q) \cup R(p, rip)\, R(rip, rip)^*\, R(rip, q) \quad \Rightarrow \quad \gamma_{ij}^{k} = \gamma_{ij}^{k-1} \cup \gamma_{ik}^{k-1} \cdot \left(\gamma_{kk}^{k-1}\right)^{*} \gamma_{kj}^{k-1}$$

Example



$R' = R(p, q) \cup R(p, rip)\, R(rip, rip)^*\, R(p, rip)$

Let *rip* be state 2. Then:

$R(1, 3) = R(1, 3) \cup R(1, rip) R(rip, rip)^* R(rip, 3)$

$= R(1, 3) \cup R(1, 2) R(2, 2)^* R(2, 3)$

$= \emptyset \cup a\ b^*\ a$

$= ab^*a$

The Algorithm *fsmtoregex*

fsmtoregex(M: FSM) =

1. $M' =$ *standardize*(M: FSM).
2. Return *buildregex*(M').

standardize(M: FSM) =

1. Remove unreachable states from M.
2. If necessary, create a new start state.
3. If necessary, create a new accepting state.
4. If there is more than one transition between states p and q, collapse them.
5. If any transitions are missing, create them with label ∅.

The Algorithm *fsmtoregex*

buildregex(M: FSM) =

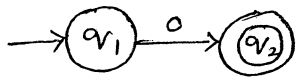
1. If M has no accepting states then return ∅;
2. If M has only one state, then return ε.
3. Until only the start and accepting states remain do:
   - 3.1 Select some state *rip* of M.
   - 3.2 For every transition from p to q, if both p and q are not *rip* then do

Compute the new label R' for the transition from p to q:

$$R'(p, q) = R(p, q) \cup R(p, rip)\, R(rip, rip)^*\, R(rip, q)$$

   - 3.3 Remove *rip* and all transitions into and out of it.
4. Return the regular expression that labels the transition from the start state to the accepting state.

(1) obtain RE for FSM



Solution :

$K = 0$

γ_{11}	ε
γ_{12}	0
γ_{21}	ϕ
γ_{22}	ε.

$K = 1$.

γ_{11}^{1}	$=$	$\gamma_{11}^{0} \cup \gamma_{11}^{0} (\gamma_{11}^{0})^{*} \gamma_{11}$.	
	$=$	$\varepsilon \cup \varepsilon(\varepsilon)^{*} . \varepsilon$.	ε.
	$=$	ε.	
γ_{12}^{1}	$=$	$\gamma_{12}^{0} \cup \gamma_{12}^{0} (\gamma_{12}^{0})^{*} \gamma_{12}^{0}$	0
	$=$	$0 \cup \varepsilon . (\varepsilon)^{*} . 0$.	
	$=$	$0 \cup \varepsilon . 0$.	
		0	
γ_{21}^{1}	$=$	$\gamma_{21}^{0} \cup \gamma_{21}^{0} (\gamma_{21}^{0})^{*} \gamma_{11}^{0}$	ϕ
		$\phi \cup \phi . (\varepsilon)^{*} \varepsilon$.	
		ϕ	
γ_{22}^{1}	$=$	$\gamma_{22}^{0} \cup \gamma_{21}^{0} (\gamma_{11}^{0})^{*} . \gamma_{12}^{0}$	
	$=$	$\varepsilon \cup \phi . (\varepsilon)^{*} . 0$.	ε.
	$=$	ε.	

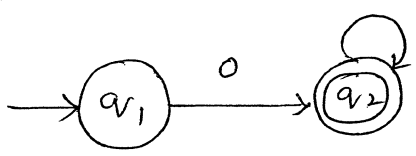
$$\gamma_{12}^{2} = \gamma_{12}^{1} \cup \gamma_{12}^{1} . (\gamma_{22}^{1})^{*} \gamma_{22}^{1}$$

$$= 0 \cup 0 (\varepsilon)^{*} . \varepsilon$$

$$= 0 \cup 0 . \varepsilon$$

$$\boxed{\gamma_{12}^{2} = 0}$$

(2) obtain R.E



Soln:

$$r_{12}^{2} = r_{12}^{1} \cup r_{12}^{1} \cdot (r_{22}^{1})^{*} \cdot r_{22}^{1}$$

$K = 0$.

r_{11} — $\not\emptyset$ ε

r_{12} — 0

r_{21} — ϕ

r_{22} — $1 + \varepsilon$

$K = 1$.		
r_{11}^{1}	$r_{11}^{0} \cup r_{11}^{0} \cdot (r_{11}^{0})^{*} r_{11}$ $\varepsilon \cup \varepsilon \cdot \varepsilon^{*} \varepsilon$	ε
r_{12}^{1}	$r_{12}^{0} \cup r_{11}^{0} (r_{11}^{0})^{*} r_{12}^{0}$ $0 \cup \varepsilon \cdot \varepsilon^{*} \cdot 0 \Rightarrow 0 \cup 0$	0
r_{21}^{1}	$r_{21}^{0} \cup r_{21}^{0} \cdot (r_{11}^{0})^{*} r_{11}^{0}$ $\phi \cup \phi \cdot \varepsilon^{*} \cdot \varepsilon \Rightarrow \phi$	ϕ
r_{22}^{1}	$r_{22}^{0} \cup r_{21}^{0} (r_{11}^{0})^{*} r_{12}^{0}$ $(1+\varepsilon) \cup (\phi) \varepsilon \cdot \varepsilon^{*} 0 \Rightarrow (1+\varepsilon)$	$1 + \varepsilon$

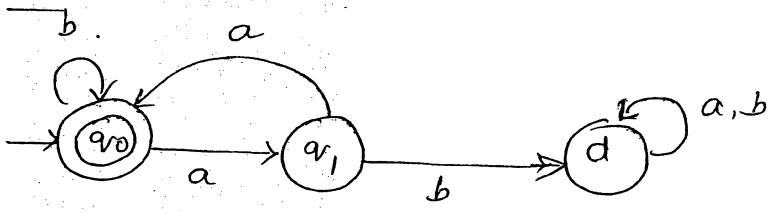
$K = 2$.

$$r_{12}^{2} = r_{12}^{1} \cup r_{12}^{1} (r_{22}^{1})^{*} r_{22}^{1}$$

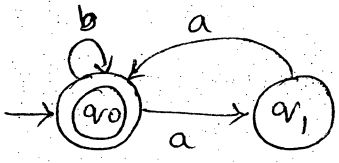
$$= 0 \cup 0 \cdot (1+\varepsilon)^{*} \cdot 1 + \varepsilon$$

$$= 0 \cup 0 1^{*}$$

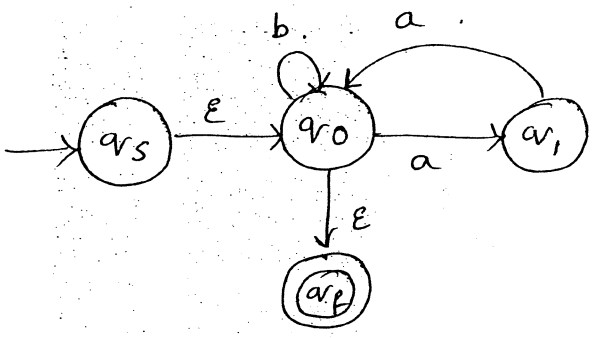
obtain rE for the FSM given below (Excercise 6th 10 problem)



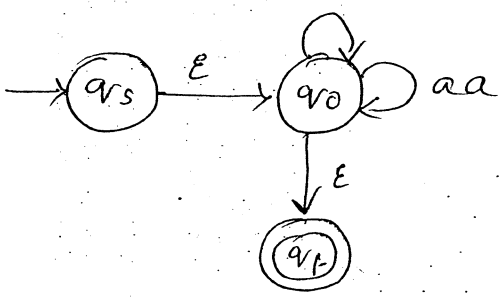
Step-1 : remove d



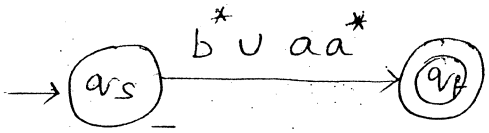
create new start state & accepting state



remove q_1



remove q_0



Equivalence of RE & FSM's :-

Theorem: The class of languages that can be defined with regular expression is exactly the class of regular languages.

proof :- every language that can be defined with a regular expression is regular. Every regular language can be defined by some regular expression.

Writing regular expression is easy: as we need to show only RE that defines it.

(i) No more than one b :-

$$a^* (b \cup \varepsilon) a^*$$

(ii) No two consecutive letters are same.

$$(b \cup \varepsilon)(ab)^* (a \cup \varepsilon)$$

or $(a \cup \varepsilon)(ba)^* (b \cup \varepsilon)$.

(iii) Floating point numbers :-

$$(\varepsilon \cup + \cup -) D^+ (\varepsilon \cup . D^+) (\varepsilon \cup (E (\varepsilon \cup + \cup -) D^+))$$

where $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

Writing FSM is easy :-

buildkeywordFSM (K: set of keywords)

1. create a start state q_0
2. for each element k of K do:
   create a branch corresponding to k.
3. create set of transitions that describes the steps when branch dies.
4. Make the states at the ends of each branch accepting.

A Special Case of Pattern Matching

Suppose that we want to match a pattern that is composed of a set of keywords. Then we can write a regular expression of the form:

$$(\Sigma^* (k_1 \cup k_2 \cup \ldots \cup k_n) \Sigma^*)^+$$

For example, suppose we want to match:

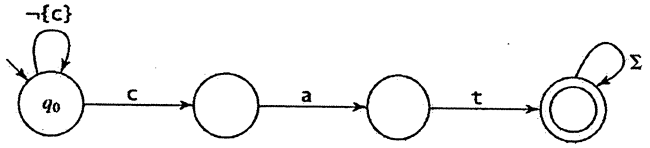
Σ^* finite state machine $\cup$

FSM $\cup$ finite state automatonΣ^*

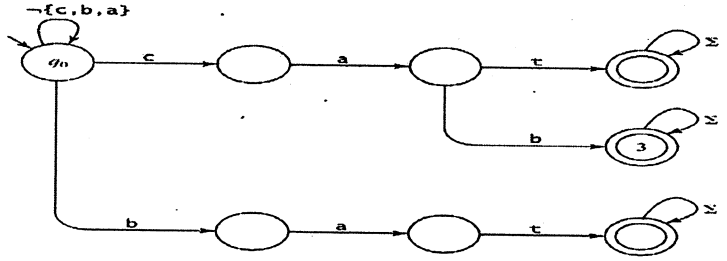
We can use *regextofsm* to build an FSM. But …We can instead use *buildkeywordFSM.*

Example: {cat, bat, cab}

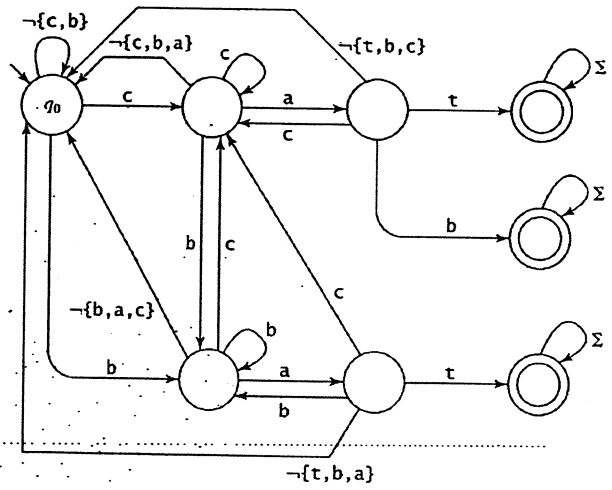
The single keyword cat:



Adding **bat**



Adding **cab**



$\alpha ? = (\alpha \cup \varepsilon)$ i.e α is optional.

α => must occur once.

α^* => may or may not occur may occur many times

α^+ = occur minimum once.

Applications of Regular Expression:

1. **Matching decimal numbers:**
   -? ([0-9]+(\.[0-9]*)? | \.[0-9]+)
2. **Matching ip addresses:**
   ([0-9]{1,3} (\. [0-9] {1,3}){3})
3. **Determining string is a legal password where**
   - **Password must begin with a letter**
   - **Password contain only letter,number, and underscore character**
   - **Password must contain atleast 4 character and no more than 8 character**

   ((A- Z) ∪ (a-z))
   ((A- Z) ∪ (a-z) ∪(0-9) ∪_)
   ((A- Z) ∪ (a-z) ∪(0-9) ∪_)
   ((A- Z) ∪ (a-z) ∪(0-9) ∪_)
   ((A- Z) ∪ (a-z) ∪(0-9) ∪_ ∪ ε)
   ((A- Z) ∪ (a-z) ∪(0-9) ∪_ ∪ ε)
   ((A- Z) ∪ (a-z) ∪(0-9) ∪_ ∪ ε)
   ((A- Z) ∪ (a-z) ∪(0-9) ∪_ ∪ ε)

$\alpha\{n,m\}$ = pattern α must occur atleast n times and no more than m times.

$\alpha\{n\}$ = pattern α must occur exactly 'n' times.

Alternatively: $((a-z) \cup (A-Z))((a-z) \cup (A-Z) \cup (0-9) \cup \_)\{3,7\}$

4. **Trawl for email addresses:**
   \b[A-Za-z0-9_%-]+@[A-Za-z0-9_%-]+ (\.[A-Za-z]+){1,4}\b

;

Simplifying Regular Expressions: Regex's describe sets:

Union:

- Union is commutative: $\alpha \cup \beta = \beta \cup \alpha$.
- Union is associative: $(\alpha \cup \beta) \cup \gamma = \alpha \cup (\beta \cup \gamma)$.
- $\emptyset$ is the identity for union: $\alpha \cup \emptyset = \emptyset \cup \alpha = \alpha$.
- Union is idempotent: $\alpha \cup \alpha = \alpha$.

Concatenation:

- Concatenation is associative: $(\alpha\beta)\gamma = \alpha(\beta\gamma)$.
- ε is the identity for concatenation: $\alpha\varepsilon = \varepsilon\alpha = \alpha$.
- $\emptyset$ is a zero for concatenation: $\alpha\emptyset = \emptyset\alpha = \emptyset$.

Concatenation distributes over union:

- $(\alpha \cup \beta)\gamma = (\alpha\gamma) \cup (\beta\gamma)$.
- $\gamma(\alpha \cup \beta) = (\gamma\alpha) \cup (\gamma\beta)$.

Kleene star:

- $\emptyset^* = \varepsilon$.
- $\varepsilon^* = \varepsilon$.
- $(\alpha^*)^* = \alpha^*$.
- $\alpha^*\alpha^* = \alpha^*$.
- $(\alpha \cup \beta)^* = (\alpha^*\beta^*)^*$.

Regular Grammars

A regular grammar G is a quadruple (V, Σ, R, S), where:

- V is the rule alphabet, which contains nonterminals and terminals,
- Σ (the set of terminals) is a subset of V,
- R (the set of rules) is a finite set of rules of the form:

  $X \rightarrow Y$,
- S (the start symbol) is a nonterminal.

In a regular grammar, all rules in R must:

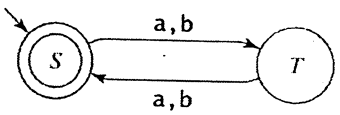
- have a left hand side that is a single nonterminal
- have a right hand side that is:
  - ε, or
  - a single terminal, or
  - a single terminal followed by a single nonterminal.

Legal: $S \rightarrow$ a, $S \rightarrow \varepsilon$, and $T \rightarrow aS$

Not legal: $S \rightarrow aSa$ and $aSa \rightarrow T$

Example:

$L = \{w \in \{a, b\}^* : |w| \text{ is even}\}$ $((aa) \cup (ab) \cup (ba) \cup (bb))^*$



$S \rightarrow \varepsilon$

$S \rightarrow aT$

$S \rightarrow bT$

$T \rightarrow a$

$T \rightarrow b$

$T \rightarrow aS$

$T \rightarrow bS$

Regular Languages and Regular Grammars

Theorem: The class of languages that can be defined with regular grammars is exactly the regular languages.

Proof: By two constructions.

Regular grammar → FSM:

grammartofsm$(G = (V, \Sigma, R, S)) =$

1. Create in M a separate state for each nonterminal in V.
2. Start state is the state corresponding to S.
3. If there are any rules in R of the form $X \rightarrow w$, for some $w \in \Sigma$, create a new state labeled #.
4. For each rule of the form $X \rightarrow w\ Y$, add a transition from X to Y labeled w.
5. For each rule of the form $X \rightarrow w$, add a transition from X to # labeled w.
6. For each rule of the form $X \rightarrow \varepsilon$, mark state X as accepting.
7. Mark state # as accepting.

FSM → Regular grammar: Similarly.

Example 1 - Even Length Strings

$S \rightarrow \varepsilon$ $T \rightarrow a$

$S \rightarrow aT$ $T \rightarrow b$

$S \rightarrow bT$ $T \rightarrow aS$

$T \rightarrow bS$

Strings that End with aaaa

$L = \{w \in \{a, b\}^* : w$ ends with the pattern aaaa$\}$.

$S \rightarrow aS$

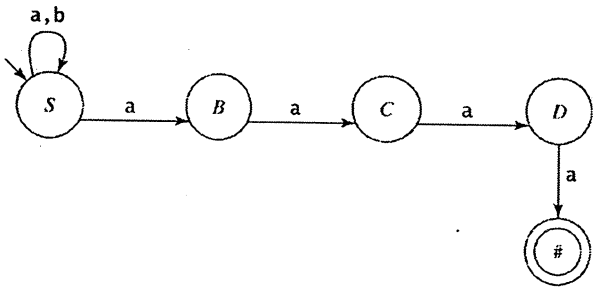
$S \rightarrow bS$

$S \rightarrow aB$

$B \rightarrow aC$

$C \rightarrow aD$

$D \rightarrow a$



Example 2 – One Character Missing

$S \rightarrow \varepsilon$ $A \rightarrow bA$ $C \rightarrow aC$

$S \rightarrow aB$ $A \rightarrow cA$ $C \rightarrow bC$

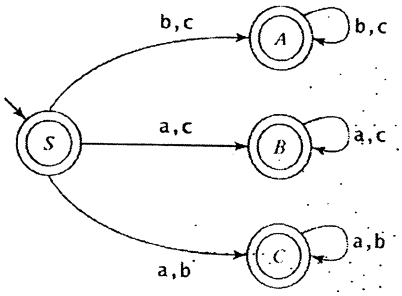
$S \rightarrow aC$ $A \rightarrow \varepsilon$ $C \rightarrow \varepsilon$

$S \rightarrow bA$ $B \rightarrow aB$

$S \rightarrow bC$ $B \rightarrow cB$

$S \rightarrow cA$ $B \rightarrow \varepsilon$

$S \rightarrow cB$



Show a regular grammar for each of the following languages:

a) $\{w \in \{a, b\}^* : w$ contains an odd number of a's and an odd number of b's$\}$.

$G = (\{EE, EO, OE, OO, a, b\}, \{a, b\}, EE, R)$, where $R =$

$EE \rightarrow a\ OE$
$EE \rightarrow b\ EO$
$OE \rightarrow b\ OO$
$OE \rightarrow a\ EE$
$EO \rightarrow \varepsilon$
$EO \rightarrow a\ OO$
$EO \rightarrow b\ EE$
$OO \rightarrow a\ EO$
$OO \rightarrow b\ OE$

b) $\{w \in \{a, b\}^* : w$ does not end in aa$\}$.

$S \rightarrow aA \mid bB \mid \varepsilon$
$A \rightarrow aC \mid bB \mid \varepsilon$
$B \rightarrow aA \mid bB \mid \varepsilon$
$C \rightarrow aC \mid bB$

$\{w \in \{a, b\}^* : w$ does not contain the substring aabb$\}$.

$S \rightarrow aA$ $A \rightarrow aB$ $B \rightarrow aB$ $C \rightarrow aA$
$S \rightarrow bS$ $A \rightarrow bS$ $B \rightarrow bC$ $C \rightarrow \varepsilon$
$S \rightarrow \varepsilon$ $A \rightarrow \varepsilon$ $B \rightarrow \varepsilon$

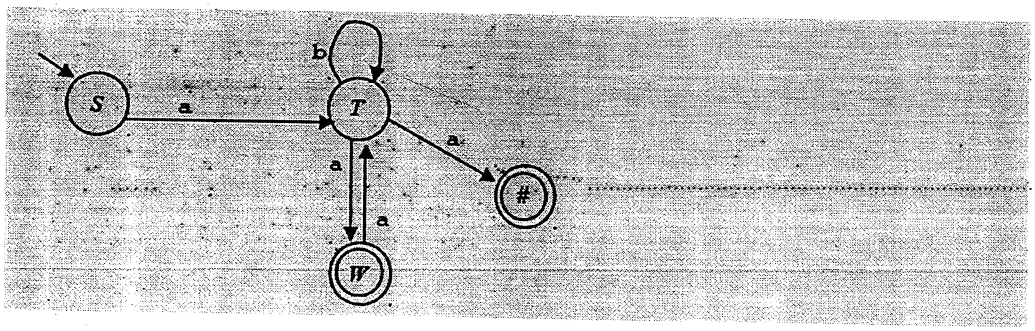
Consider the following regular grammar G:

$S \rightarrow aT$
$T \rightarrow bT$
$T \rightarrow a$
$T \rightarrow aW$
$W \rightarrow \varepsilon$
$W \rightarrow aT$

a) Write a regular expression that generates $L(G)$.

$a\,(b \cup aa)\,a$

b) Use *grammartofsm* to generate an FSM M that accepts $L(G)$.



Let $L = \{w \in \{a, b\}^* : \text{every } a \text{ in } w \text{ is immediately followed by at least one } b\}$.

a) Write a regular expression that describes L.

$(ab \cup b)^*$

b) Write a regular grammar that generates L.

$S \rightarrow bS$
$S \rightarrow aT$
$S \rightarrow \epsilon$
$T \rightarrow bS$

Regular and Nonregular Languages

Languages: Regular or Not?

{$w \in$ {a, b}* : every a is immediately followed by b} is regular. (a*b* is regular.)

{$w \in$ {a, b}* : every a has a matching b somewhere} is not. ({$a^n b^n$: $n \geq 0$} is not.)

- Showing that a language is regular.
- Showing that a language is not regular

How Many Regular Languages?

Theorem: There is a countably infinite number of regular languages.

Proof:

- Upper bound on number of regular languages:
  number of FSMs (or regular expressions).
- Lower bound on number of regular languages:

{a},{aa},{aaa},{aaaa},{aaaaa},{aaaaaa},... are all regular. That set is countably infinite.

There is a countably infinite number of regular languages.

There is an uncountably infinite number of languages over any nonempty alphabet Σ.

So there are *many* more nonregular languages than there are regular ones.

Showing that a Language is Regular

Theorem: Every finite language is regular.

Proof: If L is the empty set, then it is defined by the regular expression ∅ and so is regular. If it is any finite language composed of the strings s_1, s_2, ...s_n for some positive integer n, then it is defined by the regular expression:

$s_1 \cup s_2 \cup \ldots \cup s_n$

So it too is regular.

Example:

Let $L = L_1 \cap L_2$, where:

$L_1 = \{a^n b^n, n \geq 0\}$, and

$L_2 = \{b^n a^n, n \geq 0\}$

L= {Є}, which is finite.

- L_1 = {$w \in$ {0 - 9}*: w is the social security number of the current US president}.
  L is regular because it is finite
- L_2 = {1 if Santa Claus exists and 0 otherwise}
- L_3 = {1 if God exists and 0 otherwise}
  L 2 and L3 are perhaps a little less clear. So either the simple FSM that accepts { 0} or the simple FSM that accepts { 1} and nothing else accepts L_2 and L_3.
- L_4 = {1 if there were people in North America before 10,000 BP and 0 otherwise}
- L_5 = {1 if there were people in North America before 15,000 BP and 0 otherwise}
  L4 is clear. It is the set { 1}. L_5 is also finite and thus regular.

• $L_6 = \{w \in \{0 - 9\}^+ : w$ is the decimal representation, no leading 0's, of a prime Fermat number$\}$

Prime Fermat Numbers

$\Sigma = \{0 - 9\}$

$L = \{w \in \{0 - 9\}^+ : w$ is the decimal representation, no leading 0's, of a prime Fermat number$\}$

The Fermat numbers are defined by $F_n =$ $+ 1, n \geq 0$

Example elements of L:

$F_0 = 3, F_1 = 5, F_2 = 17, F_3 = 257, F_4 = 65{,}537$

Is L regular?

It is known that all divisors of F_n are of the form:

$k \cdot 2^{n+2} + 1$ for some k.

Finiteness - Theoretical vs. Practical

Any finite language is regular. The size of the language doesn't matter.

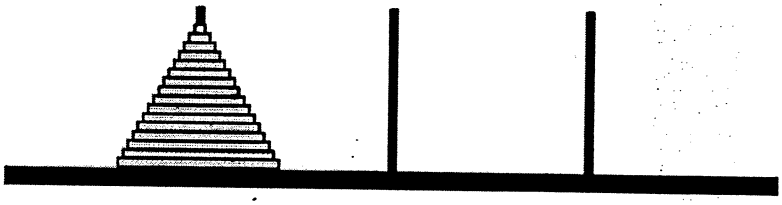
Parity Checking ⟷ Soc. Sec. # Checking

But, from an implementation point of view, it very well may.

When is an FSM a good way to encode the facts about a language?

FSM's are good at looking for repeating patterns. They don't bring much to the table when the language is just a set of unrelated strings.

Regular Does Not Always Mean Tractable



Let $\Sigma = \{12, 13, 21, 23, 31, 32\}$.

Let L be the language of strings that correspond to successful move sequences. The shortest string in L has length $2^{64} - 1$. There is an FSM that accepts L:

Showing That L is Regular

1. Show that L is finite.
2. Exhibit an FSM for L.
3. Exhibit a regular expression for L.
4. Show that the number of equivalence classes of $\approx_L$ is finite.
5. Exhibit a regular grammar for L.
6. Exploit the closure theorems.

Closure Properties of Regular Languages :- This property is a useful tool for building complex automata using the closure property, language recognizers can be built.

- Union
- Concatenation
- Kleene star
- Complement
- Intersection
- Difference
- Reverse
- Letter substitution

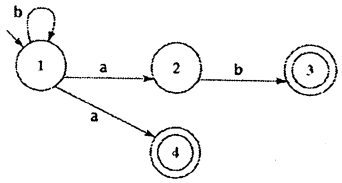
Closure of Regular Languages Under Complement

Theorem: The regular languages are closed under complement.

Proof:

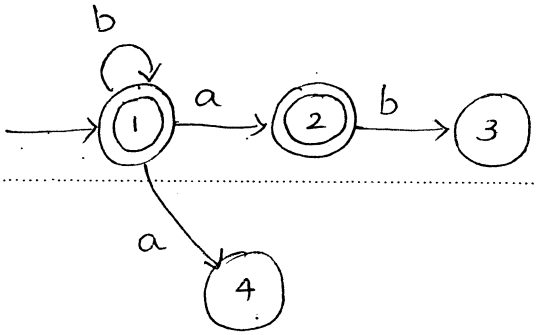
- If L_1 is regular, then there exists a DFSM $M_1=(K,\Sigma,\delta,s,A)$ that accepts it.
- The DFSM $M_2=(K, \Sigma,\delta,s,K-A)$, namely M_1 with accepting and nonaccepting states swapped, accepts $\neg(L(M_1))$ because it rejects all strings that M_1 accepts and rejects all strings that M_1 accepts.

A Complement Example



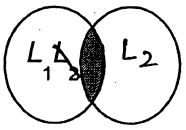
The Complement Procedure

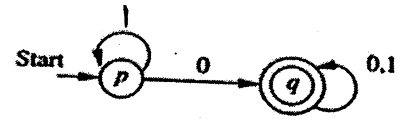
- Given: an FSM *M*,
- Construct a new machine to accept $\neg L(M)$:
- If necessary, use *ndfsmtodfsm* to construct *M'*, a deterministic equivalent of *M*.
- Make sure that *M'* is described completely.
- Swap accepting and nonaccepting states.



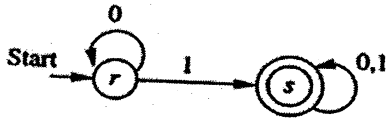
Closure of Regular Languages Under Intersection

$L_1 \cap L_2 = \quad \neg(\neg L_1 \cup \neg L_2)$

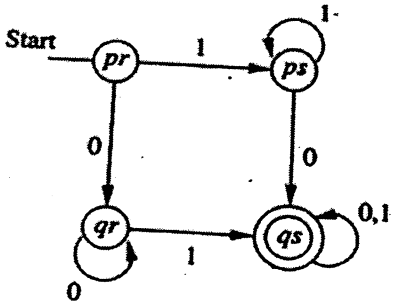




(a)



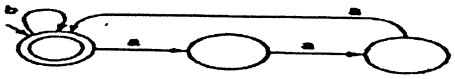
(b)



- Fig (a) is DFSM L1 which accepts strings that have 0.
- Fig(b) is DFSM L2 which accepts strings that have 1.
- Fig(c) is Intersection or product construction which accepts that have both 0 and 1.

Divide-and-Conquer

- Let L = {$w \in$ {a, b}* :w contains an even number of a's and an odd number of b's and all a's come in runs of three}.
- $L = L_1 \cap L_2$, where:
- L_1 = {$w \in$ {a, b}* : w contains an even number of a's and an odd number of b's}, and L_1 is regular already defined.
- L_2 = {$w \in$ {a, b}* : all a's come in runs of three}
  L2 is regular as we have an FSM accepting L2



L is regular because it is the intersection of two regular languages, $L = L_1 \cap L_2$

Closure of Regular Languages Under Difference $L_1 - L_2 = L_1 \cap \neg L_2$

Theorem:

The regular languages are closed under set difference.

Proof:

$L(M_1) - L(M_2) = L(M_1) \cap \neg L(M_2)$

- Regular languages are closed under both complement and intersection is shown.
- Thus regular languages are closed under set difference.

closure under reverse.

S.T If L is regular then L^R is also regular.

proof:- By definition.

→ ϕ is a RE $\therefore \{\phi\}^R = \phi$

→ ε —11— $\{\varepsilon\}^R = \{\varepsilon\}$

→ a —11— $a^R = a$

By inductions:-

$E_1 \cup E_2$ is a RE

$E_1 . E_2$ is a RE

E_1^* is a RE.

If

$E_1 \cup E_2 = E$

then $E^R = E_1^R \cup E_2^R$.

$L(E^R) = L(E_1^R) \cup L(E_2^R)$

If $E = E_1 . E_2$ then

$E^R = E_2^R . E_1^R$.

Closure under Reverse

Theorem:

The regular languages are closed under reverse.

Proof:

- $L^R = \{ w \in \Sigma^* : w = x^R \text{ for some } x \in L \}$.

Example:

1. Let L = {001,10,111} then L^R = {100,01,111}
2. Let L be defined by RE (0 U 1)0* then L^R is 0*(0 U 1)

Closure under letter substitution or Homomorphism

- The regular languages are closed under letter substitution.
- Consider any two alphabets, $\sum_1$ and $\sum 2$.
- Let sub be any function from $\sum_1$ to $\sum_2{}^*$.
- Then **letsub** is a letter substitution function from L_1 to L_2 iff **letsub**(L_1) = $\{ w \in \sum_2{}^* : \exists y \in L_1$ (w = y except that every character c of y has been replaced by sub(c))$\}$.
- Consider $\sum_1 = \{a,b\}$ and $\sum_2 = \{0,1\}$
- Let sub be any function from $\sum_1$ to $\sum_2{}^*$.
- sub(a) = 0, sub(b) = 11
- **letsub**($a^n b^n : n >= 0\}$) = $\{ 0^n 1^{2n} : n >= 0\}$

Letter Substitution

- Let Σ_1 and Σ_2 be alphabets.
- Let *sub* be any function from Σ_1 to $\Sigma_2{}^*$.

Example:

Let: $\Sigma_1 = \{a, b\}$,

$\Sigma_2 = \{0, 1\}$,

sub(a) = 0, and

sub(b) = 11.

letsub is a letter substitution function iff:

$letsub(L_1) = \{w \in \Sigma_2{}^* : \exists y \in L_1$ and

$w = y$ except that:

every character c of y

is replaced by $sub(c)\}$.

Example:

sub(a) = 0, and

sub(b) = 11.

Then *letsub*($\{a^n b^n, n \geq 0\}$) = $\{ 0^n 1^{2n} : n >= 0\}$

Showing that a Language is Not Regular

Every regular language can be accepted by some FSM.

It can only use a finite amount of memory to record essential properties.

Example:

$\{a^n b^n, n \geq 0\}$ is not regular

The only way to generate/accept an infinite language with a finite description is to use:

- Kleene star (in regular expressions), or
- cycles (in automata).

This forces some kind of simple repetitive cycle within the strings.

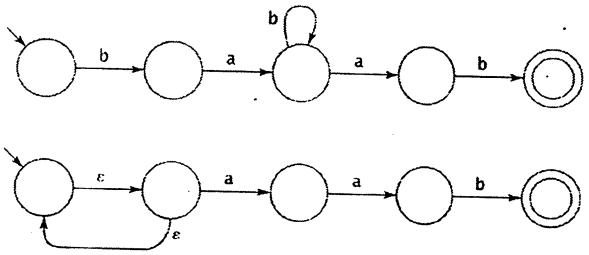
Example:

ab*a generates aba, abba, abbba, abbbba, etc.

Example:

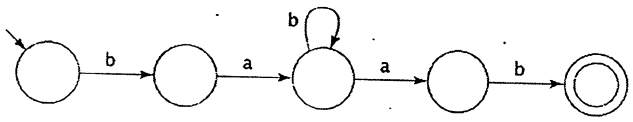
$\{a^n : n \geq 1$ is a prime number$\}$ is not regular.

How Long a String Can Be Accepted?



What is the longest string that a 5-state FSM can accept?

Exploiting the Repetitive Property



If an FSM with n states accepts any string of length $\geq n$, how many strings does it accept?

L = bab*ab

Let w=babbbab

b abbbb a b

xyz

*xy*z* must be in L.

So L includes: baab, babab, babbab, babbbbbbbbbbab

Pumping Theorem [Lemma] for Regular Languages:

Theorem: Let $M = (Q, \Sigma, \delta, q_0, F)$ be an finite automata and has 'n' number of states. Let L be the regular language accepted by M. Let for every string w in L, then there exists a constant n such that $|w| \geq n$.

If L is regular then:

$\exists K \geq 1$

($\forall$ strings $w \in L$, where $|w| \geq K$.

($\exists x, y, z$ ($w = xyz$,

$|xy| \leq K$.

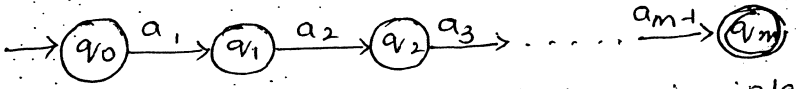
$y \neq \varepsilon$. and.

$\forall q \geq 0$ ($x y^q z$ is in L))))

proof: Let $M = (Q, \Sigma, \delta, q_0, F)$ be an FA and let L is the language accepted by DFA and is regular. Let $w = a_1 a_2 a_3 \ldots a_m$ where $m \geq n$ and each a_i in Σ.

[where n is state of DFA].

Since there are m input symbols, we have $m+1$ states in the sequence $q_0, q_1, q_2, \ldots q_m$ where q_0 is the start state and q_m will be final state.



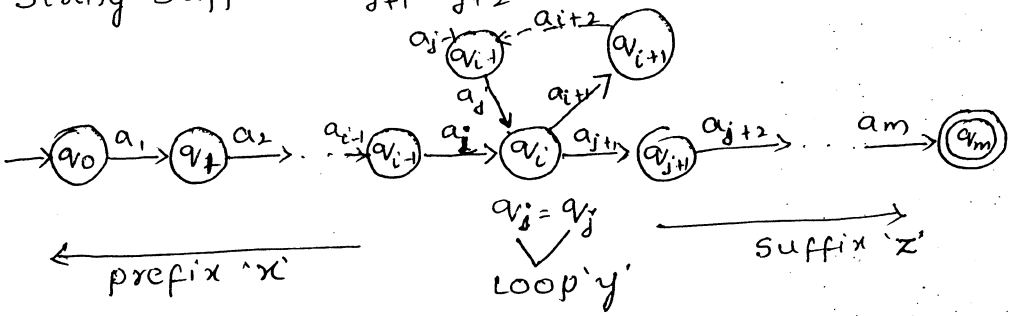
since $|w| \geq n$ by pigeon hole principle it is not possible to have distinct transitions.

Let the string w is divided into 3 substrings xyz

(i) string prefix $a_1 a_2 a_3 \ldots a_i = x$

(ii) Loop string $a_{i+1} a_{i+2} \ldots a_j = y$

(iii) string suffix $a_{j+1} a_{j+2} \ldots a_m = z$.



from the diagram prefix string x takes the machine from q_0 to q_i, loop string y takes the machine from q_i to q_j and suffix string z takes the machine from q_j to q_m. The minimum string accepted by the above m/c is

x with $i=0$;

xyz with $i=1$

xy^iz for all $i>0$.

This can be expressed as :

$$\delta(q_0, a_1 a_2 \cdots a_{i-1} a_i a_{j+1} a_{j+2} \cdots a_m).$$

$$= \delta(\delta(q_0, a_1 a_2 \cdots a_{i-1} a_i), a_{j+1} a_{j+2} \cdots a_m).$$

$$= \delta(q_i, a_{j+1} a_{j+2} \cdots a_m).$$

$$= \delta(\delta(q_j, a_{j+1} a_{j+2}) \cdots a_m).$$

$$= \delta(q_j, a_{m-1} a_m)$$

$$= \delta(\delta(q_j, a_{m-1}) a_m)$$

$$= \delta(q_k, a_m)$$

$$= q_m.$$

i.e q_m is the final state and hence it is accepted. Hence proved.

Applications of pumping lemma :-

* To check whether given language is regular or not.
* To check whether language accepted by FA is finite or infinite.

1. Prove that $L = \{a^n b^n \mid n \geq 0\}$ is not regular.

Soln :- $w = a^n b^n$

$|w| = 2n \geq n$.

Split $w = xyz$ such that $|xy| \leq n$ & $|y| \geq 1$.

$w = \underbrace{aaaa}_{x}\underbrace{a}_{y}\underbrace{bbbb}_{z}$. for $n = 4$.

i.e $a^{n-1} \cdot a \cdot b^n$.

xy^iz can be represented as.

$a^{n-1} \cdot (a)^i b^n$. $[|x| = n-1 \; |y|]$

consider $i = 0$.

$w = a^{n-1} b^n \notin L$. $[|x| = n-1; |y| = 0; |xy| = n-1 < n]$

for $i = 2$

$w = a^{n-1} \cdot a^2 \cdot b^n$

$= a^{n+1} b^n \notin L$.

Hence the language $L = \{a^n b^n \mid n \geq 0\}$ is not regular.

---

2. Show that $L = \{a^i b^j \mid i > j\}$ is not regular.

Soln :- consider $w = a^{n+1} \cdot b^n$ where $j = n$ & $i = n+1$.

$\therefore i > j$.

$|w| = 2n+1 > n$.

Split $w = xyz$.

$w = a^j a^k ab^n$ where $|x| = j \; |y| = k \; |xy| = j+k \leq n$

$k \geq 1$.

By pumping lemma;

$w = a^j (a^k)^i ab^n \cdot \in L \; \forall i \geq 0$.

for $i = 0$.

$a^j (a^k)^0 \cdot ab^n$

for $i = 0$.

$w = a^j \, a \, b^n$.

$= a^{j+1} . b^n \notin L \quad [\because j+1 < n]$

$\therefore \quad L = \{a^i b^j \mid i > j\}$ is not regular.

---

3. $L = \{a^n b^n \mid n \geq 1\}$ is not regular.

Soln:- similar to Example-1.

4. S.T $L = \{ww^R \mid w \in (0+1)^*\}$ is not regular.

Soln:- consider.

$w = 111000 \Rightarrow w^R = 000111 \quad |w| = 2n \; |w^R| = 2n \; \therefore |ww^R| = 4n$.

$\therefore ww^R = 111000000111$

Split the string into 3 parts.

$xyz = 1^{n-1} \mid 0^n . 0^n \mid 1^n$ where $x = 1^{n-1} \quad |x| = n-1$

$y = 1 \quad |y| = 1$.

$\therefore |xy| = n$.

$\therefore xy^iz = 1^{n-1} (1)^i \, 0^{2n} \, 1^n$.

$\forall i = 0 ; \quad 1^{n-1} . 0^{2n} \, 1^n \notin L$.

$\therefore L = \{ww^R \mid w \in \{0+1\}^*\}$ is not regular.

---

5. S.T $L = \{w \mid n_a(w) = n_b(w)\}$ is not regular.

Soln:- for w where $n_a(w) = n_b(w)$.

$L = \{\varepsilon, ab, aabb, abab, baba, bbaa, \ldots\}$.

$\therefore w = a^n b^n \in L$.

consider $L = \{a^n b^n \mid n \geq 0\}$.

proof is same as example-1.

6. S.T $L = \{a^n b^l c^{n+l} \mid n, l \geq 0\}$ is not regular.

Soln:- L is regular language, closed under homophorism.

$\therefore h(a) = a$; $h(b) = a$; $h(c) = c$.

$\therefore L = \{a^n b^l c^{n+l}\}$ can be represented as:

$L = \{a^n a^l c^{n+l} \mid n+l \geq 0\}$. $\quad n+l = i$

$L = \{a^i c^i \mid i \geq 0\}$

proof is similar to example (1).

7. S.T $L = \{a^{n!} \mid n \geq 0\}$ is not regular.

Soln:- $n! = 1 * 2 * 3 * \ldots n$.

Let $w = a^{n!}$ $\quad \therefore |w| \geq n$.

Split $w = a^{n!} = a^j \cdot a^k \cdot a^{n!-j-k}$.

where $x = a^j$; $y = a^k$; $z = a^{n!-j-k}$

$\therefore |x| = j$; $|y| = k$; $|xy| \neq n!$

$|xy| = j + k \leq n$.

According to pumping lemma.

$xy^i z = a^j (a^k)^i \cdot a^{n!-j-k}$.

for $i = 0$.

$= a^j \cdot a^{n!-j-k} \in L$. [i.e $xz \in L$].

$\Rightarrow$ implies $a^{n!-k} \in L$ (for $j = 0$).

$n! > n!-k$

for $k = 1$ $\quad n! > n!-1$

According to lemma $n! = n!-1$

which is contradiction.

$\therefore$ L is not regular.

7. S.T $L = \{0^n \mid n \text{ is prime}\}$ is not regular.

Soln:- $L = \{00, 000, 00000, \ldots\}$.

Let $w = 0^n \mid n$ is prime; $|w| \geq n$

Split $w = xyz$ such that $|xy| \leq n$ & $|y| \geq 1$.

$w = 0^j\ 0^k\ 0^{n-j-k}$. ; $x = 0^j$; $y = 0^k$; $z = 0^{n-j-k}$.

$|x| = j \quad |y| = k$.

$|xy| = j + k \leq n$.

According to pumping lemma.

$w = xy^iz = 0^j (0^k)^i\ 0^{n-j-k} \in L$. for $\forall i = 0, 1, 2, \ldots$

$= 0^j\ 0^{ki}\ 0^{n-j-k}$.

$= j + ki + n - j - k =$

$= n + k(i-1)$ is prime for all $i \geq 0$.

if $i = n+1$ then.

$= n + k(n+1-1)$

$= n + kn = n(k+1)$. should also be prime.

for $k \geq 1$.

if $k = 1$ $n(1+1) = 2n$ which is not a prime.

Hence $L = \{0^n \mid n$ is prime$\}$ is not regular.

8. S.T $L = \{ww \mid w \in \{a, b\}^*\}$ is not regular

9. S.T $L = \{w \mid n_a(w) < n_b(w)\}$ is not regular.

Apply pumping Lemma for the following languages & prove that they are not regular.

(i) $L = \{ww^R \mid w \in (0+1)^*\}$.

Soln: Let $w = abab$

$w^R = baba$.

String $w = \underbrace{ab}_{n}\underbrace{ab}_{n}\underbrace{ba}_{n}\underbrace{ba}_{n}$ where $|z| \geq n$.

Let $w = xyz$ the string can be split as.

$w = \underbrace{a}_{x}\underbrace{b}_{y}\underbrace{abbaba}_{z}$.

$|x| = n-1$

$|y| = 1$.

$|xy| = n-1+1 = n$.

according to pumping lemma.

$x = \text{varies}$ $w = xy^iz$.

$w = a(b)^i(abbaba)$

for $i = 2$.

$w = a(b)^2(abbaba)$

$= abbabbaba \notin L$.

Hence ww^R is not regular.

(ii) $L = \{b^{i^2} \mid i \geq 1\}$

Solution:-

$L = b^{i^2}$ if $i = 1$ $b^{1^2} = b$.

$i = 2$ $b^{2^2} = bbbb$ length $= i^2$

$|w| = i^2$. if $i = n = n^2$.

$w = xyz$. $|xy| \leq n$ & therefore

$|y| \leq n$.

$w = x y^q z$ $q = 1$.

$w = xyz = n^2$.

$|w| = |x| + |y| + |z| = n^2$.

if $q = 2$.

$|w| = |xy^2z| = |x| + 2|y| + |z|$.

$\therefore$ $n^2 < |xy^2z|$.

$\Rightarrow n^2 < [\underbrace{|x| + |y| + |z|}_{n^2} + \underbrace{|y|}_{n}] \leq n^2 + n$. $[\because |y| \leq n$.

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

if $|xy^2z| < n^2 + n$ it is also less than.

$n^2 + n + n + 2$.

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

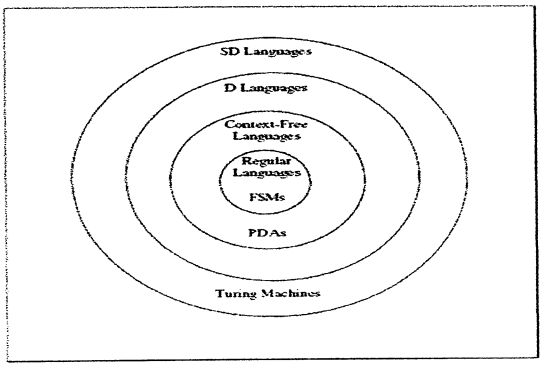
$n^2 \leq |xy^qz| \leq (n+1)^2$

Thus the string lies between 2 consecutive perfect square. But the string is not a perfect square. Hence the given language is not regular.

Module-3

Context-Free Grammars

Languages and Machines



Rewrite Systems and Grammars

A rewrite system (or production system or rule based system) is:

- a list of rules, and
- an algorithm for applying them.

Each rule has a left-hand side and a right hand side.

Example rules:

$S \rightarrow aSb$
$aS \rightarrow \varepsilon$
$aSb \rightarrow bSabSa$

Simple-rewrite

simple-rewrite(*R*: rewrite system, *w*: initial string) =

1. Set *working-string* to *w*.
2. Until told by *R* to halt do:
   - Match the LHS of some rule against some part of *working-string*.
   - Replace the matched part of *working-string* with the RHS of the rule that was matched.
3. Return *working-string*.

- If it returns some string s then R can **derive** s from w or there exists a **derivation** in R of s from w.
- A rewrite system formalism specifies:
  - The form of the rules

- How *simple-rewrite* works:
  - How to choose rules?
  - When to quit?

An Example

$w = SaS$ (Is this ONE string or many?)

Rules: [1] $S \rightarrow aSb$

[2] $aS \rightarrow \varepsilon$

- What order to apply the rules?
- When to quit?

$SaS \Rightarrow aSbaS \Rightarrow \varepsilon baS \Rightarrow b\varepsilon \Rightarrow b$

$SaS \Rightarrow aSbaS \Rightarrow aaSbbaS \Rightarrow a\varepsilon bbaS \Rightarrow abb\varepsilon \Rightarrow abb$

- The answer to the question is "many"?
- How many strings does *w* represent?

Grammars Define Languages

- A grammar is a set of rules that are stated in terms of two alphabets:
- a ***terminal alphabet***, Σ, that contains the symbols that make up the strings in $L(G)$, and
- a ***nonterminal alphabet***, the elements of which will function as working symbols that will be used while the grammar is operating. These symbols will disappear by the time the grammar finishes its job and generates a string.
- A grammar has a unique ***start symbol***, often called ***S***.

Using a Grammar to Derive a String

Simple-rewrite (G, S) will generate the strings in $L(G)$.

We will use the symbol $\Rightarrow$ to indicate steps in a derivation. A derivation could begin with:

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow \ldots$

Generating Many Strings

Multiple rules may match.

Example 1: $S \rightarrow aSb$, $S \rightarrow bSa$, and $S \rightarrow \varepsilon$

Derivation so far: $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow$ Three

choices at the next step:

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb$ (using rule 1),
$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabSabb$ (using rule 2),
$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$ (using rule 3).

One rule may match in more than one way.

Example 2: $S \rightarrow aTTb$, $T \rightarrow bTa$, and $T \rightarrow \varepsilon$

Derivation so far: $S \Rightarrow aTTb \Rightarrow$

Two choices at the next step:
$S \Rightarrow aTTb \Rightarrow abTaTb \Rightarrow S$

$\Rightarrow aTTb \Rightarrow aTbTab \Rightarrow$

When to Stop

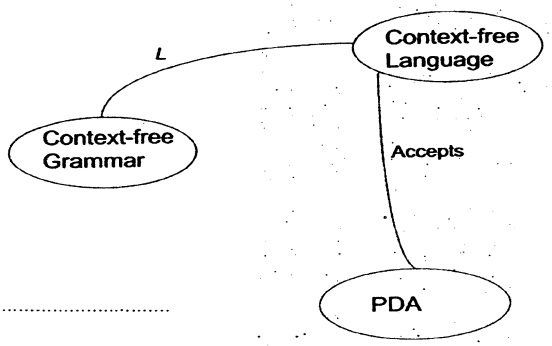
i. May stop when, the working string no longer contains any nonterminal symbols (including, when it is ε). In this case, we say that the working string is *generated* by the grammar.
Example:
$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$

ii. There are nonterminal symbols in the working string but none of them appears on the left-hand side of any rule in the grammar. In this case, we have a blocked or non-terminated derivation but no generated string.
Example:
Rules: $S \rightarrow aSb$, $S \rightarrow bTa$, and $S \rightarrow \varepsilon$
Derivations: $S \Rightarrow aSb \Rightarrow abTab \Rightarrow$ [blocked]

iii. It is possible that neither (1) nor (2) is achieved.
Example:
G contains only the rules $S \rightarrow Ba$ and $B \rightarrow bB$, with S the start symbol.
Then all derivations proceed as:
$S \Rightarrow Ba \Rightarrow bBa \Rightarrow bbBa \Rightarrow bbbBa \Rightarrow bbbbBa \Rightarrow \ldots$

In the last 2 cases, you have a "bad" grammar!!

Context-free Grammars, Languages, and Push down Automata



Recall Regular Grammar

- Have a left-hand side that is a single nonterminal
- Have a right-hand side that is e or a single terminal or a single terminal followed by a single nonterminal
- Regular grammars must always produce strings one character at a time, moving

Concatenating Independent Sublanguages

Let $L = \{a^n b^n c^m : n, m >= 0\}$.

The c^m portion of any string in L is completely independent of the $a^n b^n$ portion, so we should generate the two portions separately and concatenate them together.

$G = (\{S, N, C, a, b, c\}, \{a, b, c\}, R, S\}$ where:

$R = \{ S \rightarrow NC$

$N \rightarrow aNb$

$N \rightarrow \varepsilon$

$C \rightarrow cC$

$C \rightarrow \varepsilon$

}.

Ex 11.7: unequal a's and b's

$L = \{a^n b^m : n \neq m\}$

$G = (V, \Sigma, R, S)$, where

$V = \{a, b, S, A, B\}$,

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

$R = \{$ S → A /* more a's than b's

S → B /* more b's than a's

A → a /* at least one extra a generated

A → aA /* any number of a's

A → aAb /* equal number of a's and b's

B → b /* at least one extra b generated

B → Bb /* any number of b's

B → aBb /* equal number of a's and b's

OR

R = {S → aSb / A / B

A→aA/a

B→bB/b}

Ex 11.8: L ={w | number of a's > number of b's}

S→ AB	[more 'a' at the end]
S → BA	[more 'a' at the beginning]
S→ ABA	[more 'a' at middle]
A→ aAb | bAa| AA | ε	[equal a's and b's]
B→ aB | a	[one or more a's]

7) $\{ a^i b^j c^k : i, j, k \geq 0$ **and** $(k \leq i$ **or** $k \leq j)\}$.

$S \rightarrow A \mid B$

$A \rightarrow aAc \mid aA \mid M$

$B \rightarrow aB \mid F$

$F \rightarrow bFc \mid bF \mid \varepsilon$

$M \rightarrow bM \mid \varepsilon$

8) $\{w \in \{a, b\}^*$ **: every prefix of** w **has at least as many a's as b's**$\}$.

$S \rightarrow aS \mid aSb \mid SS \mid \varepsilon$

9) $\{a^n b^m : m \geq n$, $m-n$ **is even**$\}$.

$S \rightarrow aSb \mid S \rightarrow Sbb \mid \varepsilon$

Derivation and Parse Trees

- **Parse trees** capture the essential grammatical structure of a string.
- A program that produces such trees is called a **parser**.
- A **parse tree** is an (ordered, rooted) tree that represents the syntactic structure of a string according to some formal grammar.
- In a parse tree, the interior nodes are labeled by nonterminals of the grammar, while the leaf nodes are labeled by terminals of the grammar or ε.

Parse Trees

A parse tree, derived by a grammar $G = (V, S, R, S)$, is a rooted, ordered tree in which:

1. Every leaf node is labeled with an element of S È {e},
2. The root node is labeled *S*,
3. Every other node is labeled with some element of:
   V –S, and
4. If *m* is a nonleaf node labeled *X* and the children of *m* are labeled $x_1, x_2, \ldots, x_n$, then *R* contains the rule
   $X \rightarrow x_1, x_2, \ldots, x_n$

- A parse tree may correspond to multiple derivations.
- Parse trees are useful precisely because they capture the important structural facts about a derivation but throw away the details of the order in which the nonterminals were expanded.
- The order has no bearing on the structure we wish to assign to a string

English Grammar

S → *NP VP*
NP → the *Nominal* | a *Nominal* | *Nominal* | *ProperNoun* | *NP PP*
Nominal → *N* | *Adjs N*
N → cat | dogs | bear | girl | chocolate | rifle
ProperNoun → Chris | Fluffy
Adjs → *Adj Adjs* | *Adj*
Adj → young | older | smart
VP → *V* | *V NP* | *VP PP*
V → like | likes | thinks | shots | smells
PP → *Prep NP*
Prep → with

NOTE : Please refer class notes for problems.

Ambiguity

- Sometimes a garmmar may produce more than one parse tree for some or all of the strings it generates.
- When this happens we say that the grammar is ambiguous.
- A grammar is ***ambiguous*** iff there is at least one string in $L(G)$ for which G produces more than one parse tree.

Even a very simple grammar can be highly ambiguous

Bal={$w \in$ {),(}*: the parenthesis are balanced}.

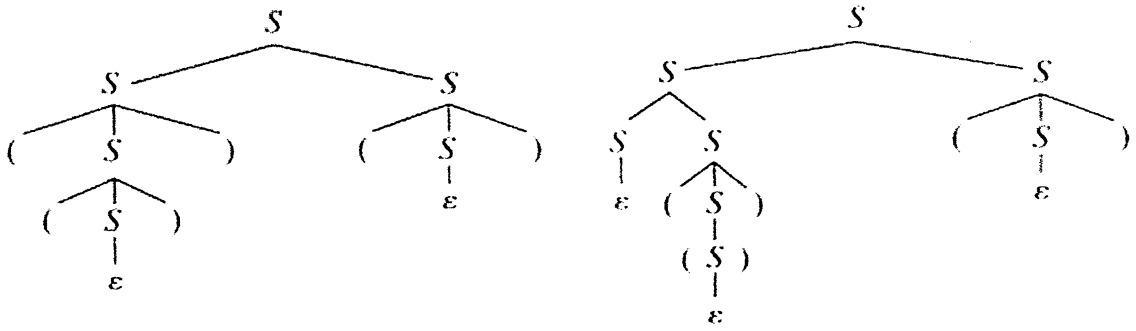
G={{S,),(}, {),(},R,S} where R={ S→ε, $S \rightarrow SS$, $S \rightarrow (S)$ }

Two leftmost derivations for the string (())()

1. $S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (( ))S \Rightarrow (())(S) \Rightarrow (( ))( )$

2. $S \Rightarrow SS \Rightarrow SSS \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (( ))S \Rightarrow (())(S) \Rightarrow (( ))( )$

Parse trees for the two leftmost derivations are



Since both the parse trees obtained for the same string (( ))( ) are different, the grammar is ambiguous.

- A grammar is unambiguous iff for all strings w, at every point in a LHS or RHS derivation of w, only one rule in G can be applied.

Techniques for reducing ambiguity

Three grammar structures that often lead to ambiguity:

1. *s.* rules like S⊚ ε $S \rightarrow \varepsilon$
2. Rules like S⊚SS or E⊚E + E. In other words recursive rules whose right- hand sides are symmetric and contain at least two copies of the nonterminal on the left-hand side.
3. Rule sets that lead to ambiguous attachment of optional postfixes.

1. Eliminating ε-rules:

Nullable variable: A variable X is *nullable* iff either:

(1) there is a rule $X \rightarrow \varepsilon$, or

(2) there is a rule $X \rightarrow PQR\ldots$ and P, Q, R, ... are all nullable.

So compute N, the set of nullable variables, as follows:

4. Set N to the set of variables that satisfy (1).
5. Until an entire pass is made without adding anything to N do

   Evaluate all other variables with respect to (2).

   If any variable satisfies (2) and is not in N, insert it.

Modifiable variable: a rule is *modifiable* iff it is of the form:

$P \rightarrow \alpha Q \beta$, for some nullable Q.

What If ε ∈ L?

Sometimes L(G) contains ε and it is important to retain it. To handle this case, we present the following algorithm which constructs a new grammar G", such that L (G") = L (G). If L(G) contains ε, then G" will contain a single ε-rule that can be thought of as being "quarantined".Its sole job is to generate the string ε. It can have no interaction with the other rules of the grammar.

atmostoneEps(G: cfg) =

1. ***G' = removeEps(G).***
2. ***If S_G is nullable then*** ***/* i. e., ε ∈ L(G)***
   - ***2.1 Create in G' a new start symbol S*.***
   - ***2.2 Add to $R_{G'}$ the two rules:***

     $S^* \rightarrow \varepsilon$

     $S^* \rightarrow S_G$
3. **Return G'.**

Ex: Remove ε- rule from balanced parentheses grammar $S \rightarrow SS \mid (S) \mid \varepsilon$

$S^* \rightarrow \varepsilon$

$S^* \rightarrow S$

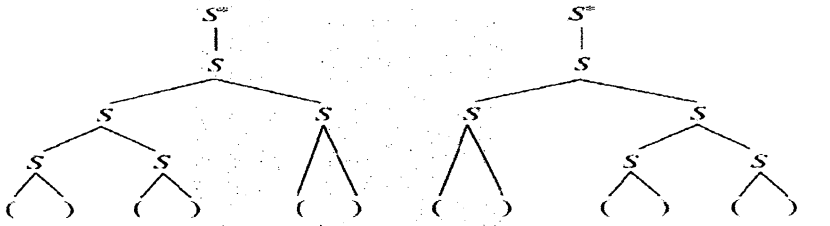
$S \rightarrow SS$

$S \rightarrow (S)$

$S \rightarrow ()$

2. Eliminating symmetric rules:

The new grammar that we just built for Bal is better than our original one. But it is still ambiguous. The string()() has two parses shown in Figure. The problem now is the rule $S \rightarrow SS$, which must be applied n - 1 times to generate a sequence of n balanced parentheses substrings. But, at each time after the first, there is a choice of which existing S to split.



The solution to this problem is to rewrite the grammar so that there is no longer a choice. We replace the rule S--+ SS with one of the following rules:

$S \rightarrow SS_1$ /* *force branching to the left*

$S \rightarrow S_1S$ /* *force branching to the right*

Then we add the rules© S1 and replace the rules $S \rightarrow$ *(S)* and $S \rightarrow$ ( )with the rules $S1 \rightarrow$ (S) and S→ ( ).What we have done is to change the grammar so that branching can occur only in one direction. Every S that is generated can branch, but no S1 can. When all the branching has happened, S rewrites to S1 and the rest of the derivation can occur.

So one unambiguous grammar for Bals G = { {S,), (}, {), (}, R, S), where:

$S^* \rightarrow \varepsilon$ $S \rightarrow SS_1$

$S^* \rightarrow S$ $S \rightarrow S_1$

$S_1 \rightarrow$

(S)

$S_1 \rightarrow$ ( )

There exists single parse tree for the string ( )( )( )

3. Ambiguous Attachment

Third source of ambiguity that we will consider arises when constructs with optional fragments are nested. Probably the most often described instance of this kind of ambiguity is known as the *dangling else problem.* Suppose that we define a programming language with an if statement that can have either of the following forms:

<stmt> ::= if <cond> then <stmt>

<stmt> ::= if <cond> then <stmt> else <stmt>

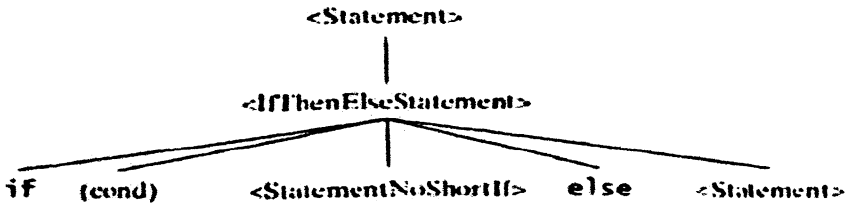
In other words. the else clause is optional. Then the following statement with just a single else clause has two parses:

if *cond1* then if *cond2* then *stmt1* else *smt2*

In the first parse, the single else clause goes with the first if. (So it attaches high in the parse tree.) In the second parse, the single else clause goes with the second if. (In this case it attaches lower in the parse tree.)

Ex: dangling else problem in Java:

<Statement> ::= <IfThenStatement> | <IfThenElseStatement> |
<IfThenElseStatementNoShortIf>
<StatementNoShortIf> ::= <block> |
<IfThenElseStatementNoShortIf> | …
<IfThenStatement> ::= if (<Expression>) <Statement>
<IfThenElseStatement> ::= if (<Expression>)
<StatementNoShortIf> else <Statement>
<IfThenElseStatementNoShortIf> ::=
if (<Expression>) <StatementNoShortIf>
else <StatementNoShortIf>



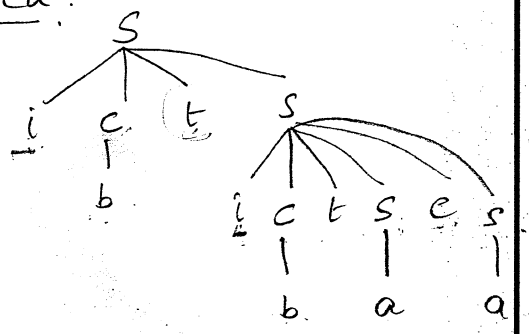
Eliminating Ambiguity:

* Dis ambiguity Rule: consider the grammar:

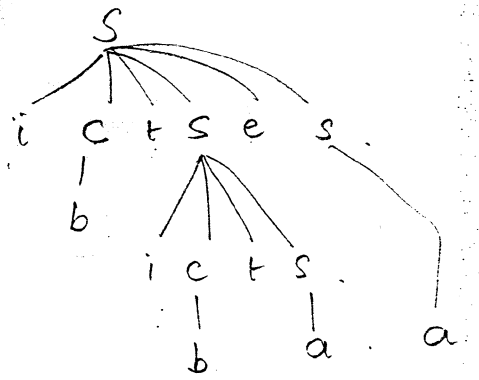
$S \rightarrow iCtS \mid iCtSeSla$

$C \rightarrow b$.

To derive the string: ibtibtaea.



$S \Rightarrow iCtS$
$\Rightarrow ibtS$
$\Rightarrow ibtiCtSeS$
$\Rightarrow ibtibtSeS$
$\Rightarrow ibtibtaeS$.
$\Rightarrow ibtibtaea$.



$S \Rightarrow iCtSeS$
$\Rightarrow ibtSeS$
$\Rightarrow ibtiCtSeS$
$\Rightarrow ibtibtSeS$.
$\Rightarrow ibtibtaeS$
$\Rightarrow ibtibtaea$.

Since 2 different parse trees exists for the same string given grammar is ambiguous.

* 1st parse tree associates else with 2nd if statement
* 2nd —11— associates else with 1st if statement.

This ambiguity is called dangling else problem.

To eliminate this "Match each else with the closest unmatched then":

$\therefore M \rightarrow iCtMeM$

unmatched statement.

$U \rightarrow iCtS$

∴ If-else statement where the statement before else is matched and statement after else is unmatched

$U \rightarrow iCtMeU$

∴ final grammar.

$S \rightarrow M \mid U \mid a.$

$M \rightarrow iCtMeM.$

$U \rightarrow iCtS.$

$U \rightarrow iCtMeU.$

$C \rightarrow b.$

* Eliminating Ambiguity using precedence & Associativity

Eg: $E \rightarrow E*E \mid E-E \mid E\hat{}E \mid E/E \mid E+E \mid E \rightarrow (E) \mid id.$

Arrange the operators in increasing order of precedence

+, -	left	E
*, /	left	T.
^	Right	P.

$a \wedge p \Rightarrow a^p$

Beginning from the basic units ( ) & id.

$F \rightarrow (E) \mid id.$

Next

$P \rightarrow F\hat{}P \mid E.$

Next

$T \rightarrow T*P \mid T/P \mid P.$

Next

$E \rightarrow E+T \mid E-T \mid T.$

∴

$E \rightarrow E+T \mid E-T \mid T$

$T \rightarrow T*P \mid T/P \mid P$

$P \rightarrow F\hat{}P \mid E.$

$F \rightarrow (E) \mid d.$

Simplifying Context-Free Grammars

Eliminate

1. Useless symbols
   - Two algorithms used to eliminate useless variables
     - Removeunproductive (G:CFG)
     - Removeunreachable (G:CFG)
2. ε-productions
3. Unit productions

1. Useless symbols

Unproductive Nonterminals

removeunproductive(*G*: CFG) =

1. $G' = G$.
2. Mark every nonterminal symbol in G' as unproductive.
3. Mark every terminal symbol in G' as productive.
4. Until one entire pass has been made without any new symbol being marked do:
   a. For each rule $X \rightarrow \alpha$ in R do:
      i. If every symbol in α has been marked as productive and X has not yet been marked as productive then:
      ii. Mark X as productive.
5. Remove from G' every unproductive symbol.
6. Remove from G' every rule that contains an unproductive symbol.
7. Return G'.

Example

$G = (\{S, A, B, C, D, a, b\}, \{a, b\}, R, S)$, where

$R = \{$

$S \rightarrow AB \mid AC$	1) a and b terminal symbols are productive
$A \rightarrow aAb \mid \varepsilon$	2) A is productive(because $A \rightarrow aAb$)
$B \rightarrow bA$	3) B is productive(because $B \rightarrow bA$)
$C \rightarrow bCa$	4) S & D are productive(because $S \rightarrow AB$ & $D \rightarrow AB$)
$D \rightarrow AB$	5) C is unproductive

$\}$

On eliminating C from both LHS and RHS the rule set R' is

$R' = \{ \boldsymbol{S \rightarrow AB}$

$\boldsymbol{A \rightarrow aAb \mid \varepsilon}$

$\boldsymbol{B \rightarrow bA}$

$\boldsymbol{D \rightarrow AB}$

$\}$

Unreachable Nonterminals

removeunreachable(*G*: CFG) =

1. $G' = G$.
2. Mark S as reachable.
3. Mark every other nonterminal symbol as unreachable.
4. Until one entire pass has been made without any new symbol being marked do:
   For each rule $X \rightarrow \alpha A \beta$ (where $A \in V - \Sigma$) in R do:
   If X has been marked as reachable and A has not then: Mark A as reachable.
5. Remove from G' every unreachable symbol.
6. Remove from G' every rule with an unreachable symbol on the left-hand side.

Return G'.

Example

$G = (\{S, A, B, C, D, a, b\}, \{a, b\}, R, S)$, where

$R' = \{S \rightarrow AB$

$A \rightarrow aAb \mid \varepsilon$

$B \rightarrow bA$

$D \rightarrow AB\ \}$

S, A, B are reachable but D is not reachable, on eliminating D from both LHS and RHS the rule set R'' is

$R'' = \{ \boldsymbol{S \rightarrow AB}$

$\boldsymbol{A \rightarrow aAb \mid \varepsilon}$

$\boldsymbol{B \rightarrow aA}\ \}$

2. *A General Technique for Getting Rid of ε-Rules:*

removeEps(G: CFG) =

1. Let G′ = G.
2. Find the set N of nullable variables in G′.
3. Repeat until G′ contains no modifiable rules that haven't been processed:
   Given the rule $P \rightarrow \alpha Q \beta$, where $Q \in N$, add the rule $P \rightarrow \alpha\beta$, if it is not already present and if $\alpha\beta \neq \varepsilon$ and if $P \neq \alpha\beta$.
4. Delete from G′ all rules of the form $X \rightarrow \varepsilon$.
5. Return G′.

Therefore, L(G′) = L(G) – {ε}.

Example: S→ aTa

$T \rightarrow ABC$

A→ aA|C

$B \rightarrow Bb \mid C$

C→ c | ε}.

On input *G*, *removeEps* behaves as follows:

S→ aa /* Since T is nullable.

$T \rightarrow BC$ /* Since A is nullable.

$T \rightarrow AC$ /* Since D is nullable.

$T \rightarrow AB$ /* Since C is nullable.

$T \rightarrow$ C /* From $T \rightarrow BC$, since B is nullable. Or from $T \rightarrow AC$.

$T \rightarrow B$ /* From $T \rightarrow BC$. since C is nullable. Or from $T \rightarrow AB$.

$T \rightarrow A$ /* FromT$\rightarrow AC$. since C is nullable. Or from $T \rightarrow AB$

$A \rightarrow$ a /* Since A is nullable.

$B \rightarrow$ b /* Since D is nullable.

C$\rightarrow$ c

Finally, step 4 deletes the rule C$\rightarrow \varepsilon$

3. *Removal of unit productions*

A ***unit production*** is a rule whose right-hand side consists of a single nonterminal symbol. The job of *remove Units* is to remove all unit productions and to replace them by a set of other rules that accomplish the job previously done by the unit productions. So, for example, suppose that we start with a grammar G that contains the following rules:

$S \rightarrow XY$

$X \rightarrow A$

$A \rightarrow B \mid a$

$B \rightarrow$ b

Once we get rid of unit productions, it will no longer be possible for X to become A (and then B) and thus to go on to generate a or b. So X will need the ability to go directly to a and b, without any intermediate steps. We can define *removeUnits( )* as follows:

***removeUnits(G:* CFG) =**

1. Let G' =G.

2. Until no unit productions remaining G' do:

2.1. Choose some unit production $X \rightarrow Y$.

2.2. Remove it from G'.

2.3. Consider only rules that still remain in G'. For every rule $Y \rightarrow \beta$ where $\beta \in V^*$, do:

Add to G' the rule $X \rightarrow \beta$ unless that is a rule that has already been removed once.

3. Return G'.

EXAMPLE : Removing Unit Productions:

S$\rightarrow$ XY

$X \rightarrow A$

$A \rightarrow B \mid$ a

B$\rightarrow$b

$Y \rightarrow T$

$T \rightarrow Y \mid$ c

Normal forms

Let *C* be any set of data objects. For example, *C* might be the set of context-free grammars. Or it could be the set of syntactically valid logical expressions or a set of database queries. We will say that a set *F* is a *normal form* for C iff it possesses the following two properties:

- For every element *c* of C, except possibly a finite set of special cases, there exists some element *f* of *F*

such that *f* is equivalent to *c* with respect to some set of tasks.

- *F* is simpler than the original form in which the elements of *C* are written. By "simpler" we mean that at least some tasks are easier to perform on elements *of F* than they would be on elements of C.

Normal Forms For Grammars:

We will define the following two useful normal forms for context-free grammars:

- ***Chomsky Normal Form****:* In a Chomsky normal form grammar $G = (V, \sum R, S)$,vall rules have one of the following two forms:
  - $X \rightarrow a$, where a $\in \sum$, or
  - $X \rightarrow BC$, where *B* and Care elements of $V - \sum$
- ***Greibach Normal form****:* In a Greibach normal form grammar $G = (V, \sum R, S)$, all rules have the following form:
  - $X \rightarrow \alpha\beta$(where $\alpha \in \sum$ and $\beta \in (V - \sum)^*$

Converting to a normal form:

Algorithms to convert grammars into normal forms generally begin with n grammar G and then operate in a series of steps as follows:

1. Apply some transformation to G to get rid of undesirable property 1. Show that the language generated by *G* is unchanged.
2. Apply another transformation to *G* get rid of undesirable property 2. Show that the language generated by G is unchanged *and* that undesirable property 1 has not been reintroduced.
3. Continue until the grammar is in the desired form.

Converting to Chomsky Normal Form

There exists a straightforward four-step algorithm that converts a grammar $G = (V, \sum, R, S)$ into a new grammar *Gc* such that *Gc* is in Chomsky normal form and $L(Gc) = L(G) - \{\varepsilon\}$. Define:

***converttotoChomsky(* G: CFG) =**

1. *Let Gc be the result of removing from G all ε-rules, using the algorithm removeEps( )*
2. Let *Gc* be the result of removing from *Gc* all unit productions (rules of the form $A \rightarrow B$) using the algorithm *removeUnits* defined below. It is important to remove ε-rules first, before applying *removeUnits( )*. Once this step has been completed, all rules whose right- hand sides have length 1 are in Chomsky normal form (i.e . they are composed of a single terminal symbol).
3. Let *Gc* be the result of removing from *Gc* all rules whose right-hand sides have length greater

than 1 and include a terminal (e.g : $A \rightarrow aB$ or $A \rightarrow$ BaC). This step is simple and can be performed by the algorithm *remove Mixed* given below. Once this step has been completed all rules whose right-hand sides have length 1 or 2 are in Chomsky normal form.

4. Let *Gc* be the result of removing from *Gc* all rules whose right-hand sides have length greater than 2 (e.g : $A \rightarrow BCD£$). This step too is simple. It can be performed by the algorithm *remove Long* given below.
5. Return *Gc*.

A ***unit production*** is a rule whose right-hand side consists of a single nonterminal symbol. The job of *remove Units* is to remove all unit productions and to replace them by a set of other rules that accomplish the job previously done by the unit productions. So, for example, suppose that we start with a grammar *G* that contains the following rules:

$S \rightarrow XY$

$X \rightarrow A$

$A \rightarrow B \mid a$

$B \rightarrow$ b

Once we get rid of unit productions, it will no longer be possible for X to become A (and then B) and thus to go on to generate a or b. So X will need the ability to go directly to a and b, without any intermediate steps. We can define *removeUnits( )* as follows:

***removeUnits(G:* CFG) =**

4. Let G' =G.

5. Until no unit productions remaining G' do:

5.1. Choose some unit production $X \rightarrow Y$.

5.2. Remove it from G'.

5.3. Consider only rules that still remain in G'. For every rule $Y \rightarrow \beta$ where $\beta \in V^*$, do:

Add to G' the rule $X \rightarrow \beta$ unless that is a rule that has already been removed once.

6. Return G'.

EXAMPLE : Removing Unit Productions:

S→ XY

$X \rightarrow A$

$A \rightarrow B \mid$ a

B→b

$Y \rightarrow T$

$T \rightarrow Y \mid$ c

• Every string in $L(G')$ is also in $L(G)$: Every new rule can be simulated by old rules.

Conversion to Chomsky Normal Form

1. Remove all ε-rules, using the algorithm *removeEps*.

2. Remove all unit productions (rules of the form $A \rightarrow B$).

3. Remove all rules whose right hand sides have length greater than 1 and include a terminal:

   (e.g., $A \rightarrow aB$ or $A \rightarrow BaC$)

4. Remove all rules whose right hand sides have length greater than 2:

   (e.g., $A \rightarrow BCDE$)

Removing ε-Productions

Remove all ε productions:

(1) If there is a rule $P \rightarrow \alpha Q \beta$ and Q is nullable,

Then: Add the rule $P \rightarrow \alpha\beta$.

(2) Delete all rules $Q \rightarrow \varepsilon$.

Example:

$S \rightarrow aA$

$A \rightarrow B \mid CDC$

$B \rightarrow \varepsilon$

$B \rightarrow a$

$C \rightarrow BD$

$D \rightarrow b$

$D \rightarrow \varepsilon$

3.8.3 Converting to Chomsky Normal Form

There exists a straightforward four-step algorithm that converts a grammar $G = (V, \sum, R, S)$ into a new grammar *Gc* such that *Gc* is in Chomsky normal form and $L(Gc) = L(G) - \{\varepsilon\}$. Define:

***converttotoChomsky(* G: CFG) =**

1. **Let *Gc* be the result of removing from *G* all ε-rules, using the algorithm *removeEps( )***
2. **Let *Gc* be the result of removing from *Gc* all unit productions (rules of the form *A* ©B) using the algorithm *removeUnits* defined below. It is important to remove ε-rules first, before applying *removeUnits( ).* Once this step has been completed, all rules whose right- hand sides have length 1 are in Chomsky normal form (i.e . they are composed of a single terminal symbol).**
3. **Let *Gc* be the result of removing from *Gc* all rules whose right-hand sides have length greater than 1 and include a terminal (e.g : *A*© *aB* or *A* © BaC). This step is simple and can be performed by the algorithm *remove Mixed* given below. Once this step has been completed all rules whose right-hand sides have length 1 or 2 are in Chomsky normal form.**
4. **Let *Gc* be the result of removing from *Gc* all rules whose right-hand sides have length greater than 2 (e.g : *A* © *BCD£).* This step too is simple. It can be performed by the algorithm *remove Long* given below.**
5. **Return *Gc.***

Unit Productions

A ***unit production*** is a rule whose right-hand side consists of a single nonterminal symbol.

Example:

$S \rightarrow X\,Y$

$X \rightarrow A$

$A \rightarrow B \mid \mathrm{a}$

$B \rightarrow \mathrm{b}$

$Y \rightarrow T$

$T \rightarrow Y \mid \mathrm{c}$

Removing Unit Productions

removeUnits(G) =

1. Let $G' = G$.
2. Until no unit productions remain in G' do:
   2.1 Choose some unit production $X \rightarrow Y$.
   2.2 Remove it from G'.
   2.3 Consider only rules that still remain. For
   every rule $Y \rightarrow \beta$, where $\beta \in V^*$, do:
   Add to G' the rule $X \rightarrow \beta$ unless it is a rule
   that has already been removed once.
3. Return G'.

After removing epsilon productions and unit productions, all rules whose right hand sides have length 1 are in Chomsky Normal Form.

Example:

$S \to XY$ $X \to A$ $A \to B \mid a$ $B \to b$ $Y \to T$ $T \to Y \mid c$	$S \to XY$ $A \to a \mid b$ $B \to b$ $T \to c$ $X \to a \mid b$ $Y \to c$

Mixed Rules

removeMixed(G) =

1. Let $G' = G$.
2. Create a new nonterminal T_a for each terminal a in Σ.
3. Modify each rule whose right-hand side has length greater than 1 and that contains a terminal symbol by substituting T_a for each occurrence of the terminal a.
4. Add to G, for each T_a, the rule $T_a \to a$.
5. Return G'.

$A \to a$ $A \to a\,B$ $A \to BaC$ $A \to BbC$	$A \to a$ $A \to T_a B$ $A \to BT_a C$ $A \to BT_b C$ $T_a \to a$ $T_b \to b$

Long Rules

removeLong(G) =

1. Let $G' = G$.
2. For each rule r of the form:

$$A \to N_1 N_2 N_3 N_4 \ldots N_n,\ n > 2$$

create new nonterminals $M_2, M_3, \ldots M_{n-1}$.

3. Replace r with the rule $A \rightarrow N_1M_2$.

4. Add the rules:

$$M_2 \rightarrow N_2M_3,$$

$$M_3 \rightarrow N_3M_4, \ldots$$

$$M_{n-1} \rightarrow N_{n-1}N_n.$$

5. Return G'.

Example:

$A \rightarrow BCDEF$

$S \rightarrow aACa$

$A \rightarrow B \mid a$

$B \rightarrow C \mid c$

$C \rightarrow cC \mid \varepsilon$

removeEps returns:

$S \rightarrow aACa \mid aAa \mid aCa \mid aa$

$A \rightarrow B \mid a$

$B \rightarrow C \mid c$

$C \rightarrow cC \mid c$

$S \rightarrow aACa \mid aAa \mid aCa \mid aa$

$A \rightarrow B \mid a$

$B \rightarrow C \mid c$

$C \rightarrow cC \mid c$

Next we apply *removeUnits*:

Remove $A \rightarrow B$. Add $A \rightarrow C \mid c$.

Remove $B \rightarrow C$. Add $B \rightarrow cC$ ($B \rightarrow c$, already there).

Remove $A \rightarrow C$. Add $A \rightarrow cC$ ($A \rightarrow c$, already there).

So *removeUnits* returns:

$S \rightarrow aACa \mid aAa \mid aCa \mid aa$

$A \rightarrow a \mid c \mid cC$

$B \rightarrow c \mid cC$

$C \rightarrow cC \mid c$

Next we apply *removeMixed*, which returns:

$S \rightarrow T_aACT_a \mid T_aAT_a \mid T_aCT_a \mid T_aT_a$
$A \rightarrow a \mid c \mid T_cC$
$B \rightarrow c \mid T_cC$
$C \rightarrow T_cC \mid c$
$T_a \rightarrow a$
$T_c \rightarrow c$

$S \rightarrow T_aACT_a \mid T_aAT_a \mid T_aCT_a \mid T_aT_a$
$A \rightarrow a \mid c \mid T_cC$
$B \rightarrow c \mid T_cC$
$C \rightarrow T_cC \mid c$
$T_a \rightarrow a$
$T_c \rightarrow c$

Finally, we apply *removeLong*, which returns:

$S \rightarrow T_aS_1$ $S \rightarrow T_aS_3$ $S \rightarrow T_aS_4$ $S \rightarrow T_aT_a$

$S_1 \rightarrow AS_2$ $S_3 \rightarrow AT_a$ $S_4 \rightarrow CT_a$

$S_2 \rightarrow CT_a$
$A \rightarrow a \mid c \mid T_cC$

$B \rightarrow c \mid T_cC$

$C \rightarrow T_cC \mid c$

$T_a \rightarrow a$

$T_c \rightarrow c$

3.15

Module-4

RASHMI. M
Asst. Prof,
Dept. of CSE,
RNSIT

RASHMI. M
Asst. Prof.,
Dept. of CSE,
RNSIT

Module-4

Properties of CFLs

We know that $L = \{a^n b^n \mid n \geq 0\}$ is context-free. But, $L = \{a^n b^n c^n \mid n \geq 0\}$ is not context free. This is because, using stack it is not possible to count all three regions and compare them. We know that, $L = \{w.w^R \mid w \in \{a,b\}^*\}$ is context free language. But, language $L = \{ww \mid w \in \{a,b\}^*\}$ is not context free.

Where do CFLs fit? In this, we see the relationship between the regular languages and context free languages.

Theorem:- The context free languages properly contain the regular languages.

Proof:- We know that every regular language is context free since we can write a PDA to accept regular language also. Now, we should show that there exists atleast one context free language that is not regular.

Let us construct a context-free language using regular language.

→ We assume that L is regular and it is accepted by some DFA $M = (Q, \Sigma, \delta, q_0, F)$.

→ From M, we construct a PDA $M' = (Q', \Sigma', T', \delta', q_0', F')$ to accept L. In this case, M' is simply M ignoring the stack as shown below. Let $M' = (Q, \Sigma, \phi, \delta', q_0, F)$ where δ' is constructed as shown below:

For every transition $\delta(q_i, a_j) = c$ introduce the transition $\delta' = (q_i, c, \epsilon) = (a_j, \epsilon)$

→ NOW, M' behave identically to M and hence $L(M) = L(M')$. So, the regular languages are a subset of the CFL.

→ The regular languages are a proper subset of the CFL because, there exists at least one language $L = \{a^n b^n \mid n \geq 0\}$ which is context free but not regular.

Pumping lemma for CFLs :

State and Prove pumping lemma for context free languages.

Statement: Let L be the CFL and is infinite. Let z is sufficiently long string and $z \in L$ so that $|z| \geq n$ where n is some positive integer. If the string z can be decomposed into combination of strings.

$$z = uvwxy$$

such that $|vwx| \leq n$, $|vx| \geq 1$, then $uv^iwx^iy \in L$ for $i = 0, 1, 2, \ldots$ where

→ n is the length of the longest string that can be generated by the parse tree where the same non-terminal occurs twice on the same path through the tree.

→ The string z is sufficiently long so that it can be decomposed into various substrings u, v, w, x and y in that sequence.

→ The two substrings v and x are present somewhere in z.

✱

→ The substring u appears before v, the substring w is in between v and x and the substring y appears after x.

→ The string w in between v and x cannot be too long since $|vwx| \leq n$ for some positive integer n.

→ Both the substrings v and x cannot be empty since $|vx| \geq 1$. one of them can be empty.

→ If all the points mentioned above are satisfied, and if we duplicate substring v and x same number of times, the resultant string will be definitely be in L and the string $z \in L$ is context free. otherwise, the string $z \in L$ is not context free.

Proof: According to pumping lemma, it is assumed that string $z \in L$ is finite and is context free language. We know that z is string of terminals which is derived by applying series of productions. Proof of this theorem leads to the following two cases

Case 1: To generate a sufficiently long string z (it is assumed that the string is infinite), one or more Variables (non terminals) must be recursive (Note that infinite string can be generated if the grammar has some non-terminal A such that

$$A \Rightarrow^{*} \alpha A \beta$$

for some α and β) and should be applied more than once

Case 2: $z \in L$ implies that after applying some/all productions some number of times, we get a string of terminals and the derivation stops. If we can prove these two cases, we have the proof for pumping lemma.

Proof of Case 1: To Generate a sufficiently long string z (It is assumed that the string is infinite), one or more Variables (non terminals) must be recursive. Let us assume that the language is infinite and the grammar has a finite number of Variables and each production has finite length. The only way to derive sufficiently long string using such productions is that the grammar should have one or more recursive Variables. Assume that no Variable is recursive.

Since no Variable is recursive, each Variable must be defined only in terms of terminals / other Variables. Since those Variables are also non-recursive, they have to be defined in terms of terminals and other Variables and so on. If we keep applying the productions like this, there are no Variables at all in the final derivation and finally we get a string of terminals and the generated string is finite. From this, we conclude that there is a limit on the length of the string that is generated from the start symbol S. This contradicts our assumption that the languages is finite. Therefore, the assumption that one or more Variables are non-recursive is incorrect. It means that one or more Variables are recursive and hence the proof.

Proof of Case 2: $z \in L$ implies that after applying some / all productions some number of times, we get finally string of terminals and the derivation stops. Let $z \in L$ is sufficiently long string and so the derivation must have involved recursive use of some non-terminal.

A and the derivations must have the form:

$$S \Rightarrow^* uAy$$

Note that any derivations should start from the start symbol S. Since A is used recursively, the derivation can take the following form:

$$S \overset{*}{\Rightarrow} uAy \overset{*}{\Rightarrow} uvAxy$$

and the final derivation should be of the form

$$S \overset{*}{\Rightarrow} uAy \overset{*}{\Rightarrow} uvAxy \overset{*}{\Rightarrow} uvwxy = z$$

It implies that the following derivations

$$A \overset{*}{\Rightarrow} vAx$$

and

$$A \overset{*}{\Rightarrow} w$$

are also possible, From this we can easily conclude that the derivation

$$A \overset{*}{\Rightarrow} vwx$$

must also be possible. Next, we have to prove that the longest string vwx is generated without recursion since it is assumed that $|vwx| \leq n$. This can be easily proved since CFG that generates CFL does not contain ε-productions or unit productions. It shows that every derivation step either increases the length of the sentential form (using recursive variable) or introduces a terminal. The derivation

$$A \overset{*}{\Rightarrow} vAx$$

Used earlier clearly shows that

$$|vx| \geq 1$$

Note from the derivation

$$S \overset{*}{\Rightarrow} uAy \overset{*}{\Rightarrow} uvAxy$$

that uvAxy occurs in the derivation, and

$$A \overset{*}{\Rightarrow} vAx$$

and

$$A \overset{*}{\Rightarrow} w$$

are also possible, it follows that

$$uv^i wx^i y \in L$$

and hence the proof.

Note :- Showing a language is Context-free.

The given language can be represented as CFL -

→ If it belongs to context free grammar.

→ If a Push down Automata can be designed for the given language.

→ If the closure properties are followed.

The language can be proved to be non Context free using pumping lemma.

Note : Pumping Theorem for CFL

→ The pumping lemma for regular sets states that every sufficiently long string in a regular set contains a short substring that can be pumped. In other words, if a long string is given and if we push or pump any number of substrings in any number of times then we always get a regular set.

→ According to pumping lemma for CFL's there are always two short substrings which are close to each other and these both the substrings can be repeated as many times as required.

Applications of Pumping lemma for CFLs :

The Pumping Lemma for CFLs is used to Prove that certain languages are not context free languages. Note that pumping lemma cannot be used to prove that certain languages are context free.

The general strategy used to prove that a given language is not context free is shown below :

→ Assume that the language L is infinite & it is context free.

→ Select the string say z, where $|z| \geq n$ and break it into substrings u, v, w, x, and y such that $z = uvwxy$ where $|vwx| \leq n$ and $|vx| \geq 1$

→ Find any i such that $uv^iwx^iy \notin L$.

According to pumping lemma, $uv^iwx^iy \in L$. So, the result is a contradiction to the assumption that the language is context free. Therefore, the given language L is not context free.

Examples :-

1) Show that $L = \{a^p \mid p = q^2\}$ is not context free.

Sol:- Assume that L is context-free and is infinite.

So, we can apply pumping-lemma.

Let z be any string $\in L$.

→ Since $|z| \geq n$, A/c to Pumping lemma we can split the string $z = uvwxy = a^p$

such that $|vwx| \leq n$ and $|vx| \geq 1$

so that $uv^iwx^iy \in L$ for $i = 0, 1, 2, \ldots$

Let $w = a^p \mid p = q^2$

Let $q = 2$

$\therefore w = a^{2^2} = a^4 = a\,a\,a\,a$

$w = u\,v^i\,w\,x^i\,y$

for $i = 2$

$w = u\,v^2\,w\,x^2\,y = u\,v\,v\,w\,x\,x\,y$

$w = \underset{u}{a}\ \underset{v}{a}\ \underset{w}{a}\ \underset{y}{a}$

Note:- x is absent here.

$w = a\,a\,a\,a\,a\,a = a^5 \notin L$

$a^5 \Rightarrow 5$ is not the square of any numbers.

$\therefore$ So, the language $L = \{ a^p \mid p = q^2 \}$ is not Context free.

[Refer class notes for more examples.]

1) Show that $L = \{ a^n b^n c^n \mid n \geq 0 \}$ is not a context free language.

Sol: Let us assume that

$L = a^n b^n c^n$ is a context free language

Let, w be any string such that $w \in L$

Let

$w = uvwxy$

Let $|vx| \geq 1$ and $|vwx| \leq n$

Now let us consider

$w = uv^i wx^i y$

i.e we have additional occurrences of v and x.

consider various cases :

Case 1 : Consider $i = 0$ then,

$w = uv^i wx^i y = uv^0 wx^0 y$

$w = \underbrace{a\ a}_{u}\ \underbrace{b\ b}_{v}\ \underbrace{c}_{w}\ \underbrace{c}_{x\ y}$

if $i = 0$, then v^0 and x^0. That means v and x are absent then,

$w = uwy$

i.e, $w' = aabC \notin L$

Hence our assumption of L being CFG is wrong.

Case 2 : Consider $i = 2$ then,

$w = uv^i wx^i y = uv^2 wx^2 y$

$w = uvvwxxy$

$w = \underbrace{a\ a}_{u}\ \underbrace{b\ b}_{v}\ \underbrace{c}_{w}\ \underbrace{c}_{x\ y}$

$w = aabbbccc$

$w = a^2 b^3 c^3 \notin L$ [Assumption is wrong]

∴ This proves that the given language L is not a context free.

1) S.T $L = \{a^n b^n c^n \mid n \geq 0\}$ is not context free.

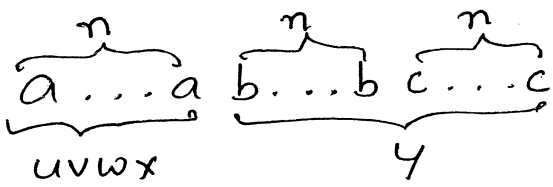
Sol:- Let L is context free and is infinite.

Let $z = a^n b^n c^n \in L$

Note that $z \geq n$ and so we can split z into $uvwxy$ such that

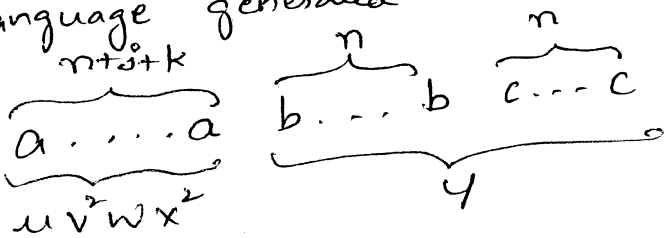
$|vwx| \leq n$ and $|vx| \geq 1$

and so $uv^i wx^i y \in L$ for $i = 0, 1, 2, \ldots$



Let $v = a^j$, $x = a^k$ where $|vx| = j+k \geq 1$ and $|vwx| \leq n$

A/c to pumping lemma, $uv^2wx^2y \in L$ for $i = 2$ and the language generated is as shown below:



Note that $uv^2wx^2y = a^{n+j+k} b^n c^n \notin L$ when $j+k \geq 1$

Note that the string should have some number of a's followed by equal number of b's and c's.

But a/c to pumping lemma, $uv^2wx^2y \in L$, which is the contradiction. So, the given language is not context free.

Important closure properties of CFLs

The context free languages are closed under some operation means after performing that particular operation on those CFLs the resultant language is context free language. These properties are as below:

1. The CFLs are closed under union.
2. The CFLs are closed under concatenation.
3. The CFLs are closed under Kleene closure.
4. The CFLs are not closed under intersection
5. The CFLs are not closed under complement.

Theorem 1:- Show that if L_1 and L_2 are context free languages then $L = L_1 \cup L_2$ is also context free.

i.e, the CFLs are closed under union.

Proof:- Let L_1 and L_2 are two CFLs generated by the CFGs:

$G_1 = (V_1, T_1, P_1, S_1)$ and

$G_2 = (V_2, T_2, P_2, S_2)$ respectively

and assume that V_1 and V_2 are disjoint.

Now, let us consider the language L_3 generated by the grammar:

$G_3 = (V_1 \cup V_2 \cup S_3, T_1 \cup T_2, P_3, S_3)$

where,

→ S_3 is a start symbol for the grammar G_3 and $S_3 \notin (V_1 \cup V_2)$

→ $P_3 = P_1 \cup P_2 \cup \{S_3 \rightarrow S_1 | S_2\}$

If we assume $w \in L_1$, then the possible derivation from S_3 is

$$S_3 \Rightarrow S_1 \overset{*}{\Rightarrow} w$$

(or) If we assume $w \in L_2$, then the possible derivation from S_3 is

$$S_3 \Rightarrow S_2 \overset{*}{\Rightarrow} w$$

So, if $w \in L_3$, one of the derivations is possible.

$$S_3 \Rightarrow S_1$$

(or)

$$S_3 \Rightarrow S_2$$

is possible. In the first case, all the variables in V_1 and all the terminals in T_1 may be used to get the derivation

$$S_1 \overset{*}{\Rightarrow} w$$

The above derivation uses only the productions in P_1.

In the second case, all the variables in V_2 and all the terminals in T_2 may be used to get the derivation

$$S_2 \overset{*}{\Rightarrow} w$$

The above derivation uses only the productions in P_2.

$\therefore L_3 = L_1 \cup L_2$

Thus it is proved that CFLs are closed under union.

Example :-

If $L_1 = \{a^n b^n : n \geq 1\}$ and $L_2 = \{(ab)^m : m \geq 1\}$ Show that $L_3 = L_1 \cup L_2$ is a CFL.

Sol :- To generate L_1, the productions are :

$$S_1 \longrightarrow aS_1b \mid ab$$

To generate L_2, the productions are :

$$S_2 \longrightarrow abS_2 \mid ab$$

To generate L_3, the productions are :

$$S_3 \longrightarrow S_1 \mid S_2$$
$$S_1 \longrightarrow aS_1b \mid ab$$
$$S_2 \longrightarrow abS_2 \mid ab$$

$\therefore$ L_1 is the language generated from $L(G_1)$
L_2 is the language generated from $L(G_2)$

$\therefore$ $G_1 = (V_1, T_1, P_1, S_1)$
$G_2 = (V_2, T_2, P_2, S_2)$

$\therefore$ L_3 is the language generated from $L(G_3)$

$$G_3 = (V_1 \cup V_2 \cup \{S_3\}, T_1 \cup T_2, P_1 \cup P_2 \cup \{S_3 \rightarrow S_1 \mid S_2\}, S_3)$$

Theorem 2 :-

Show that if L_1 and L_2 are two CFLs, then $L_1 . L_2$ is also CFL.

i.e Context free languages are closed under concatenation.

Proof :- Let L_1 and L_2 are two CFLs generated from the CFG.

$G_1 = (V_1, T_1, P_1, S_1)$ and $G_2 = (V_2, T_2, P_2, S_2)$

Now, let us consider the language L_4 generated by the grammar

$G_4 = (V_1 \cup V_2 \cup S_4, T_1 \cup T_2, P_4, S_4)$

where

→ $P_4 = P_1 \cup P_2 \cup \{S_4 \rightarrow S_1 S_2\}$

→ S_4 is the start symbol for the grammar G_4. and $S_4 \notin (V_1 \cup V_2)$

∴ It is clear from this that the grammar G_4 is context free and the language generated by this grammar is also context free.

∴ $L_4 = L_1 . L_2$

Thus, it is proved that CFLs are closed under concatenation.

Example :-

If $L_1 = \{ a^n b^n : n \geq 1 \}$ and $L_2 = \{ (ab)^m : m \geq 1 \}$ S.T $L_4 = L_1 \cdot L_2$ is a CFL.

Sol :- To generate L_1, the productions are :

$S_1 \rightarrow aS_1b \mid ab$

To generate L_2, the productions are :

$S_2 \rightarrow abS_2 \mid ab$

To generate L_4, the productions are :

$S_4 \rightarrow S_1 \cdot S_2$

$S_1 \rightarrow aS_1b \mid ab$

$S_2 \rightarrow abS_2 \mid ab$

$\therefore G_4 = (V_1 \cup V_2 \cup \{S_4\}, T_1 \cup T_2, P_1 \cup P_2 \cup \{S_4 \rightarrow S_1 S_2\})$

Theorem 3 :- CFLs are closed under star-closure

Proof :- Let L_1 be the CFL generated from the grammar $G_1 = (V_1, T_1, P_1, S_1)$ and

Let us consider the language L_5 generated by the grammar $G_5 = (V_1 \cup S_5, T_1, P_5, S_5)$

Where,

→ S_5 is the start symbol for the grammar G_5

→ $P_5 = P_1 \cup \{S_5 \rightarrow S_1 S_5 \mid \epsilon\}$

It is clear from this that the grammar G_5 is context free and the language generated by this grammar is also context free.

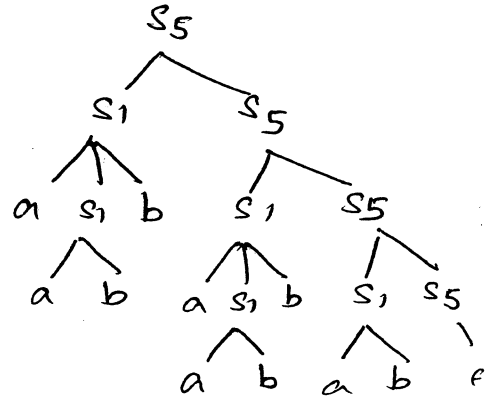
$\therefore$ $L^* = L_5$

Example :-

If $L = \{a^n b^n \mid n \geq 1\}$. Show that $L = L_1^*$ is also a CFG.

Sol :- $S_5 \rightarrow S_1 S_5 \mid \epsilon$

$S_1 \rightarrow a S_1 b \mid ab$



$(a^2b^2)(a^2b^2)(ab)$

Theorem : 4 :-

CFLs are not closed under intersection

Statement : If L_1 and L_2 are context free languages, it is not always true that $L_1 \cap L_2$ is context free language.

Proof :- Let us prove this theorem by taking counter examples. Consider the two languages

$L_1 = \{a^n b^n c^m \mid n \geq 0, m \geq 0\}$ and

$L_2 = \{a^m b^m c^n \mid n \geq 0, m \geq 0\}$

The two languages are context free, as we can easily obtain the corresponding CFG

$S \rightarrow S_1 S_2$

$S_1 \rightarrow a S_1 b \mid \epsilon$

$S_2 \rightarrow C S_2 \mid \epsilon$

and

~~$S \rightarrow aS \mid S_1$~~

~~$S_1 \rightarrow bS_1c \mid \epsilon$~~

$S \rightarrow S_1 S_2$

$S_1 \rightarrow a S_1 b \mid \epsilon$

$S_2 \rightarrow c S_2 \mid \epsilon$

Now, let us take

$L_1 \cap L_2 = \{ a^n b^n c^n \mid n \geq 0 \}$

we have already proved earlier that this language is not context free. Thus we can prove that the family of CFL's is not closed under intersection.

$L_1 = \{ a^n b^n c^m \mid n \geq 0, m \geq 0 \}$ and

$L_2 = \{ a^m b^m c^n \mid n \geq 0, m \geq 0 \}$

Let $m = 2$ and $n > 4$

$L_1 = \{ a^4 b^4 c^2 \} = aaaa\,bbbb\,cc$

$L_2 = \{ a^4 b^2 c^4 \} = aaaa\,bb\,cccc$

$L_1 \cap L_2 = aa\,bb\,cc$

i.e $a^n b^n c^n$ language is followed. But, $a^n b^n c^n$ is not a CFL.

Theorem 5:

CFLs are not closed under complementation.

Statement:- If L is context free language, it is not true that complement of L is context free language.

Proof:- Let us prove this theorem by contradiction. Suppose that CFLs are closed under complementation. So, if L_1 and L_2 are CFLs, then $\bar{L}_1$ and $\bar{L}_2$ are also context free.

We have already proved that CFLs are closed under union. So, $\bar{L}_1 \cup \bar{L}_2$ must be context free. Since we have assumed that the CFLs are closed under complementation,

$\overline{L_1 \cup L_2}$ must be context free. But, according to De-Morgan's law we have :

$$\overline{L_1 \cup L_2} = L_1 \cap L_2$$

So, $L_1 \cap L_2$ must be context free which is a contradiction. Since the CFLs are not closed under intersection, our assumption that the CFLs are closed under complementation is not true. So, the family of CFLs are not closed under complementation.

Note :- we have seen that the following languages are not context free and it is not possible to represent using context free grammars and so we cannot have the corresponding PDA for these types of languages. Even then some of the important points we can make at this stage are :

1. $L = \{ a^n b^n c^n \mid n \geq 0 \}$
2. $L = \{ w \mid w \in \{a, b, c\}^* \text{ where } n_a(w) = n_b(w) = n_c(w) \}$
3. $L = \{ ww \mid w \in \{a, b\}^* \}$
4. $L = \{ a^{n!} \mid n \geq 0 \}$
5. $L = \{ a^p b^q \mid p = q^2 \}$

→ The regular languages are the subset of CFL and so CFL are more powerful than the regular languages.

→ The above statement automatically implies that PDAs are more powerful than the finite automaton.

→ But, the PDAs are not strong enough to accept some of the languages as pointed out earlier that are not context free and so we need much more powerful automaton than the PDA such as Linear Bounded Automaton or most powerful Turing machine.

Deterministic Context Free Language

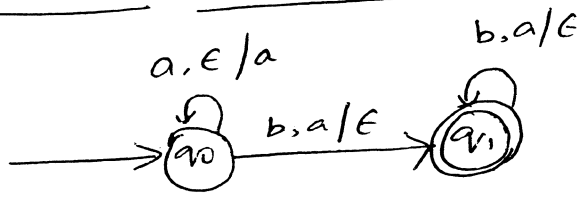
A language L is deterministic if and only if $L\$$ can be accepted by some PDA where $\$$ is end-of-string marker. Instead of $\$$ we can use line feed or carriage return.

4) Obtain a deterministic and non-deterministic PDA to accept the language:

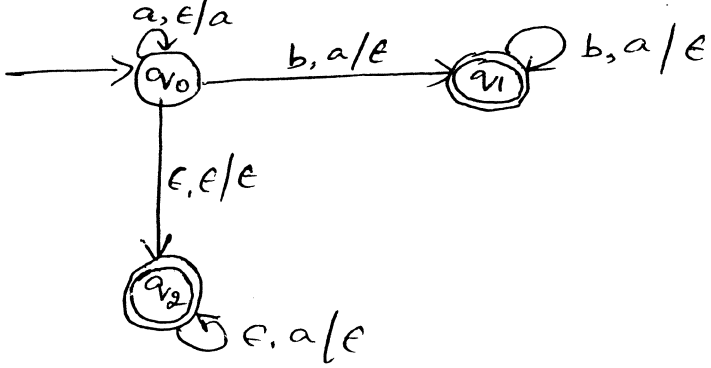
$$L = a^* \cup \{a^n b^n \mid n > 0\}$$

Sol:

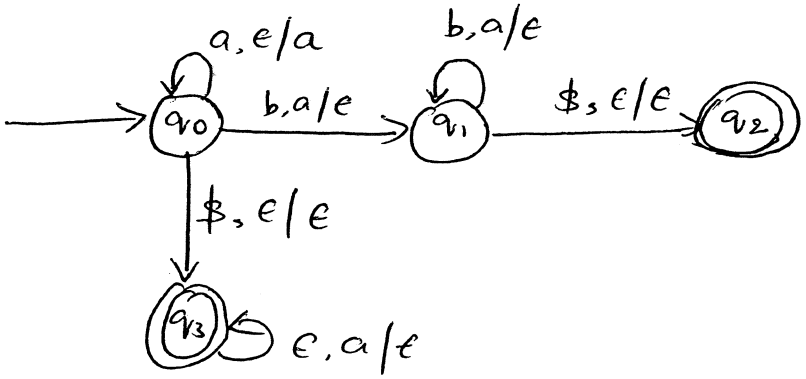
Non-deterministic PDA:



But, in state q_0, if we input only a's and not followed by b's, then also the string has to be accepted. In such situation, we can have another transition from state q_0 to final state q_2 where we delete all a's pushed onto the stack as shown:



Deterministic PDA : The deterministic PDA can be built with an end-of-string marker as shown below :



RASHMI.M
Asst. Prof,
Dept. of CSE, RNSIT

Turing Machine

The Turing machine provides an ideal theoretical model of a computer.

Turing machine are useful in several ways:

→ It is used for determining the undecidability of certain languages.

→ As an automaton, the Turing machine is the most general model, it accepts type-0 languages.

→ It can also be used for computing functions. It turns out to be a mathematical model of partial recursive functions.

→ Measuring the space and time complexity of problems.

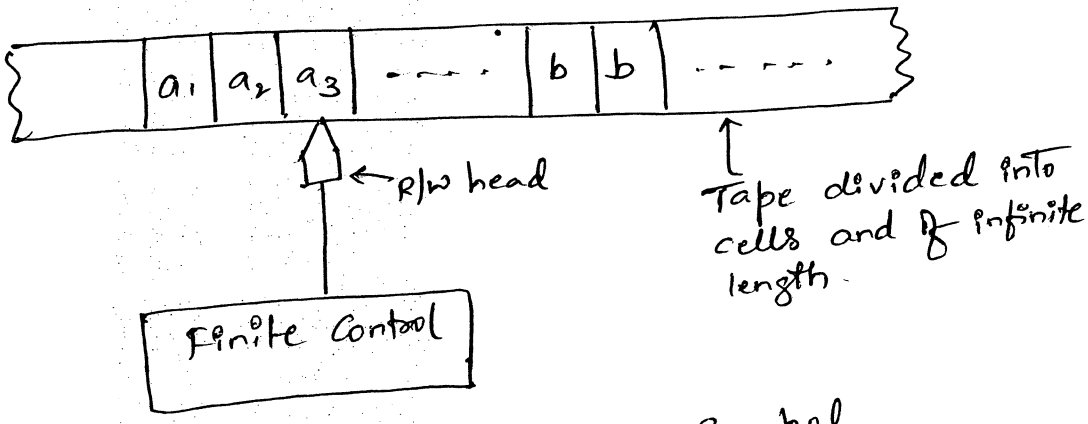
Turing assumed that while computing, a person writes symbols on a one-dimensional paper which can be viewed as a tape divided into cells. In turing machine one scans the cells one at a time and usually performs one of the three simple operations, namely:

(i) writing a new symbol in the cell being currently scanned.

(ii) Moving to the cell left of the present cell

(iii) Moving to the cell right of the present cell

Turing machine model :



→ Each cell can store only one symbol

→ The input to and the output from the finite state automaton are affected by the R/W head which can examine one cell at a time.

In one move, the machine examines the present symbol under the R/W head on the tape and the present state of an automaton to determine :

(i) A new symbol to be written on the tape in the cell under the R/W head.

(ii) A motion of the R/W head along the tape : either the head moves one cell left (L), or one cell right (R)

(iii) The next state of the automaton, and

(iv) whether to halt or not

Definition :

Turing machine M is a 7-tuple, namely $(Q, \Sigma, T, \delta, q_0, b, F)$, where

1. Q is a finite non empty set of states
2. T is a finite non-empty set of tape symbols.
3. $b \in T$ is the blank character
4. Σ is a non empty set of input symbols and is a subset of T and $b \notin \Sigma$.
5. δ is the transition function mapping (q, x) onto (q', y, D) where D denotes the direction of movement of R/W head.
   → $D = L$ or R (accordingly as the movement is to the left or right)
6. $q_0 \in Q$ is the initial state
7. $F \subseteq Q$ is the set of final states.

Note :

1) The acceptability of a string is decided by the reachability from the initial state to some final state.

2) δ may not be defined for some elements of $Q \times T$.

* If transition is not defined for any input in a particular state, then it means that the transition is leading to the <u>trap state</u>.

Representation of Turing Machines :-

We can describe a Turing machine employing

i) Instantaneous descriptions using move-relations

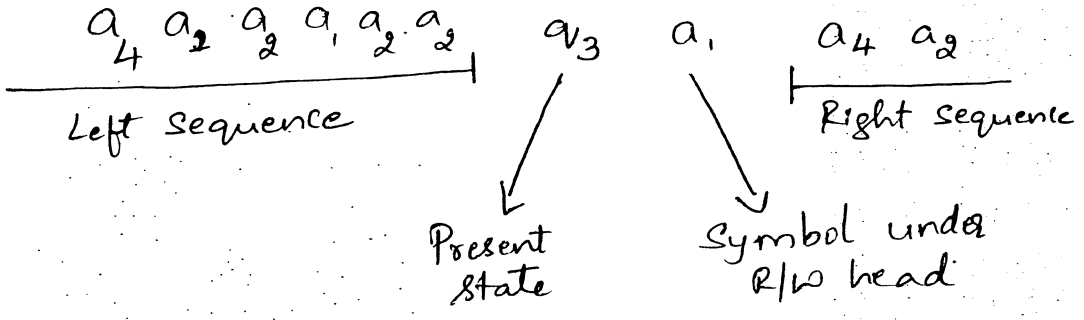
ii) Transition table

iii) Transition diagram

Representation by instantaneous descriptions

Definition : An ID of a turing machine M is a string $\alpha\beta\gamma$, where β is the present state of M, the entire input string is split as $\alpha\gamma$, the first symbol of γ is the current symbol a under the R/W head and γ has all the subsequent symbols of the input string, and the string α is the substring of the input string formed by all the symbols to the left of a.

Eg



Note :-

1) For constructing the ID, we simply insert the current state in the input string to the left of the symbol under the R/W head.

2) We observe that the blank symbol may occur as part of the left or right substring.

Representation by transition table

We give the definition of δ in the form of a table called the transition table.

$$\delta(q, a) = (\gamma, \alpha, \beta)$$

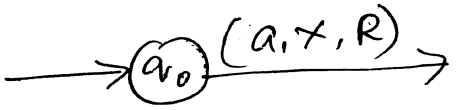
where,

γ → Denotes the new state into which the turing machine enters

α → It is written in the current cell

β → Gives the movement of the head (L or R)

Representation by transition diagram



→ The machine changes the state from q_0 to q_f.

→ The edge is labelled with (a, X, R)

→ where a is the current input symbol read from a tape which has to be replaced by X on the tape moving the read/write head to left / right.

→ The transition from one state to another state is indicated by a directed edge.

→ The start state is a state which has an arrow not originating from any node and entering into the state. This is labelled with "→".

→ The final states or accepting states which are in F are represented by double circles. The states which are not in F are represented by a single circle.

Standard Turing Machine :- Since there can be several variants of TM, the TM that we are studying now can be called as Standard Turing Machine with the following features :

→ The Turing machine has a tape that is divided into number of cells with each cell capable of storing only one symbol. The tape is unbounded (i.e, no boundary on the left as well as on the right) with any number of left or right moves.

→ The machine is deterministic. It can have either zero or one transition for each configuration.

→ Some of the symbols on the tape can be considered as the input. The symbols on the tape (some symbols or all the symbols) can be considered as the output whenever the TM halts.

Language acceptance by a Turing Machine :

The turing machine can do one of the following things :

→ Halt and accept by entering into final state.

→ Halt and reject. This is possible if the transition is not defined i.e $\delta(q, a)$ is not defined.

→ The TM will never halt and enters into an infinite loop. It is true that there is no algorithm to determine and tell whether a given machine always halts.

What is the language accepted by a Turing machine?

Definition :- Let $M = (Q, \Sigma, T, \delta, q_0, B, P)$ be a TM. The language $L(M)$ accepted by M is defined as -

$$L(M) = \{ w \mid q_0 w \vdash^* \alpha_1 P \alpha_2 \text{ where } w \in \Sigma^*, P \in F \text{ and } \alpha_1, \alpha_2 \in T^* \}$$

where $q_0 w$ is the initial ID and $\alpha_1 P \alpha_2$ is the final ID. The set of all those words w in Σ^* which causes M to move from start state q_0 to the final state P.

Note :-

→ The string w which is the string to be scanned should end with infinite numbers of blanks.

→ Initially, the machine will be in the start state q_0 with R/W head pointing to the first symbol of w from left.

→ After some sequence of moves, if the TM enters into final state and halts, then we say that the string w is accepted by TM.

→ The language accepted by TM is called Recursively enumerable language or RE language.

What is recursively enumerable language?

A language L is recursively enumerable, if it is accepted by a TM i.e, given a string w which is input to TM, the machine halts and outputs Yes if it belongs to the language. If w does not belong to the language L, the TM halts and outputs NO.

The languages with Turing machine which will always halts and output "yes" if it belongs to the languages or output "no" if it does not belongs to the language are called decidable languages or recursive languages.

The TM that always halt irrespective of whether they accept or not are a good model for an algorithm. If an algorithm exists to solve a given problem, then the problem is decidable otherwise it is un-decidable problem.

Design of Turing Machine:

What are the basic guidelines for designing a Turing machine?

→ After scanning the symbol below the R/W head is to know what activity has to be done in future. The machine must remember all the scanned symbols and the symbols to be scanned. The machine can remember the scanned symbol by moving or to next state.

→ The numbers of states in TM must be minimized. It is achieved by changing the state only when tape symbol is changed or there is a change in the movement of the R/W head.

1) Given the following ID of a Turing machine: $a_1 a_2 a_3 a_4 q_2 a_5 a_6 a_7 a_8 \ldots$ and $\delta(q_2, a_5) = (q_3, b_1, R)$ is the transition, what is the next ID?

Sol:- The ID : $\underbrace{a_1 a_2 a_3 a_4}_{\alpha} \underbrace{q_2}_{q} \underbrace{a_5 a_6 a_7 a_8}_{\beta}$

The above ID indicates that:
→ The machine is in state q_2
→ The next symbol to be scanned is a_5

when the transition $\delta(q_2, a_5) = (q_3, b_1, R)$ is applied the following transitions are performed:
→ The machine enters into state q_3.
→ The current input symbol a_5 is replaced by b_1
→ R indicates that the R/w head is moved one symbol towards right and the following ID is obtained:

$a_1 a_2 a_3 a_4 b_1 q_3 a_6 a_7 a_8 \ldots$

This can be represented by a move as shown below:

$a_1 a_2 a_3 a_4 q_2 a_5 a_6 a_7 a_8 \ldots \vdash a_1 a_2 a_3 a_4 b_1 q_3 a_6 a_7 a_8 \ldots$

2) Given the following ID of a TM: $a_1 a_2 a_3 a_4 q_2 a_5 a_6 a_7 a_8 \ldots$ and $\delta(q_2, a_5) = (q_3, b_1, L)$ is the transition, what is the next ID?

Sol:- The next ID is $a_1 a_2 a_3 q_3 a_4 b_1 a_6 a_7 a_8 \ldots$

Move is:-

$a_1 a_2 a_3 a_4 q_2 a_5 a_6 a_7 a_8 \ldots \vdash a_1 a_2 a_3 q_3 a_4 b_1 a_6 a_7 a_8 \ldots$

In general, the actions performed by TM depends on

→ The current state
→ The whole string to be scanned
→ The current position of the Read-write head

The action performed by the machine consists of:

→ changing the states from one state to another
→ Replacing the symbol pointed to by the read-write head.
→ Movement of the read-write head towards left or right.

1) Design a Turing Machine to accept the foll. language : $L = \{ a^n b^n \mid n \geq 0 \}$. Also write the instantaneous description for the string "aabb".

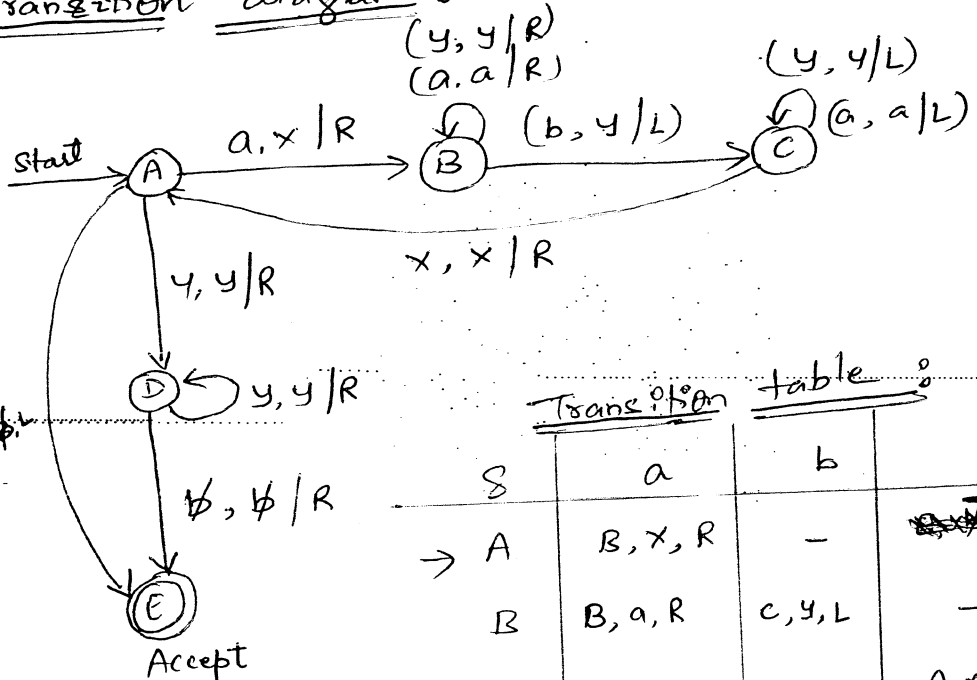
Sol:-

General Procedure :

Read-write head points to the first symbol of the string to be scanned.

1. change 'a' to 'x'
2. move Right to First 'b'
   If none : Reject
3. change 'b' to 'y'
4. Move left to leftmost 'a'
5. Repeat the above steps until no more a's.
6. Make sure no more b's remain

Transition diagram :



Transition table :

	S	a	b	x	y	b
→	A	B, X, R	–		D, y, R	
	B	B, a, R	C, y, L	–	B, y, R	–
	C	C, a, L	–	A, x, R	C, y, L	–
	D	–	–	–	D, y, R	E, b,
*	E	–	–	–	–	–

ID for "aabb" :

$\delta(q_0, aabb) \vdash x q_1 a b b$

$\vdash x a q_1 b b$

$\vdash x q_2 a y b$

$\vdash q_2 x a y b$

$\vdash x q_0 a y b$

$\vdash x x q_1 y b$

$\vdash x x y q_1 b$

$\vdash x x q_2 y y$

$\vdash x q_2 x y y$

$\vdash x x q_0 y y$

$\vdash x x y q_3 y$

$\vdash x x y y q_3 B$

$\vdash x x y y$

Accepted

ID for "aab" :

$\delta(q_0, aab) \vdash x q_1 a b$

$\vdash x a q_1 b$

$\vdash x q_2 a y$

$\vdash q_2 x a y$

$\vdash x q_0 a y$

$\vdash x x q_1 y$

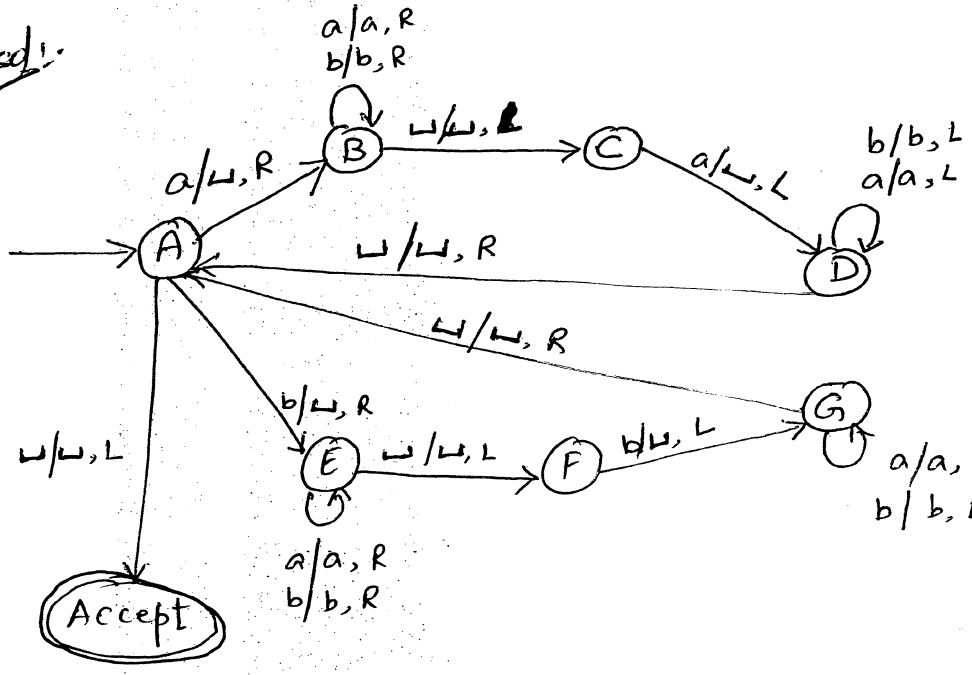
$\vdash x x y q_1$

Rejected

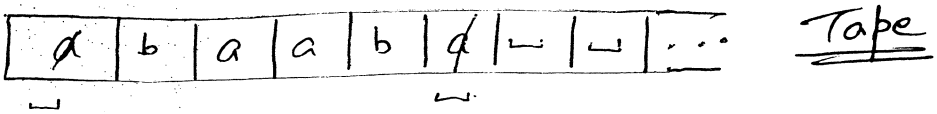
In state q_1, transition is not defined for the blank character. So it is leading to the dead state and the string is Rejected.

2) Write the Turing machine to accept even palindromes over the alphabet $\Sigma = \{a, b\}$

Sol:-



Transition Diagram



Tape

Approach :- Consider the string 'abaaba'. Assuming it is already present in the tape. In the tape, it looks like as shown in the above fig. For the first/to begin tape head will be pointing to the first symbol i.e a. When it reads the first symbol, replaces it with the blank character and it moves right till the end of the string ie until it encounters the blank character. When it encounters the blank character move left. The symbol to which the tape head is pointing should be same as the first symbol it was read before. If it is same, & move left replace the symbol with blank else reject the string. This procedure continues for the entire string until the string becomes blanks. If all are blank means then the string is accepted otherwise it is rejected.

ID for the string abaaba :

$\delta(A, abaaba) \vdash (\sqcup Bbaaba)$

$\vdash (\sqcup bBaaba)$

$\vdash (\sqcup baBaba)$

$\vdash (\sqcup baaBba)$

$\vdash (\sqcup baabBa)$

$\vdash (\sqcup baabaB)$

$\vdash (\sqcup baabCa)$

$\vdash (\sqcup baaDb\sqcup)$

$\vdash (\sqcup baDab\sqcup)$

$\vdash (\sqcup bDaab\sqcup)$

$\vdash (\sqcup Dbaab\sqcup)$

$\vdash (\sqcup baab\sqcup)$

$\vdash (\sqcup\sqcup baab\sqcup)$

$\vdash (\sqcup\sqcup Eaab\sqcup)$

$\vdash (\sqcup\sqcup aEab\sqcup)$

$\vdash (\sqcup\sqcup aaEb\sqcup)$

$\vdash (\sqcup\sqcup aabE\sqcup)$

$\vdash (\sqcup\sqcup aaFb\sqcup)$

$\vdash (\sqcup\sqcup aGa\sqcup\sqcup)$

$\vdash (\sqcup\sqcup GAa\sqcup\sqcup)$

$\vdash (\sqcup GH aa\sqcup\sqcup)$

$\vdash (\sqcup\sqcup Aaa\sqcup\sqcup)$

$\vdash$ (␣ ␣ ␣ B a ␣ ␣)
$\vdash$ (␣ ␣ ␣ a B ␣ ␣)
$\vdash$ (␣ ␣ ␣ C a ␣ ␣)
$\vdash$ (␣ ␣ D ␣ ␣ ␣)
$\vdash$ (␣ ␣ ␣ A ␣ ␣ ␣)
$\vdash$ (␣ ␣ Accept ␣ ␣ ␣)

String accepted.

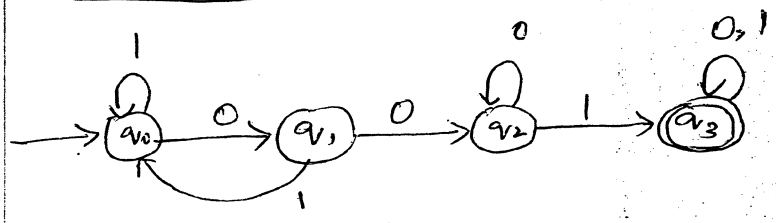
Transition diagram :-

δ / State	Tape symbol a	b	␣
A	(B, ␣, R)	(E, ␣, R)	(Accept, ␣, L)
B	(B, a, R)	(B, b, R)	(C, ␣, L)
C	(D, ␣, L)	–	–
D	(D, a, L)	(D, b, L)	(A, ␣, R)
E	(E, a, R)	(E, b, R)	(F, ␣, L)
F	–	(G, ␣, L)	–
G	(G, a, L)	(G, b, L)	(A, ␣, R)
Accept	–	–	–

2) Design a Turing machine to accept the foll. language : $L = \{w \mid w \in (0+1)^*\}$ Containing the substring 001.

Sol:- The DFA which accepts the language consisting of strings of 0's and is having a substring 001 is shown below :

Transition diagram for DFA :



Transition table

	δ	0	1
→	q_0	q_1	q_0
	q_1	q_2	q_0
	q_2	q_2	q_3
*	q_3	q_3	q_3

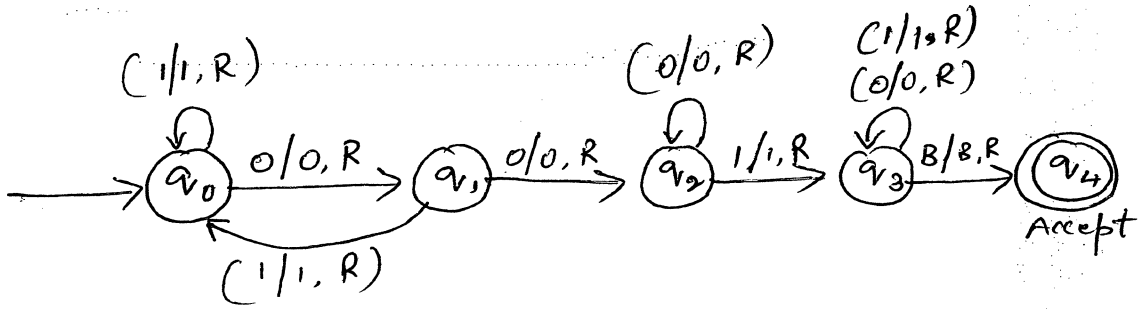
Approach :-

1) As the DFA processes the input string from left to right in only one direction, TM also processes the i/p string in only one direction.

2) For each scanned input symbol (either 0 or 1), in whichever state the DFA was in, TM also enters into the same states on same input symbols, replacing 0 by 0 and 1 by 1 and the tape head moves towards right.

3) So, the transition table for DFA and TM remains same (different format).

Transition diagram for TM :



Transition table :

δ	0	1	B
q_0	q_1, 0, R	q_0, 1, R	—
q_1	q_2, 0, R	q_0, 1, R	—
q_2	q_2, 0, R	q_3, 1, R	—
q_3	q_3, 0, R	q_3, 1, R	q_4, B, R
q_4	—	—	—

The TM is given by, $M = (Q, \Sigma, T, \delta, q_0, B, F)$

where,

$Q = \{q_0, q_1, q_2, q_3, q_4\}$

$\Sigma = \{0, 1\}$

$T = \{0, 1, B\}$

δ is shown in the form of transition table above

q_0 is the start state

B is the blank character

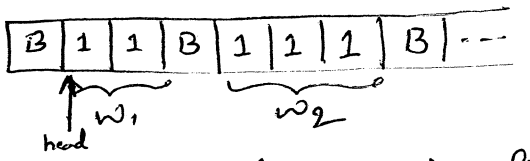
$F = \{q_4\}$ is the final state

4) Design a TM over {1, B} which can compute a concatenation function over $\Sigma = \{1\}$. If a pair of words (w_1, w_2) is the input, the output has to be $w_1 . w_2$.

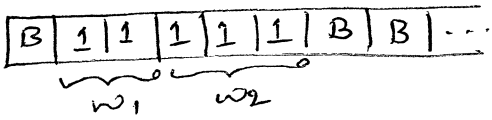
Sol:-

Design: Consider the string $w_1 = 11$, $w_2 = 111$

Input tape

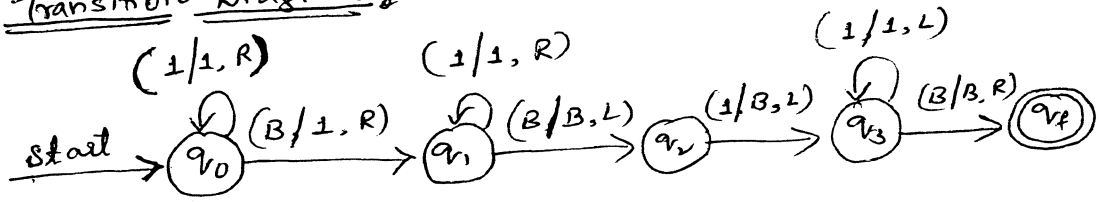


Output tape



Observe that in the first fig, B is in between w_1 and w_2. In the second fig, B is moved to the end.

Transition Diagram:



Transition table:

	δ	1	B
→	q_0	q_0, 1, R	q_1, 1, R
	q_1	q_1, 1, R	q_2, B, L
	q_2	q_3, B, L	—
	q_3	q_3, 1, L	q_f, B, R
*	q_f	—	—

The TM is given by: $M = (Q, \Sigma, T, \delta, q_0, B, F)$

where

$Q = \{q_0, q_1, q_2, q_3, q_f\}$

$\Sigma = \{1\}$

$T = \{1, B\}$

δ is shown using the transition table

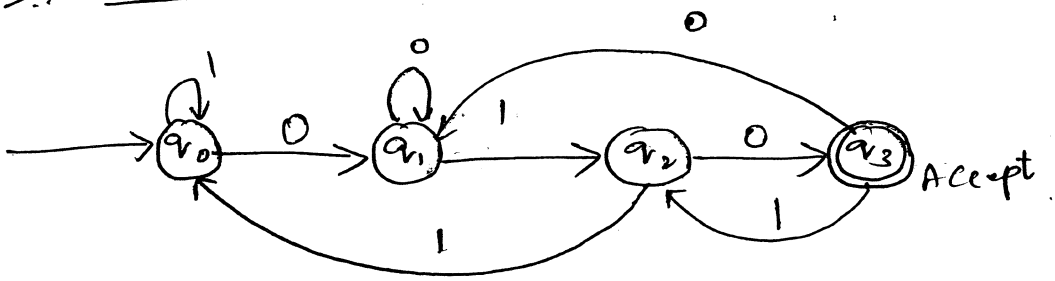
q_0 is the start state

B is the blank character

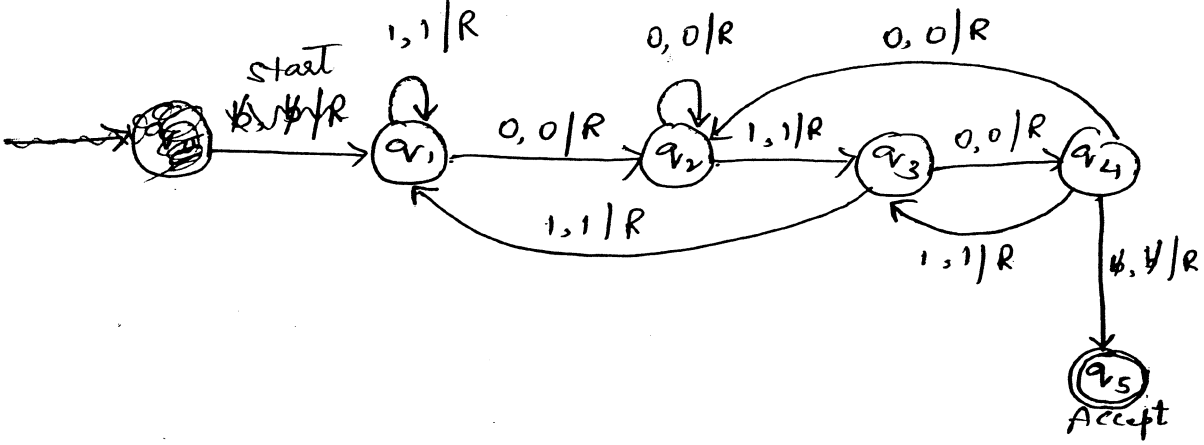
$F = \{q_f\}$ is the final state

5) Construct a TM to accept set of L of all strings over $\Sigma = \{0, 1\}$ ending with '010'.

Sol: DFSM:-



TM :-

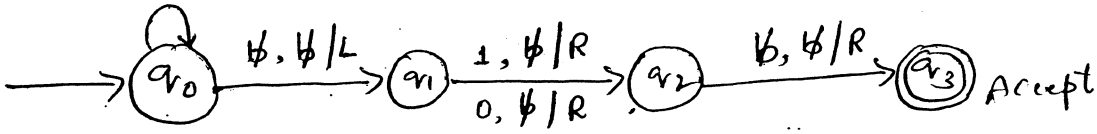


6) Design a TM on $\Sigma = \{0, 1\}$ which erases right most symbol.

Sol (1, 1 / R)
(0, 0 / R)

Eg

0	1	0	1	$\not b$	$\not b$	

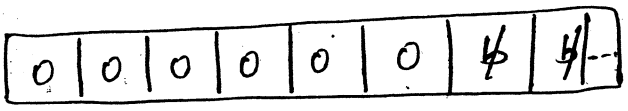


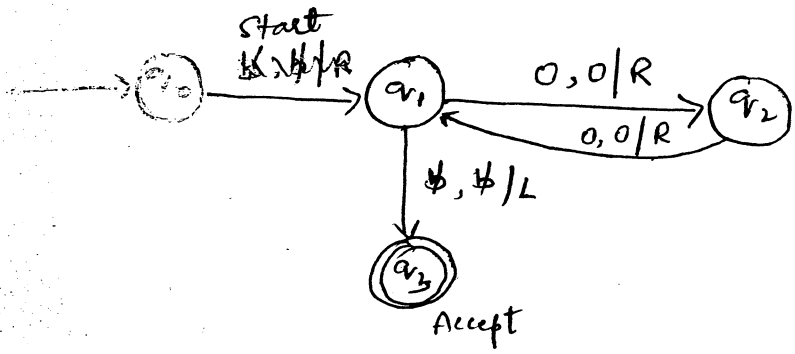
δ	0	1	$\not b$
q_0	$(q_0, 0, R)$	$(q_0, 1, R)$	$(q_1, \not b, L)$
q_1	$(q_2, \not b, R)$	$(q_1, \not b, R)$	–
q_2	–	–	$(q_3, \not b, R)$
q_3	–	–	–

7) Design a TM that accepts $L = \{0^{2^n} \mid n \geq 0\}$

Sol:-

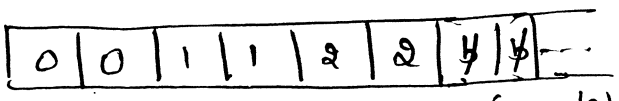
$n=0$	$n=1$	$n=2$	$n=3$
0^0	$0^2 = 00$	$0^4 = 0000$	$0^6 = 000000$





8) Design a TM that accepts $L = \{0^n 1^n 2^n \mid n \geq 1\}$.

Sol:-



Note:- The transition which belongs to state q_5 can be written in state q_4 (self-loop). In such a case, state q_5 becomes the final state.

Techniques for TM construction :

The TM that we have discussed so far is called "Standard Turing machine". Now, let us discuss some high-level conceptual tools using which we can construct TM's very easily.

What are the various techniques for TM construction?

→ TM with stationary head
→ Storage in the state
→ Multiple track TM
→ Subroutines

1. Turing Machine with stationary head :

In standard TM δ is defined as :

$$\delta(q, a) = (P, y, D)$$

where D stands for direction.
So D can be left or right denoted by L or R.
Here, after consuming the input symbol a, the R/W head moves either towards left or towards right. An option such that the R/W head remains in the same cell for some input symbols can be implemented. This can be implemented using the transition :

$$\delta(q, a) = (P, y, S)$$

This means that, the TM in state q, after consuming the input symbol a, replace the input symbol a by y and changes its state from q to P without updating R/W head either towards left or towards right.

The formal definition of TM with stay-option is given by :

$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where

$Q \Rightarrow$ set of final states

$\Sigma \rightarrow$ set of input alphabets

$\Gamma \rightarrow$ set of tape symbols

$\delta \rightarrow$ Transition function from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R, S\}$ indicating the TM may move towards left or right or stay in the same position after updating the symbol on the tape.

$q_0 \rightarrow$ start state

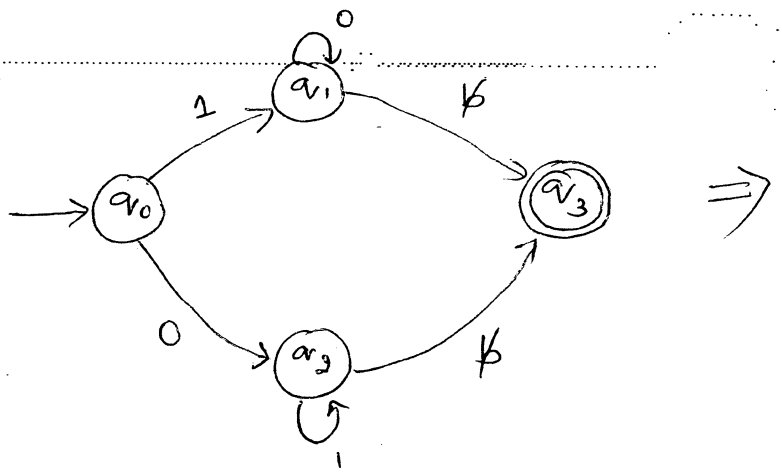
$B \rightarrow$ special symbol indicating blank character

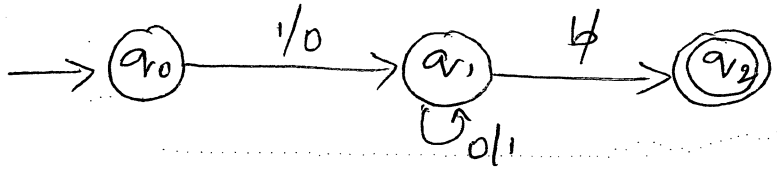
$F \rightarrow$ set of final states

2. Storage in the state :

In all our machines such as FA / PDA / TM, we used states to remember things. Apart from this, we can use a state to store a symbol as well. So, in TM where we store in a state, the state becomes a pair (q, a) where q is the state and a is the tape symbol stored in the state (q, a). So, the new set of states is given by : $Q \times \Gamma$

Eg $01^* \cup 10^*$

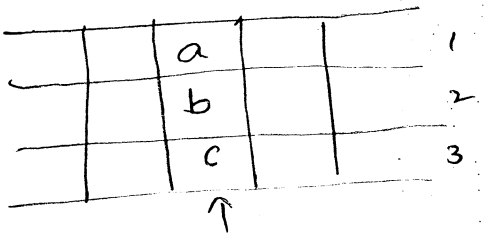




$$\delta([q_0, 0], 0) \longrightarrow ([q_1, 1], b, R)$$
$$\delta([q_1, 1], 1) \longrightarrow ([q_1, 1], b, R)$$
$$\delta([q_1, 1], b) \longrightarrow (q_2, b, R)$$
$$\delta([q_0, 1], 1) \longrightarrow ([q_1, 0], b, R)$$
$$\delta([q_1, 0], 0) \longrightarrow ([q_1, 0], b, R)$$
$$\delta([q_1, 0], b) \longrightarrow (q_2, b, R)$$

3. Multiple Track Turing Machine : In a standard TM only one tape was used to store the data. In multiple track TM, a single tape is assumed to be divided into a number of tracks. If a single tape is divided into k tracks, then the tape alphabets consists of k-Tuples of tape symbols. So, only difference between standard TM and TM with multiple tracks is the set of tape symbols. In case of standard TM, tape symbols are denoted by symbol T whereas in case of TM with multiple tracks, the tape symbols are denoted by T^k. The working remains same as that of standard Turing Machine.

Eg. $\delta(q, \{a, b, c\}) \rightarrow$
$(p, \{b, b, b\}, R)$



4, Subroutines : we know that subroutines are used in programming languages whenever a task has to be done repeatedly. The same facility can be used in TM and complicated TM's can be built using subroutines. A TM subroutine is a set of states that performs some pre-defined task. The TM subroutine has a start state and a state without any moves. This state which has no moves serves as the return state and passes the control to the state which calls the subroutine.

Design a TM to multiply two unary numbers separated by the delimiter 1.

Sol:- Let us assume we have two unary numbers x and y such that x has m number of 0's and y has n number of 0's separated by the delimiter 1 as shown below:

$$0^m 1 0^n$$

The product of x and y should be stored on the tape and the original numbers should not be destroyed. This can be visualized as shown below:

$$0^m 1 0^n 1 0^{m.n}$$

Eg: Let x = 00 and y = 0000 and are separated by delimiter 1. Assume y ends with 1 which can act as the delimiter for the input string which in turn is followed by infinite number of blanks (B's)

$$\underbrace{00}_{x}\ 1\ \underbrace{0000}_{y}\ 1\ BBB\ldots$$

The output should be of the form

$$\underbrace{00}_{x}\,1\,\underbrace{0000}_{y}\,1\,\underbrace{00000000}_{xy}\,B\,BB\ldots$$

Note:- It is clearly visible from the above figure that the unary number y is copied two times (equal to the number of 0's in the number x) in the result xy.

i.e output should be 00000000 // $4 \times 2 = 8$

So, to start with we think of how to copy the unary number y once so that we can call this TM (which is represented as a subroutine) repeatedly m times to achieve the result.

General Procedure: (Algorithm):

To start with let us assume a tape will have a string of the form $0^m 1 0^n BBBB\ldots$ which is divided into three groups as shown below:

$$\underbrace{0^m}_{p}\;1\;\underbrace{0^n}_{q}\;1\;\underbrace{BBBB\ldots}_{n}$$

(I group) (II group) (III group)

[Note that all B's will be replaced by the result $0^{m.n}$ later].

Step 1: Now, change a 0 in P to B.

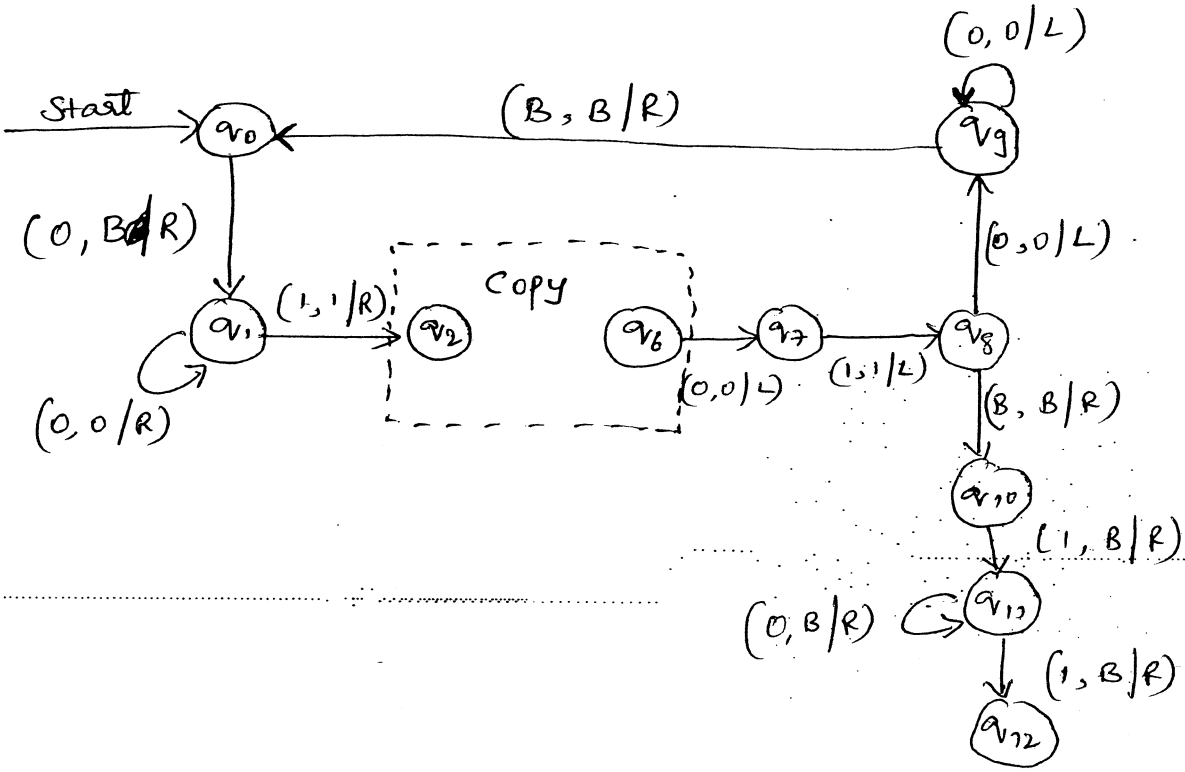
Step 2: Copy n numbers 0's from q to n by replacing n number of B's by n number of 0's.

Step 3 : Repeat above two steps till no 0's appears in P.

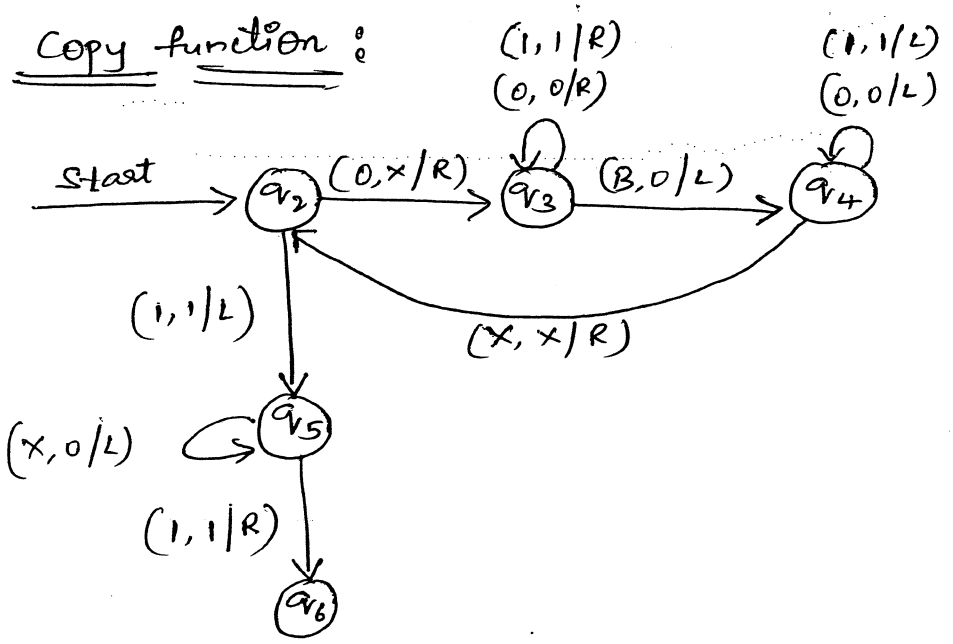
Note :- It is clear from second step that "we copy a group of n 0's from q by replacing n B's in q to n 0's whenever a 0 in P is changed to B".

When all 0's in P are changed to B's there will be definitely be mn number of 0's in q as shown below :

$$\underbrace{B^m}_{P} \; 1 \; \underbrace{0^n}_{q} \; 1 \; \underbrace{0^{mn}}_{q} BBBB$$



Copy function :



The TM to accept the given language is given by : $M = (Q, \Sigma, T, \delta, q_0, B, F)$

where,

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}, q_{12}\}$

$\Sigma = \{0, 1\}$

$T = \{0, 1, x, B\}$

$q_0 \in Q$ is the start state of machine

$B \in T$ is the blank symbol

$F = \phi$

δ is shown using the transition diagram.

Variants of Turing Machine :

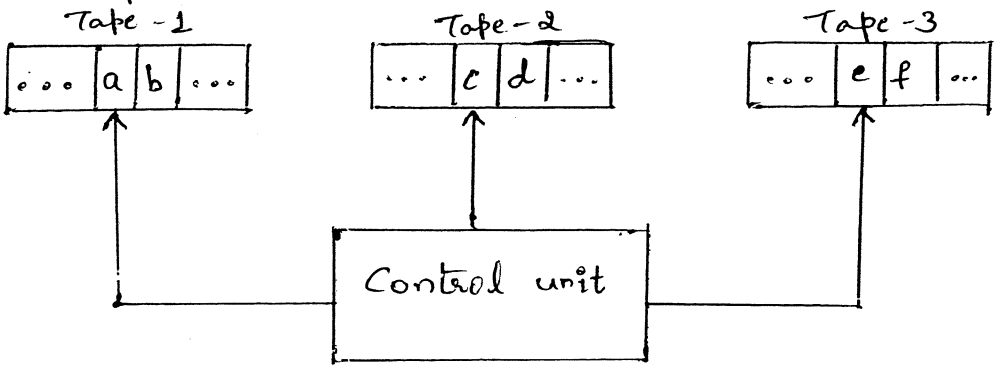
Two variants of TM are -

1. Multi-tape TM
2. Non-deterministic TM

1. Multi-tape TM :

A multi-tape turing machine is nothing but a standard turing machine with more number of tapes. Each tape is controlled independently with independent read-write head.

The pictorial representation of multi-tape TM with n tapes is shown below :



The various components of multi-tape TM are :

→ Finite control

→ Multiple R/W heads where each tape having its own symbols and R/W head.

Observe the following points from above TM.

→ Each tape is divided into cells which can hold any symbol from the given alphabet.

To start with, the TM should be in start state q_0.

→ If the R/w head is pointing to tape 1 moves towards right, the R/w head pointing to tape 2 and tape 3 may move towards right or left depending on the transition.

→ The move of the multi-tape TM depends on the current state and the scanned symbol by each of the tape heads.

Eg:- If number of tapes in TM is 3 and if there is a transition

$$\delta(q, a, b, c) = (p, x, y, z, L, R, S)$$

where q is the current state.

The transition can be interpreted as follows:

→ The TM in state q will be moved to state p only when the first R/w head points to a, the second R/w head points to b and third R/w head points to c.

→ The R/w head will be moved to left in the first case and right in the second case. But, the R/w head with respect to third tape will not be altered.

→ At the same time, the symbols a, b and c will be replaced by x, y and z.

The formal definition of multi-tape TM can be defined as follows:

Definition: The multi-tape TM is an n-tape machine

$$M = (Q, \Sigma, T, \delta, q_0, B, F)$$

Where,

$Q \rightarrow$ Set of finite states

$\Sigma \rightarrow$ Set of input alphabets

$T \rightarrow$ Set of tape symbols

P.T.O

$\delta \rightarrow$ Transition function from $Q \times T^n$ to $Q \times T^n \times \{L, R\}^n$

$q_0 \rightarrow$ Start state

$B \rightarrow$ Blank character

$F \rightarrow$ Set of final states

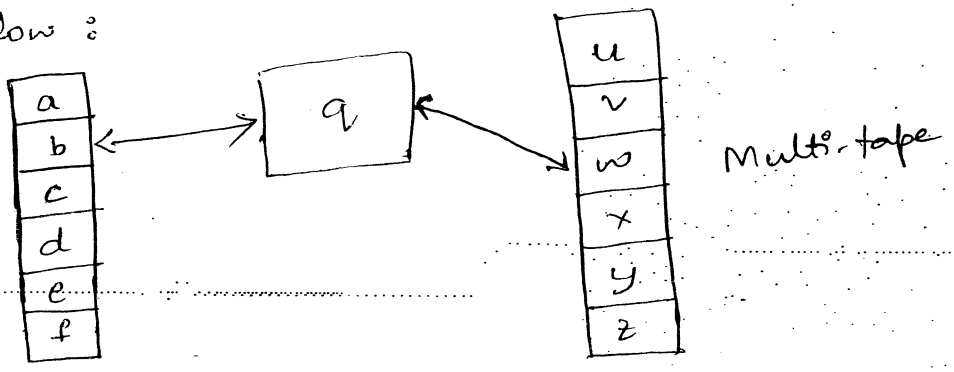
∴ It can be easily shown that the n-tape TM in fact is equivalent to the single tape Standard TM.

Theorem: Every language accepted by a multi-tape TM is accepted by standard Turing machine with single tape.

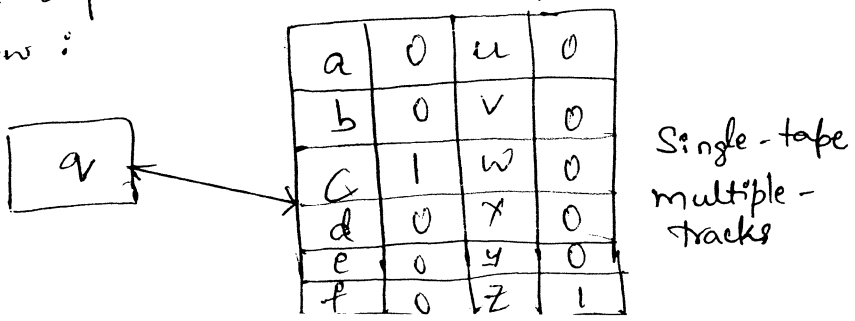
Proof:-

The Proof is by constructing a new machine. This can be shown by simulation.

For example, consider a TM with two tapes as shown below:



The above two-tapes TM can be simulated using single tape TM which has four tracks as shown below:



→ The first and third tracks consist of symbols from first and second tape respectively. The second and fourth track consists of the positions of the read/write head with respect to first and second tape respectively.

→ The value 1 indicates the position of the R/W head. It is clear from the above figure that, when the machine is on state q_1, first R/W head points to the symbol c, the second R/W head points to the symbol 7.

→ The machine can enter from one state to other, if and only if this transition is defined for TM with multi-tapes

→ So, whatever transitions have been applied for multi-tape TM, if we apply the same transitions for the new machine that we have constructed, then the two machines are equivalent.

2. Non-deterministic Turing machines :

The difference between non-deterministic TM and deterministic TM lies only in the definition of δ.

The formal definition of non-deterministic TM is shown below :

Definition : The non-deterministic TM (M), $M = (Q, \Sigma, T, \delta, q_0, B, F)$ where

$Q \longrightarrow$ set of finite states

$\Sigma \longrightarrow$ set of input alphabets

$T \longrightarrow$ set of tape symbols

P.T.O

δ is transition function which is a subset of $Q \times \Gamma \times \{L, R\}$

$q_0 \rightarrow$ Start State

$B \rightarrow$ Blank character

$F \rightarrow$ Set of final States

It is clear from the definition of δ that for each state q and tape symbol x, $\delta(q, x)$ is a set of triples :

$\{(q_1, x_1, D), (q_2, x_2, D), (q_3, x_3, D), \ldots (q_i, x_i, D)\}$

where,

$i \rightarrow$ finite integer

$D \rightarrow$ Direction indicating left / Right

* The machine can choose any of the triples as the next move.

Module-5

RASHMI.M
Asst. Prof.
Dept. of CSE,
RNSIT

RASHMI.M
Asst. Prof
Dept. of CSE
RNSIT

Module - 5

Decidability & Complexity

What is an algorithm ? Is the algorithm can be executed by Turing Machine ?

An algorithm is defined as step by step procedure to solve a given problem. The procedure is finite sequence of instructions that have to be executed to complete a task. The algorithm is terminated after finite number of steps for any input.

The church-Turing thesis states that any algorithm that can be carried out by a human or a computer, can also be carried out by a Turing machine. Thus, the TM is considered as an ideal theoretical model for an algorithm. The TM was considered as a tool to attach Hilbert's tenth problem. Hilbert's tenth problem is a challenge to provide a general algorithm which, for any given polynomial equation with integer coefficients and a finite number of unknowns can decide whether the equation has a solution with all unknowns taking integer values.

Decidability :

If an algorithm terminates eventually ; the TM also terminates.

Turing machine halts in foll. two situations :

→ when a TM reaches a final state, then it halts

→ when a TM reaches a state q and the input symbol is a and if the transition $\delta(q, a)$ is not defined, then it halts.

But, there are some T.M's that never halt on some inputs in any of the above situations. So, we have to make a distinction between the languages that are accepted by TM; and halts on all inputs and a TM that never halts on some input strings.

What is recursively enumerable language?

A language $L \subseteq \Sigma^*$ is recursively enumerable if and only if there exists a TM (M) such that $L = T(M)$ where $T(M)$ is the language accepted by Turing machine.

What is recursive language?

A language $L \subseteq \Sigma^*$ is recursive if and only if there exists a TM that satisfies the foll. two conditions:

→ If $w \in L$ then w is accepted by TM on reaching the final state and the machine halts.

→ If $w \notin L$, then w is rejected by TM without reaching the final state and the machine halts.

A machine may accept a language or it may reject a language. So, the output of the machine may be accept or reject. This problem is called membership problem. Formally, the membership problem can be stated as "Given a machine M and a string x, does M accept x?" The output will be yes/no. Given a language, the machine may have to identify whether the language is finite or infinite. All such problems with two answers yes/no, accept/reject, finite/infinite are called decision problems. For majority of the decision problems, we can design decision algorithms.

Define decidable language and undecidable language.

A decision problem or decision language with yes/no answer is decidable if and only if the corresponding language is recursive. In other words, a decidable language is a recursive language with yes/no answers. otherwise, it is undecidable language.

Decidable Languages

Prove that DFA is decidable language.

Theorem: DFA is decidable

Proof:- Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. we have to construct a TM M' that always halts and accepts $L(M)$. we know that DFA always ends in some state after reading the string w. NOW, we shall construct a TM M' that simulates DFA.

Let us define TM as shown below:

1. If w is the string to be scanned by DFSM M, then (M, w) is input to Turing machine M'.
2. Simulate M and input w in TM M'. Here, TM M' checks whether input (M, w) is valid input. If (M, w) is invalid, the TM M' rejects (M, w) and halts. If (M, w) is valid input, M' writes the initial state q_0 and leftmost input symbol of w. It updates the state using δ and reads the next symbol in w.
3. If simulation ends in an accepting state of M, then TM M' accepts w. If the simulation ends in non accepting state of M, then M' rejects w.

Thus, it is evident that TM M' accepts (M, w) if and only if w is accepted by DFA M.

If a language L is not accepted by a Turing machine, then the language is not recursively enumerable. One important problem which is not recursively enumerable that is unsolvable / undecidable decision problem is "Halting Problem".

Halting Problem:

Problem statement: Given a Turing machine M and an input string w with the initial configuration q_0, after some (or all) computations do the machine M halts?

In other words, if M is a turing machine, we have to identify whether (M, w) halts or does not halt when w is applied as the input. Given the description of an arbitrary turing machine M and the input string w, we are looking for a singe turing machine that will predict whether or not the computation of M applied to w will halt.

Note:- Alan Turing in 1990 proved that the halting problem is undecidable, and therefore cannot be solved.

A reduction technique is used to prove the undecidability of halting problem of a turing machine. using this technique, a problem A is reducible to problem B if a solution to the problem B can be used to solve the problem A. Thus,

→ If A is reducible to B and B is decidable, then, A is decidable

→ If A is reducible to B and B is undecidable, then A is undecidable.

Prove that halting problem of Turing machine is undecidable".

Proof: Let L_{Hlt} is decidable and get a contradiction. Let M_1 is a turing machine such that $L(M_1) = L_{Hlt}$ and Let M_1 halt eventually halts on all (M, w). Let us construct turing machine M_2 as follows:

1) For TM M_2, (M, w) is input
2) The TM M_1 acts on (M, w)
3) If TM M_1 rejects (M, w) then TM M_2 rejects (M, w).
4) If TM M_1 accepts (M, w), simulate TM M on the input string w until M halts
5) If Turing machine M has accepted w, Turing machine M_2 accepts (M, w); otherwise M_2 rejects (M, w).

Observe the following points to prove:

→ When M_1 accepts (M, w) (see step 4), the TM M halts on w.
→ In the first case (first alternative in step 5) M_2 accepts (M, w)
→ In the second case (second alternative in step 5) M_2 rejects (M, w).

So, it follows from definition of M_2 that the Turing machine M_2 halts eventually and hence, $L(M_2) = \{(M, w) :$ The turing machine accepts w $\}$ which is undecidable. This is a contradiction. So, the turing machine M halts on input w is undecidable.

Post Correspondence Problem :

Definition :- The post correspondence problem can be stated as follows. Given two sequences of n strings on some alphabet Σ say :

$A = w_1, w_2, \ldots, w_n$ and

$B = v_1, v_2, \ldots, v_n$

we say that there exists a post correspondence solution for pair (A, B) if there is a nonempty sequence of integers $i, j, \ldots, k$, such that

$$w_i w_j \ldots w_k = v_i v_j \ldots v_k$$

The post correspondence problem is to devise an algorithm that will inform us, for any (A, B) whether or not there exists a PC - solution.

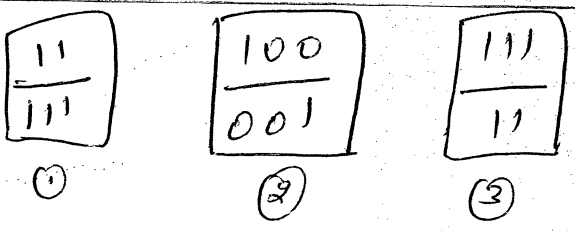
Example :-

Given two lists A, B. where A is w_1, w_2, w_3 and B is v_1, v_2, v_3. Find whether there exists a Post Correspondence Problem solution or not.

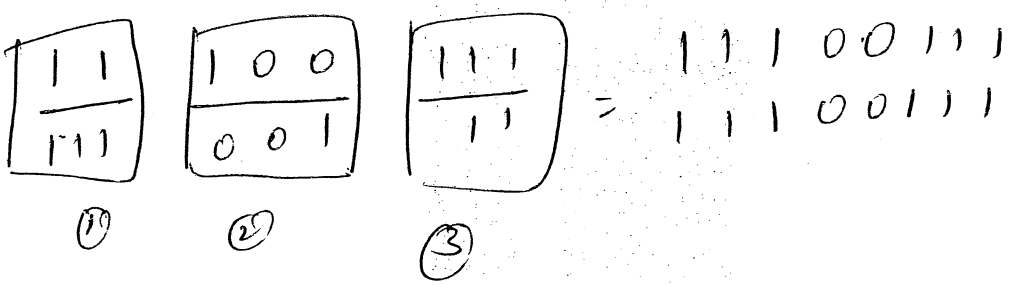
$w_1 = 11, \quad w_2 = 100, \quad w_3 = 111$

$v_1 = 111, \quad v_2 = 001, \quad v_3 = 11$

Sol:



For this case, there exists a PC-solution as shown below:



If we take

$w_1 = 00$, $w_2 = 001$, $w_3 = 1000$

$v_1 = 0$, $v_2 = 11$, $v_3 = 011$

there cannot be any PC-solution simply because any string composed of elements of A will be longer than the corresponding string from B.

Complexity:

When a problem or language is decidable, it means that the problem is computationally solvable in principle. But, it may require enormous amount of computation time and enormous amount of memory to solve it completely. In complexity theory, many scientists belive that $P \neq NP$. But, this is still open and any one can challenge this relation.

Observe the following points:

→ P stands for polynomial time. This class of Problems can be solved by a deterministic algorithm in a polynomial time.

→ NP stands for non-deterministic polynomial time : This class of problems can be solved by a non-deterministic algorithm in a polynomial time.

Growth Rate of Functions :

Given two algorithms to solve the same problem, it is necessary to compare those algorithms to find out which is more efficient. we normally use big-oh notation which is defined as shown below :

Definition :- Let $f(n)$ be the time efficiency of an algorithm. The function $f(n)$ is said to be $O(g(n))$ denoted by

$$f(n) = O(g(n))$$

if and only if these exists a posifive constant c and positive integer n_0 satisfying the constraint

$$f(n) \leq c * g(n) \text{ for all } n \geq n_0$$

Classes of P and NP :

What is the time complexity of a Turing machine which accepts a string w of length n ?

Definition : Given an input string w of length n, the time complexity of a Turing machine is given by $T(n)$. It indicates that the TM halts after making maximum of $T(n)$ moves. In this case, TM halts and it is called Deterministic TM.

Define class P with respect to Turing machine.

Let $L(n)$ is the language accepted by Turing machine with time complexity $T(n)$. The language L is in class P if there exists some Polynomial $P(n)$ such that $L = L(M)$ for some deterministic turing machine M with time complexity $T(n)$.

Obtain the time complexity of a Turing machine which accepts $L = \{a^n b^n \mid n \geq 1\}$

Sol:- Step1:- When we construct the turing machine for the above languages, every leftmost a is replaced by X and every leftmost b is replaced by Y. So, the total numbers of moves required $= 2n$.

Step2:- In step2 of constructing a machine, we see that the above procedure is repeated till all n number of a's are replaced by X's. So, the above procedure has to be repeated n times.

Step3:- So, total numbers of moves $= 2n(n) = 2n^2$

$\Rightarrow n^2$ (By neglecting the constants).

So, the time complexity is given by $O(n^2)$.

Define Class NP with respect to Turing machine

Let $L(n)$ is the language accepted by Turing machine with time complexity $T(n)$. The language L is in class NP if there exists some Polynomial $P(n)$ such that $L = L(M)$ for some non-deterministic Turing machine M with time complexity $T(n)$.

Quantum Computation:

Quantum Computer ?

The Computers that are built based on the principles of quantum mechanics are called quantum Computers.

Quantum computers are different from binary digital electronic computers based on transistors. The data in digital computers is represented using binary states 0 and 1 whereas the data in quantum computers is represented mathematically as shown below:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

The qubit can be explained as shown below:

→ Two possible states are represented as $|0\rangle$ and $|1\rangle$

→ unlike classical bit 0 or 1, a qubit can have infinite number of states other than $|0\rangle$ and $|1\rangle$ and can be represented as

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

where $|\alpha|^2 + |\beta|^2 = 1$

→ In qubits 0 and 1 are called computational basis states, $|\psi\rangle$ is called a superposition and $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ is called quantum state.

→ In digital computer, data can be represented as 0 or 1. But, it is not possible to determine the quantum state on observation. In quantum computers we get a state $|0\rangle$ with probability $|\alpha|^2$ and state $|1\rangle$ with probability $|\beta|^2$

→ Multiple qubits can be defined in a similar manner. For example, a two-qubit system has four computational basis states, $|00\rangle$, $|01\rangle$, $|10\rangle$, $|11\rangle$ and quantum states are represented as:

$$|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$$

where $$|\alpha_{00}|^2 + |\alpha_{01}|^2 + |\alpha_{10}|^2 + |\alpha_{11}|^2 = 1$$

→ Normal NOT gate interchanges 0 and 1 whereas in case of qubit NOT gate, $\alpha|0\rangle + \beta|1\rangle$ is changed to $\alpha|1\rangle + \beta|0\rangle$

Thus, a quantum computer can be defined as: A system built from number of quantum circuits containing wires and elementary quantum gates such as OR, NOT, AND, NOR etc, to carry out manipulation of quantum information.

Church Turing Hypothesis / (church - Turing thesis)

Various formal models of computations such as Recursive functions and Post systems were established by three prominent persons - A. church, S.C. klene and E. Post.

A function is called primitive recursive if and only if it can be constructed from the basic functions by successive composition and primitive recursion. A Post system is similar to ~~understand~~ unrestricted grammar consisting of an alphabet and some production rules by which successive strings can be derived.

In addition to recursive functions and Post systems, many other formal computations models have been proposed. On examination it was found that though the computational models looked quite different, they expressed the same thing. This observation was formalized in churche thesis which is stated as follows:

Any "effective computation" or "any algorithmic" procedure that can be carried out by a human being & a team of human beings or a computer, can be carried out by some Turing machine.

In other words, there is an effective procedure to solve a decision problem P if & only if there is a turing machine that answers yes on input $w \in P$ and no for $w \notin P$.

This theory maintains that all the models of computations those are proposed and yet to be proposed are equivalent in their power to recognize languages or compute functions. This thesis predicts that it is unable to construct models of computation more powerful than the existing ones

The above statement is known as "church's thesis" named after the logician A. church. Since the church's thesis is closely related to turing's thesis which states that we cannot go beyond Turing machines or their equivalent, it is also called church-Turing thesis.

Since there is no precise definition for "effective computation" or there is no precise statement. So, this statement is not proved at the same time it has been not been disapproved. Even though it is simply stated and not proved, now majority of scientists have accumulated enough evidence over the years that has caused Church's thesis to be generally accepted.

Linear Bounded Automata (LBA)

We know that the power of Turing machine cannot be extended beyond the power of Standard Turing machine by complicating the TM with multiple tapes or by using multi dimensional tapes. Instead, the power of TM can be restricted by restricting the tape usage. An example of this is the PDA which can be considered as a non-deterministic TM. In PDA, it can be assumed that the tape is used like a stack. We can also assume finite portion of the tape as input that leads to finite

automaton. with slight alteration in usage of the tape, let us restrict the workspace on the tape using two delimiters '[' and ']'. The given string has to be enclosed between these two limiters. so, longer the string, longer the workspace. This leads to another class of machine called Linear Bounded Automaton (LBA). So, LBA is a TM which is bounded based on the length of the input string. Thus, using TM and by restricting the usage of tape, we can obtain LBA, we construct PDA and finite automaton. It is clear from all these types of machines that TM is superset of all these machines.

Define Linear bounded Automaton (LBA)

The LBA is a TM

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

where

⇒ $Q \rightarrow$ set of finite states

$\Sigma \rightarrow$ set of input alphabets which also has two special symbols '[' and ']'

$\Gamma \rightarrow$ set of tape symbols

$\delta \rightarrow$ subset of $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$ with two more transitions of the form :

$$\delta(q_i, [) = (q_j, [, R)$$

$$\delta(q_i,]) = (q_j,], L)$$

forcing the read/write head to be within the boundaries '[' and ']'.

q_0 → Start State

B → Blank character

F → Final States

Observe the following Points :

→ Σ contains two special symbols '[' and ']'

→ The symbol '[' is called the left end marker which is entered in the leftmost cell of the input tape preventing the R/W head from getting off the left-end of the tape.

→ The symbol ']' is called the right end marker which is entered in the rightmost cell of the input tape preventing the R/W head from getting off the right-end of the tape.

→ Both the end-markers should not appear on any other cell within the input tape.

→ R/W head should not print any other symbol over both end-markers.

It is clear from the definition that the R/W head cannot go out of the boundaries specified as '[' and ']'. Now, the string can be accepted by LBA only if there is a sequence of moves such that

$$q_0[w] \vdash^* [x q_f z]$$

for some $q_f \in F$ and $x, z \in \Gamma^*$

THEOREM: If $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots a_1 n^1 + a_0$ with $a_k > 0$ then $p(n) = O(n^k)$.

SOLUTION:

It is given that $f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n^1 + a_0$

Each term in the summation is of the form $a_i n^i$. Since, n is non-negative, a particular term will be negative only if $a_i < 0$. So,

$$|f(n)| = |a_k n^k + a_{k-1} n^{k-1} + \cdots a_1 n^1 + a_0|$$

$$\leq |a_k| n^k + |a_{k-1}| n^{k-1} + \cdots |a_1| n^1 + |a_0|$$

$$= n^k \left[|a_k| + \frac{|a_{k-1}|}{n} + \cdots \frac{|a_1|}{n^{k-1}} + \frac{|a_0|}{n^k} \right]$$

$$\leq n^k \left(|a_k| + |a_{k-1}| + \cdots + |a_1| + |a_0| \right)$$

$$\leq c * n^k \text{ where } c = (|a_k| + |a_{k-1}| + \cdots |a_1| + |a_0|)$$

i.e, $f(n) \leq c * n^k$ where $c = (|a_k| + |a_{k-1}| + \cdots |a_1| + |a_0|)$ for $n > 1$

$\Downarrow$

$f(n) \leq c * g(n)$ for $n > n_0$ where $g(n) = n^k$ and $n_0 = 1$.

by definition, we can write.

$$f(n) = O(g(n)) \text{ or } f(n) \in O(g(n))$$

i.e $\boxed{f(n) = O(n^k) \text{ or } f(n) \in O(n^k)}$