Theory of Computations

Alphabet: finite, nonempty set Σ of symbols, called the alphabet.

Strings which are finite sequences of symbols from the alphabet

Concatenation of two strings w and υ is the string obtained by appending the symbols of υ to the right end of w, that is, if

$w = a_1 a_2 \cdots a_n$ And $v = b_1 b_2 \cdots b_m$, then the concatenation of w and υ, denoted by **wυ**, is

$$wv = a_1 a_2 \cdots a_n b_1 b_2 \cdots b_m$$

Reverse of a string is obtained by writing the symbols in reverse order.

If $w = a_1 a_2 \cdots a_n$ then $w^R = a_n \cdots a_2 a_1$

tring.

$$|\lambda| = 0,$$
$$\lambda w = w\lambda = w$$

Empty string, which is a string with **no symbols at**

Length of a string w, denoted by |w|, is the number of symbols in the s **all**. It will be denoted by **λ**.

$|u^n| = n\,|u|$ for all strings u and all n.

Substring string of consecutive symbols in some w. If w = uv

Prefix if w = abbab then prefix = {λ, a, ab, abb, abba, abbab}

Suffix: if w = abbab then suffix = { λ, b, ab, bab, bbab, abbab}

If **w is a string**, then w^n stands for the string obtained by **repeating w, n times.** As a special case defined as $w^0 = \lambda$.

If Σ is an alphabet, then we use Σ* to denote the set of strings obtained **by concatenating zero or more symbols** from Σ. The set Σ* always contains λ. $\Sigma^+ = \Sigma^* - \{\lambda\}$

While Σ is **finite** by assumption, **Σ* and Σ^+ are always infinite** since there is no limit on the length of the strings in these sets.
A language is defined very generally as a subset of Σ*. A string in a language L will be called a **sentence** of L.

The set $\{a, aa, aab\}$ is a language on Σ. Because it has a finite number of sentences, we call it a **finite language**.

The **complement of a language** is defined $\overline{L} = \Sigma^* - L$

The **reverse of a language** is the set of all string reversals, that is, $L^R = \{w^R : w \in L\}$

$$a^R = a,$$
$$(wa)^R = aw^R$$

For all a ∈ Σ, w ∈ Σ*.

1. $(uv)^R = v^R u^R$ For all u, υ ∈ Σ+.
2. $(w^R)^R = w$ for all $w \in \sum^*$

The **concatenation of two languages** L1 and L2 is the set of all strings obtained by concatenating **any element of L1 with any element of L2**; specifically $L_1L_2 = \{xy : x \in L_1, y \in L_2\}$
For every language L., We define **L^n** as **L concatenated with itself n times**, with the special cases

$$L^0 = \{\lambda\}$$

And

$$L^1 = L$$

We define the **star-closure** of a language as

$$L^* = L^0 \cup L^1 \cup L^2 \cdots$$

And the **positive closure** as

$$L^+ = L^1 \cup L^2 \cdots .$$

Example: If

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

Then

$L^2 = \{a^n b^n a^m b^m : n \geq 0, m \geq 0\}$ n and m are unrelated.

The of L is

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

Grammars

A grammar G is defined as a **quadruple G = (V, T, S, P)**

Where V is a **finite** set of objects called **variables**,
T is a **finite** set of objects called **terminal symbols**,
S ∈ V is a special symbol called the **start variable**,
P is a **finite** set of productions.

$w \Rightarrow z$ We say that w derives z or that z is derived from w.

$w_1 \overset{*}{\Rightarrow} w_n$ The * indicates that an **unspecified number of steps (including zero)** can be taken to derive wn from w1.
The set of all **such terminal strings is the language defined or generated by the grammar**.

Let G = (V, T, S, P) be a grammar. Then the set

$$L(G) = \left\{ w \in T^* : S \overset{*}{\Rightarrow} w \right\}$$

is the language generated by G.

Sentential forms

If w ∈ L (G), then the sequence. $S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n \Rightarrow w$ is a **derivation** of the sentence w.
The strings **S, w1, w2… wn**, which contain variables as well as terminals, are called **sentential forms** of the derivation.

Two grammars ***G*1 and *G*2 are equivalent** if they generate the same language

$$L(G_1) = L(G_2)$$

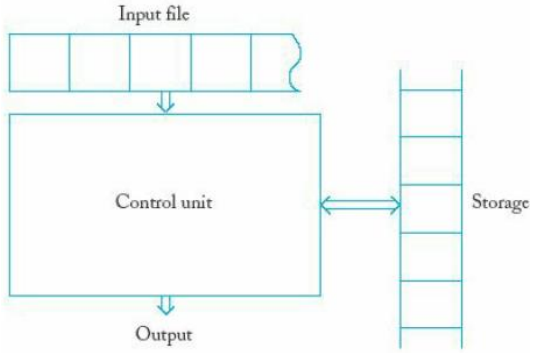
Note:- A given language has many grammars that generate it.

Automata

An automaton is an abstract model of a digital computer. It will be assumed that the input is a string over a given alphabet, written on an **input file**, **which the automaton can read but not change**.
The input file is divided into **cells**, each of which can hold one symbol. The input mechanism can read the input file from left to right, one symbol at a time.
The input mechanism can also detect the **end of the input** string (by sensing an end-of-file condition)

The automaton has a **control unit**, **which can be "in" any one of a finite number of internal states**, and which can change state in some defined manner.

Transition function it gives the next state in terms of the **current state**, the **current input symbol**, and the information currently in the **temporary storage**.



Deterministic Automata
A deterministic automaton is one in which each move is uniquely determined by the current configuration. If we know the internal state, the input, and the contents of the temporary storage, we can predict the future behavior of the automaton exactly.

Nondeterministic automata

Nondeterministic automaton may have several possible moves, so we can only predict a set of possible actions.
Note

a. An automaton whose output response is limited to a simple "yes" or "no" is called an **accepter**.
b. Automaton, capable of producing strings of symbols as output, is called a **transducer**.

Properties:-

1. (L*)' != (L')* as epsilon doesn't belong to LHS language.
2. $(L_1L_2)^R = L_2^R L_1^R$ for all languages L1 and L2.
3. (L*)* = L*
4. $(L_1 \cup L_2)^R = L_1^R \cup L_2^R$ for all language L1 and L2.
5. $(L^R)^* = (L^*)^R$ for all languages L.
6. Let L be any language on non-empty alphabet, both L and L' can't be finite as **L U L' = Σ***
7. Take Σ = { a, b}, and let na(w) and nb(w) denote the number of a's and b's in the string w, respectively. Then the grammar G with productions:

$$S \to SS,$$
$$S \to \lambda,$$
$$S \to aSb,$$
$$S \to bSa$$

$$L = \{w : n_a(w) = n_b(w)\}$$

(b) $L = \{w : n_a(w) > n_b(w)\}$.

*(c) $L = \{w : n_a(w) = 2n_b(w)\}$.

Solution

(b) $S \to aS \,|S_1 S|\, aS_1$.

where S_1 derives the language in Example 1.13.

(c) $S \to aSbSa \,|aaSb|\, bSaa \,|SS|\, \lambda$.

8. **Elimination of null productions :**

S → aABbCD	S → aABbCD | aABbC | aAbCD | aAbC | aBbCD | aBbC | abCD | abC
A → ASd | ε	A → ASd | Sd
B → SAc | hC | ε	B → SAc | hC | Sc
C → Sf | Cg	C → Sf | Cg
D → BD | ε	D → BD | B

9. Φ* = ε, $L_1\Phi = \Phi$

Finite Automata

A **deterministic finite accepter** or dfa is defined by the quintuple **M = (Q, Σ, δ, q0, F)**,

Where Q is a **finite** set of **internal states**,
Σ is a **finite** set of symbols called the input alphabet,
δ :Q × Σ → Q is a **total function** called the **transition function**,
q0 ∈ Q is the **initial** state,
F ⊆Q is a set of **final** states.

Extended transition function

$$\delta^* : Q \times \Sigma^* \rightarrow Q.$$

The second argument of δ* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string.

The language accepted by a DFA M = (Q, Σ, δ, q0, F) is the set of all strings on Σ accepted by M.
In formal notation,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}$$

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\}$$

Nondeterministic Finite Accepters

Nondeterminism means **a choice of moves** for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves.

A **nondeterministic finite accepter** or nfa is defined by the **quintuple**:

M=(Q, Σ, δ, q0, F)

Where Q, Σ, q0, F are defined as for deterministic finite accepters, but

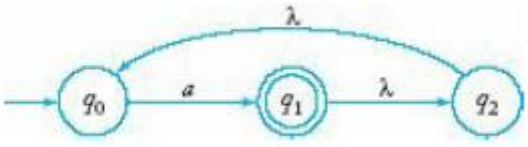
$$\delta : Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q$$

In a nondeterministic accepter, **the range of δ is in the powerset 2^Q, so that its value is not a single element of Q but a subset of it**. This subset defines the set of all possible states that can be reached by the transition. **Let L be a language accepted by a DFA with q states then there exists NFA with q or less than q states which accepts L.**

$$\delta(q_1, a) = \{q_0, q_2\}$$

1. Either q0 or q2 could be the next state of the nfa.
2. We allow λ as the second argument of δ. This means that the nfa can make a transition without consuming an input symbol.
3. Although we still assume that the input mechanism can only travel to the right, **it is possible that it is stationary on some moves**.
4. Finally, in an nfa, **the set δ (qi,a) may be empty**, meaning that **there is no transition defined** for this specific situation (Dead configuration). **Note dead configuration is diferent from dead state.**

A string is accepted by an nfa **if there is some sequence of possible moves that will put the machine in a final state at the end of the string**. A string is **rejected** (that is, not accepted) only i**f there is no possible sequence of moves** by which a final state can be reached.



We see that δ*(q2, λ) contains q0. Also, **since any state can be reached from itself by making no move**, thus

$$\delta^*(q_2, \lambda) = \{q_0, q_2\}$$
$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}.$$

The language L accepted by an nfa **M = (Q, Σ, δ, q0, F)** is defined as the set of all strings accepted in the above sense. Formally,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \varnothing\}$$

The language consists of all strings w for which there is a walk labeled w from the initial vertex of the transition graph to some final vertex.

Reduction of the Number of States in Finite Automata

For every input from a particular state there always be some transition in DFA but not in NFA.

For a given language, there are many DFA's that accept it.

1. Two states p and q of a dfa are called **indistinguishable** or **equivalent** if

$$\delta^*(p, w) \in F \text{ implies } \delta^*(q, w) \in F,$$

OR

$$\delta^*(p, w) \notin F \text{ implies } \delta^*(q, w) \notin F,$$

On same input w, states p and both are either going to final state or non-final state will be called **equivalent states.**

2. For all w ∈ Σ*. If, on the other hand, there exists some string w ∈ Σ* such that

$$\delta^*(p, w) \in F \text{ and } \delta^*(q, w) \notin F,$$

OR vice versa then the states p and q are said to be **distinguishable** by a string w.

1. First remove **unreachable** states from initial state
2. Then, merge **equivalent** states.
3. If there are more than one final states then check for final states also if they are equivalent or not

Regular Languages and Regular Grammars

Let Σ be a given alphabet. Then

1. Ø, λ and a ∈ Σ are all regular expressions. These are called **primitive regular expressions**.
2. If r1 and r2 are regular expressions, so are r1+ r2, r1.r2, r1*, and (r1).
3. For every a ∈ Σ, a is a regular expression denoting {a}.

$$L_1 : \{ww^r\ c\ w \mid w, c \in (a+b)^*, w^r = \text{reverse of } w\}$$

Above language is regular by setting C = (a+b)* and w=E we have L = (a+b)* which covers all langauge.
Fact : In case of NFA, L(M') language accepted by complement of machine and complement of language accepted by machine (L(M))' is not equivalent. You can verify through top state diagram.

Special Cases of question :

1) No. of final state in any dfa accepting languages like $L = \{a^n : n \bmod 7 = 0\} \cup \{a^n : n \bmod 15 = 0\}$.
(15*7/7)+ (15*7/15) - 1 = 21

2) For any integer m = 2k x d where d is odd, the minimal number of states in the DFA to recoginze divisivility by m is k+d.

3) You cannot combine final state with other non final state in merging the state. But in solving minimum dfa don't apply merge state procedure first check if you find any pattern in strings. don't forget to delete state that don't participate in strings formation.

Complement a DFA:

In a given DFA,

1. Convert final states into non-final states, and
2. Convert non-final states into final states.
3. Don't change initial state

This DFA will accept Complement of the language accepted by the original DFA.

Reversal of DFA

L = Language start with a.
L^r = Language ends with a.

In the given DFA,

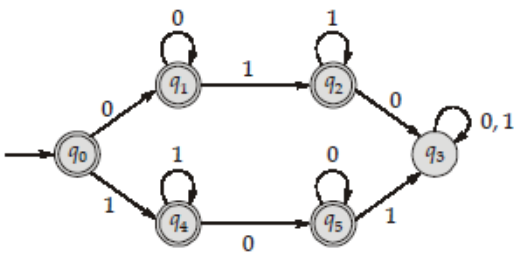
1. Make the final states as initial state.
2. Make the initial state as final state.
3. Reverse all the transition from q0 → q1 to q1→q0 for any two states in DFA.
4. Self-loops are unchanged.
5. Reversal of a DFA may result into a DFA or an NFA.
6. If there are multiple initial state into resulted DFA (or NFA) take an initial state and add all initial state with epsilon transitions.

Number of equivalence classes :

Consider the regular expression R = 0*1* + 1*0*
The number of equivalence classes of Σ* to represent a language which is equivalent to R is

Minimal DFA for given regular expression is

$$= (\in + 00^*)(\in + 11^*) + (\in + 11^*)(\in + 00^*)$$
$$= \in + 00^* + 11^* + 00^* 11^* + 11^* 00^*$$



Number of equivalence classes = Number of states is minimized DFA = 6.

Types of Grammars:

1. **Regular Grammar (Type – 3 grammar) (Regular Language)**

   a. **Right Linear Grammar** a grammar G =(V, T, S, P) is said to be **right-linear** if all productions are of the form.

   A → xB,

   A → x,

   where $A, B \in V$, and $x \in T^*$

   b. **Left Linear Grammar** a grammar is said to be **left-linear** if all productions are of the form.

   A → Bx,

   A → x

   where $A, B \in V$, and $x \in T^*$

   Conversion from Left Linear grammar to Right Linear grammar

   Given any left-linear grammar G with productions of the form

   A → Bx,

   A → x

   We construct from it a right-linear grammar G^ by replacing every such production of G with

   A->x^RB

   A-> x^R respectively. It will give the reverse of the language.

   L(G) = (L(G^))^R

2. **Context Free Grammar (Type – 2 grammar) (Context Free Language)**

   A grammar G = (V, T, S, P) is said to be **context-free** if all productions in P have the form

   A → x where $A \in V$ and $x \in (V \cup T)^*$

3. **Context Sensitive Grammar (Type – 1 grammar) (Context Sensitive Language)**

   A grammar G = (V, T, S, P) is said to be **context-sensitive** if all productions are of the form

   X → y and $|x| \leq |y|$

   where $x, y \in (V \cup T)^+$ and

4. **Unrestricted Grammar (Type – 0 grammar) (RE Language)**

   A grammar G =(V, T, S, P) is called **unrestricted** if all the productions are of the form

   u → v

   where u is in $(V \cup T)^+$ and v is in $(V \cup T)^*$

   In an unrestricted grammar, essentially no conditions are imposed on the productions.

Context-Free Languages

Simple grammar or S-grammar

A context-free grammar G = (V, T, S, P) is said to be a **simple grammar** or **s-grammar** if all its productions are of the form.

A → ax,

Where **A ∈ V, a ∈ T, x ∈ V***, and **any pair (A, a) occurs at most once** in P.

a. The grammar S → aS | bSS | c
   Is an s-grammar.
b. The grammar
   S-> aS | bSS | aSS | c
   Is not an s-grammar because the pair (S,a) occurs in the two productions S→aS and S→aSS.

Note: - If G is an **s-grammar**, **any string w in L(G) can be parsed with efforts proportional to |w|**. Each step produces one terminal symbol and hence the whole process must be completed in no more than |w| steps.

Ambiguity in Grammars and Languages

1. A context-free grammar G is said to be **ambiguous** if there exists some w ∈ L(G) that has **at least two distinct derivation trees**.
2. Alternatively, ambiguity implies the existence of **two or more leftmost or rightmost derivations**.

Inherently ambiguous

If L is a context-free language for which there exists an **unambiguous grammar**, then L is said to be **unambiguous**. If every grammar that generates L is ambiguous, then the language is called **inherently ambiguous**.

Simplification of Context-Free Grammars and Normal Forms

1. **Removing Useless Productions**
   Remove productions from a grammar that can never take part in any derivation.
   Let G = (V, T, S, P) be a **context-free grammar**. A **variable A ∈ V** is said to be useful if and only if there is **at least one w** ∈ L (G) such that

$$S \overset{*}{\Rightarrow} xAy \overset{*}{\Rightarrow} w,$$

   With x, y in (V U T)*. A variable is useful if and only if it occurs in at least one derivation.

   A variable that is not useful is called **useless**. **A production is useless if it involves any useless variable**.

   There are two reasons why a variable is useless
   a. Either it can't be reached from start variable S.
   b. It cannot derive a terminal string.

1. First remove those variables which can't generate at least a terminal string.
2. Second. Remove those variables those can't be reached from start variable. Draw a dependency graph and remove them.
3. Remove those productions also where useless appear.

Let $G = (V, T, S, P)$ be a context-free grammar. Then there exists an equivalent grammar $\hat{G} = (\hat{V}, \hat{T}, S, \hat{P})$ that does not contain any useless variables or productions.

Removing λ-Productions

Any production of a context-free grammar of the form

$$A \rightarrow \lambda$$

Is called a **λ-production**.

Any variable A for which the derivation

$$A \overset{*}{\Rightarrow} \lambda$$

Is possible is called **nullable**.

Note: A grammar may generate a language not containing λ, yet have some λ-productions or nullable variables. In such cases, **the λ-productions can be removed**.

1. Let G be any context-free grammar with λ not in L (G). Then there exists an equivalent grammar having no λ-productions.
2. If a language contains empty string then we can't remove **λ-productions** from the grammar.

Chomsky Normal Form

DEFINITION 2.8

A context-free grammar is in ***Chomsky normal form*** if every rule is of the form

$$A \rightarrow BC$$
$$A \rightarrow a$$

where a is any terminal and A, B, and C are any variables—except that B and C may not be the start variable. In addition we permit the rule $S \rightarrow \epsilon$, where S is the start variable.

Note:- we can convert any grammar into CNF

1. If the start symbol S occurs on some right side, we create a new start variable S' and add a new productions S' → S.
2. Then, we eliminate all **λ-productions** of the form **A → λ**
3. We also eliminate all **unit rules** of the form A → B.
4. If there is production S->S remove it as it's trivial production.
5. We need (2n-1) productions to generate n length string.
6. Every context free grammer in CNF that generate L has more than logk variabes. Where K is the length of longest string in L.

Any context-free grammar $G = (V, T, S, P)$ with $\lambda \notin L(G)$ has an equivalent grammar $\widehat{G} = \left(\widehat{V}, \widehat{T}, S, \widehat{P}\right)$ in Chomsky normal form.

Greibach Normal Form

We put restrictions not on the length of the right sides of a production, but on the positions in which terminals and variables can appear.

A context-free grammar is said to be in **Greibach normal form** if all productions have the form

A → ax,

Where **a ∈ T** and **x ∈ V***

Example-
The grammar

$$S \rightarrow AB,$$
$$A \rightarrow aA \,|bB|\, b,$$
$$B \rightarrow b$$

From CFG to GNF

$$S \rightarrow aAB \,|bBB|\, bB,$$
$$A \rightarrow aA \,|bB|\, b,$$
$$B \rightarrow b,$$

Note:-

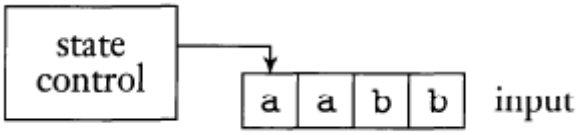
For every context-free grammar G with $\lambda \notin L(G)$, there exists an equivalent grammar $\widehat{G}$ in Greibach normal form.

A Membership Algorithm for Context-Free Grammars

CYK Membership and parsing algorithms for context-free grammars exist that require approximately **|w|^3** steps to parse a string w.
CYK algorithm works only if the grammar is in CNF and succeeds by breaking one problem into a sequence of smaller ones.

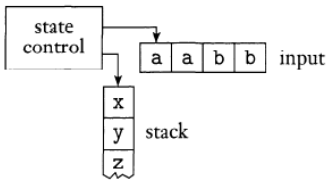
Push Down Automata

Pushdown automata, are like non-deterministic finite automata but have an extra component called a **stack.**
Pushdown automata are equivalent in power to context-free grammars.



Schematic of finite automata

With the addition of a stack component we obtain a schematic representation of a PDA



Schematic of PDA

A pushdown automaton (PDA) can write symbols on the stack and read them back later. Writing a symbol "pushes down" all the other symbols on the stack. At any time the symbol on the top of the stack can be read and removed. The remaining symbols then move back up. Writing a symbol on the stack is often referred to as ***pushing*** the symbol, and removing a symbol is referred to as ***popping*** it. Note that all access to the stack, for both reading and writing, may be done only at the top. In other words a stack is a "last in, first out" storage device. If certain information is written on the stack and additional information is written afterward, the earlier information becomes inaccessible until the later information is removed.

A nondeterministic pushdown accepter (npda) is defined by the septuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$$

Where
Q is a finite set of internal states of the **control unit**,
Σ is the **input alphabet**,
Γ is a finite set of symbols called the **stack alphabet**
δ: Q × (Σ ∪ {λ}) × Γ → set of finite subsets of Q × Γ* is the transition function.
q0 ∈ Q is the initial state of the control unit,
z ∈ Γ is the **stack start symbol**,
F ⊆ Q is the set of final states.

Transition function: δ: Q × (Σ ∪ {λ}) × Γ → set of finite subsets of Q × Γ*

1. The arguments of δ are the **current state of the control unit**, the **current input symbol**, and the **current symbol on top of the stack**.
2. The **result is a set of pairs (q, x)**, where **q is the next state of the control unit** and **x is a string** that is put on top of the stack in place of the single symbol there before.

3. Note that the second argument of δ **may be λ**, indicating that a move that does not consume an input symbol is possible. **We will call such a move a λ-transition**.
4. Note also that δ is defined so that it needs a stack symbol; **no move is possible if the stack is empty**.
5. Finally, the requirement that the elements of the range of δ be a **finite subset is necessary because Q × Γ* is an infinite set and therefore has infinite subsets**.
6. While an npda may have several choices for its moves, this **choice must be restricted to a finite set of possibilities**.

Suppose the set of transition rules of an npda contains

$$\delta(q_1, a, b) = \{(q_2, cd), (q_3, \lambda)\}.$$

If at any time the control unit is in state q1, the input symbol read is a, and the symbol on top of the stack is b, **then one of two things can happen**: (1) the control unit goes into state q2 and the string cd replaces b on top of the stack, or (2) the control unit goes into state q3 with the symbol b removed from the top of the stack. In our notation **we assume that the insertion of a string into a stack is done symbol by symbol, starting at the right end of the string**.

The Language Accepted by a Pushdown Automaton

$$L(M) = \left\{ w \in \Sigma^* : (q_0, w, z) \overset{*}{\vdash}_M (p, \lambda, u), p \in F, u \in \Gamma^* \right\}$$

The language accepted by M is the **set of all strings that can put M into a final state at the end of the string**.
The final stack content u is irrelevant to this definition of acceptance.

Deterministic Pushdown Automata and Deterministic Context-Free Languages

A **deterministic pushdown accepter** (**dpda**) is a pushdown automaton that never has a choice in its Move.

$$M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$$

Where
Q is a finite set of internal states of the **control unit**,
Σ is the **input alphabet**,
Γ is a finite set of symbols called the **stack alphabet**
It is subject to the restrictions that, for every q ∈ Q, a ∈ Σ ∪ {λ} and b ∈ Γ,
1. δ(q, a, b) **contains at most one element**.
2. If δ (q, λ, b**) is not empty**, then δ **(q, c, b) must be empty for every c ∈ Σ**.

- The first of these conditions simply requires that for any given input symbol and any stack top, **at most one move can be made**.
- The second condition is that when **a λ-move is possible for some configuration, no input-consuming alternative is available**.

Note:-
1. We retain λ-transitions in DPDA also.
2. λ-transitions **does not automatically imply nondeterminism**. Also, some transitions of a dpda

may be to the empty set, that is, undefined, so there may be **dead configurations**.

3. Only criterion for determinism is that at all times at most one possible move exists.

A language L is said to be a deterministic context-free language if and only if there exists a dpda M such that L = L (M).

A Pumping Lemma for Context-Free Languages

If A is a context-free language, then there is a **number p** (the pumping length) where, **if s is any string in A of the length at least p**, then **s may be divided into five pieces s = uvxyz** the conditions

1. for each $i \geq 0$, $uv^ixy^iz \in A$
2. $|vy| > 0$, and
3. $|vxy| \leq p$

1. Case 2: When S is being divided into uvxyz, condition 2 says that either v or y is not empty string**.**
2. Case 3: the pieces v, x and y together have length at most p.

Note: -

1. A DPDA with acceptance by **EMPTY STACK** is proper subset of the languages accepted by a DPDA with **final state.**
2. For each DCFL which satisfied prefix property, can be accepted by a DPDA with empty stack.

A Pumping Lemma for Regular Languages

Given an infinite regular language L, there exists an integer **p > 0**(pumping leema length) for any string w, such that |w|>=p, we can divide string w into xyz.
|xy|<=p (at most of length p) and |y|>=1, is not null.
Such that for i>=0 xy^iz belongs to L.
If language is finite then minimum pumping length is (finite string length + 1).

Note :

1. When language is finite then no need to apply pumping lemma because finite languages are always regular.
2. Pumping lemma is negative test means if one language satisfy conditions then it can be or can not be regular (undecidable) but if it does not satisfy the condition it is not a regular language (decidable).
3. For pumping lemma split the string into xyz and pump the y values if it is not belong to language then it is not regular.
4. Power of deterministic PDA < Power of non deterministic PDA. Deterministic languages cannot handle languages r grammer with ambiguity. So every NDPDA cannot be converted to an equivalent deterministic PDA.
5. A linear grammar is a context-free grammar that has at most one nonterminal in the right-hand side of each of its productions. That means all regular languages are linear but reverse is not true.
6. below language is CFL as we can create NDPA not DPA. Note that NDPA can create two states at the same time.

$L_1 \cup L_1 = \{a^n b^n c^m \cup a^m b^n c^n \mid n >= 0, m >= 0\}$

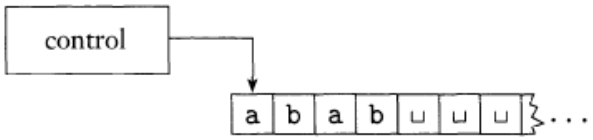
7. **Prefix Property**: If a string is a member of the language then no proper prefix of that string should be a member of the language. Example : a*b satisfy prefix property.

DPDA with empty stack acceptance has lesser power than DPDA with final state acceptance. NPDA with empty stack has the same power as NPDA with final state. We can construct a DPDA with empty stack for all those DCFLs which satisfy prefix property.

Turing Machine

A Turing Machine can do everything that a real computer do. Nonetheless, even a Turing machine cannot solve certain problems. **Those problems are beyond the theoretical limits of computation**.

1. The Turing machine model uses an **infinite tape as unlimited memory**.
2. It has a **tape head** that can **read and write** symbols and move around on the tape.
3. Initially **the tape contains only the input string** and is blank everywhere else.
4. If the machine needs to store information, it may **write the information on the tape**.
5. To read the information that it has written, the machine can move its **head back** over it.
6. The machine continues computing until it decides to produce an output.
7. The outputs **accept** and **reject** are obtained by **entering designated accepting and rejecting states**.
8. If it **doesn't enter an accepting or a rejecting state, it will go on forever**, **never halting**.



Schematic of a Turing Machine

The following list summarizes the differences between finite automata and Turing machines

1. A Turing machine can both write on the tape and read from it.
2. The read-write head can move both to the left and to the right.
3. The tape is infinite.
4. The special states for rejecting and accepting **take effect immediately**

Definition of a Turing Machine

A Turing machine M is defined by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$$

Where
Q is the set of **internal states**,
Σ is the **input alphabet** not containing blank symbol.
Γ is the finite set of symbols called the **tape alphabet**,
δ is the **transition function**,
$\square \in \Gamma$ is a special symbol called the **blank**,
q0 ∈ Q is the **initial state**,
F ⊆ Q is the **set of final states**.

Note: - In the definition of a Turing machine, we assume that $\Sigma \subseteq \Gamma - \{\square\}$

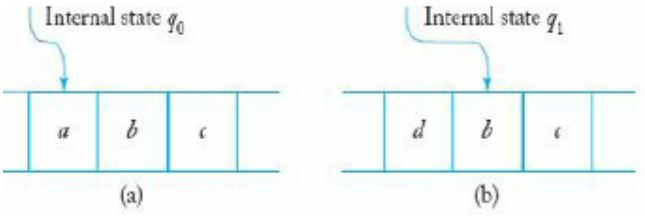
Transition Function: In general, **δ is a partial function on Q × Γ**; for a Turing machine, δ takes the form: $Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$

When machine is in a certain state q and the head is over a tape cell containing a symbol a, and If δ(q, a) = (r, b, L), the **machine writes the symbol b replacing a** and machine goes to state r.

The third component is either L or R and indicates whether the head moves to the left or right after writing. In this case the L indicate a move to the left.

Example: shows the situation before and after the move

δ (q0, a) = (q1, d, R).



A Turing machine is said to halt whenever it reaches a configuration for which δ is not defined; **this is possible because δ is a partial function**. We will assume that no transitions are defined for any final state, so the Turing machine will halt whenever it enters a final state.

Turing Machines as Language Accepters

1. A string w is written on the tape, with blanks filling out the unused portions.
2. The machine is started in the i**nitial state q0 with the read write head positioned on the leftmost symbol of w**.
3. If, after a sequence of moves, the Turing machine enters a final state and halts, then w is considered to be accepted.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ be a Turing Machine. Then the language accepted by M is

$$L(M) = \left\{ w \in \Sigma^+ : q_0 w \overset{*}{\vdash} x_1 q_f x_2 \text{ for some } q_f \in F, x_1, x_2 \in \Gamma^* \right\}$$

1. This definition indicates that the input w is written on the tape with **blanks on either side**.
2. Exclusion of blanks from the input assures us that all the input is re**stricted to a well-defined region of the tape**, **bracketed by blanks on the right and left**.
3. Without this convention, the machine could not limit the region in which it must look for the input; no matter how many blanks it saw, it could never be sure that there was not some nonblank input somewhere else on the tape.

Note that the Turing machine also halts in a final state if started in state q0 on a blank. We could interpret this as acceptance of λ, but for technical reasons the empty string is not included.

Note:-
The collection of strings that M accepts **is the language of M**, or the **language recognized by M**, denoted L(M)

Call a language ***Turing-recognizable*** if some Turing machine recognizes it.[1]

Call a language ***Turing-decidable*** or simply ***decidable*** if some Turing machine decides it.[2]

Variants of Turing Machine

1. **Multiple Turing Machine** It's like an ordinary T.M with several tapes. **Each tape has its own head for reading and writing**. Initially the input appears on tape 1, and the others start out blank. The transition function is:

$$\delta: Q \times \Gamma^k \longrightarrow Q \times \Gamma^k \times \{\mathrm{L}, \mathrm{R}, \mathrm{S}\}^k$$

Where k is the number of tapes.

Every multiple Turing machine has an equivalent single-tape Turing machine

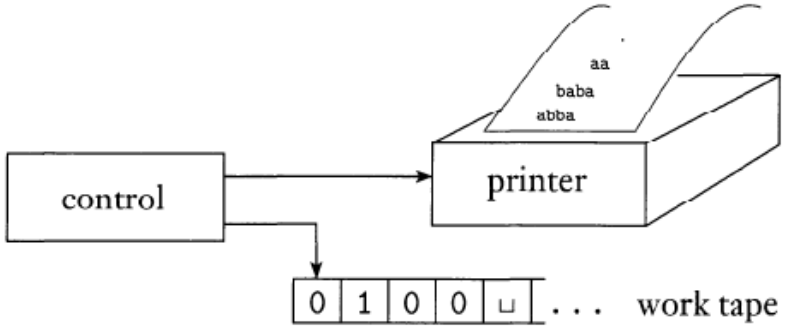
2. **Nondeterministic Turing Machine** transition function for a NTM has the form

$$\delta: Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\})$$

I think, P is powerset.

Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

3. **Enumerators** Some people use the term recursively enumerable language for Turing-recognizable language. Loosely defined, an enumerator is a Turing machine with an attached printer. The Turing machine can use that printer as an output device to print strings.



Schematic of an enumerator

An enumerator E starts with a blank input tape. If the enumerator doesn't halt it may print an infinite list of strings. The language enumerated by E is the collection of all strings that eventually prints out.

A language is Turing recognizable if and only if some enumerator enumerates it.

1. If we have an enumerator E that enumerates a language A, **A TM M recognizes A**. The TM M works in the following way:
M – "On input w"
   a. Run E. Every time that E outputs a string, compare it with w
   b. If w ever appears in the output of E, **accept.**
Clearly, M accepts those strings that appear on E's list.

2.

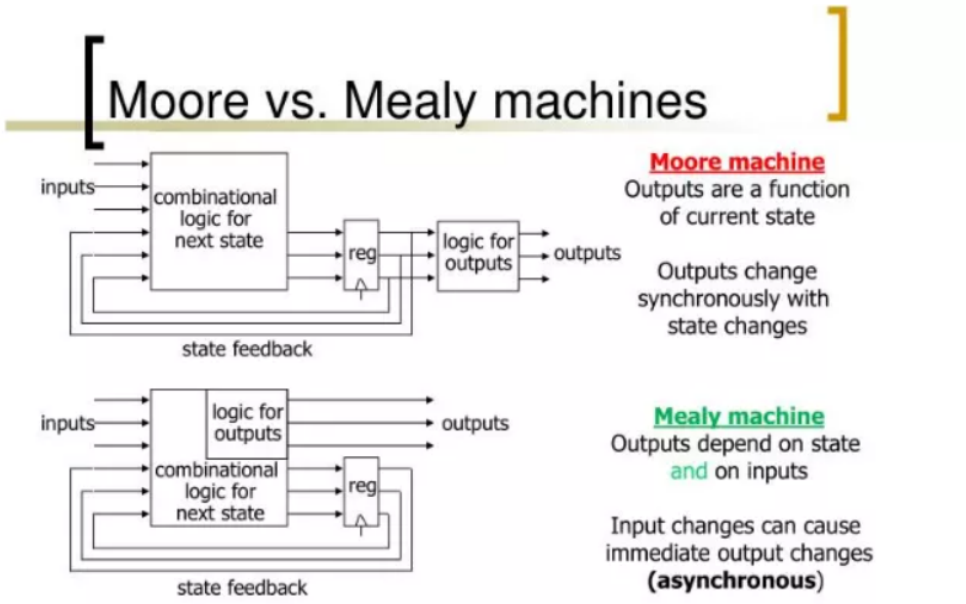
Now we do the other direction. If TM M recognizes a language A, we can construct the following enumerator E for A. Say that $s_1, s_2, s_3, \ldots$ is a list of all possible strings in Σ^*.

E = "Ignore the input.

1. Repeat the following for $i = 1, 2, 3, \ldots$
2. Run M for i steps on each input, $s_1, s_2, \ldots, s_i$.
3. If any computations accept, print out the corresponding s_j."

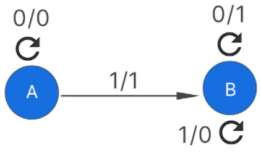
Finite State Machine

Finite state machines(FSM), and these machines are sequential circuits with a limited number of ways in which their history might affect their future behavior. And, they are used because there is no machine with an infinite storage capacity.
we will learn about the two types of finite state machines, the Mealy machine and the Moore machine.



1) Mearly Machine :
Characteristics of Mealy Machines
The mealy machine can be symbolized as (Q, ∑, O, δ, X, q0) where −

Q is a finite set of states.
∑ It is a finite set of symbols known as the input alphabet.
O is a finite set of symbols known as the output alphabet.
δ is the input transition function (δ: Q × ∑ → Q)
X is the output transition function (X: Q × ∑ → O)
q0 is the initial state; any input is processed from this state(q0 ∈ Q).
Example :
We can design a Mealy machine that generates 2's complement of a binary number.

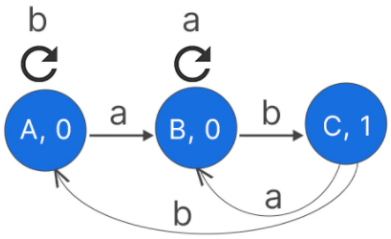


	0		1	
	State	Out put	State	Output
A	A	0	B	1
B	B	1	B	0

Here read input from righthand side but in this example only.
Note : In Mearly machine, on every input from one state to other state there is different output.

2) Moore Machine :
Characteristics of Moore Machines
Moore machine can be symbolized as (Q, ∑, O, δ, X, q0) where −
Q is a finite set of states.
∑ It is a finite set of symbols known as the input alphabet.
O is a finite set of symbols known as the output alphabet.
δ is the input transition function (δ: Q × ∑ → Q)
X is the output transition function (X: Q → O)
q0 is the initial state; any input is processed from this state(q0 ∈ Q).

Example :
We can design a Moore machine that takes a set of all strings having symbols {a, b} as input and can print '1' as output for each occurrence of 'ab' as a substring.



	a	b	Δ
A	B	A	0
B	B	C	0
C	B	A	1

Note :
> In Moore machine, on every input from one state to other state there is same output. As you can notice there are no final state in mearly and moore machine.
> We can easily convert mearly to moore and vice versa.
> If an N state Mealy Machine having M outputs is converted into a Moore machine, then the resulting machine can have upto MN states, so we can have M(N – 1) additional states after the conversion. But In case of Moore to Mealy number of state do not change you can varify this fact.

Decidability

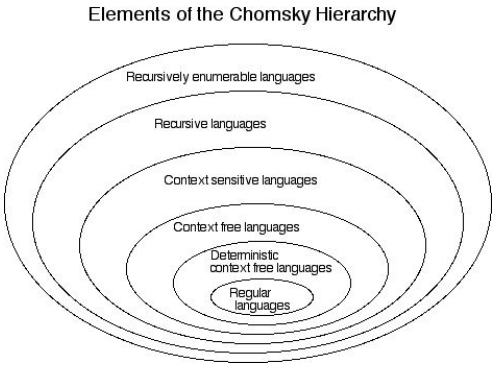
Closure Properties

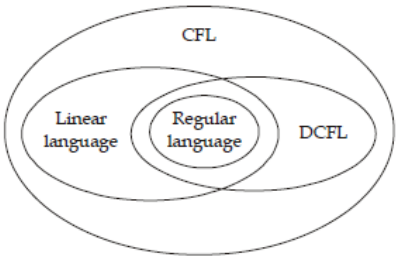
Operation	Regular	DCFL	CFL	CSL	Recursive	RE
Union	yes	no	yes	yes	yes	yes
Intersection	yes	no	no	yes	yes	yes
Complement	yes	yes	no	yes	yes	no
Concatenation	yes	no	yes	yes	yes	yes
Kleene star	yes	no	yes	yes	yes	yes
Homomorphism	yes	no	yes	no	no	yes
ϵ-free Homomorphism	yes	no	yes	yes	yes	yes
Substitution (ϵ-free)	yes	no	yes	yes	no	yes
Inverse Homomorphism	yes	yes	yes	yes	yes	yes
Reverse	yes	no	yes	yes	yes	yes
Intersection with a regular language	yes	yes	yes	yes	yes	yes

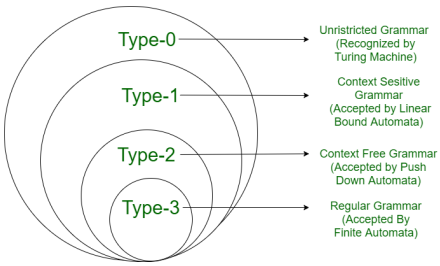
Not closed means may or may not be in language but if closed then it is for sure in language.

1. All types of languages are closed under all the operations with regular languages such as LUR, L∩R, L-R.
2. CFLs are not closed under difference operation as L1-L2 = L1∩L2^c, and CFLs are not closed under complement operation.
3. No languages are closed under subset ⊆ and Infinite union.
4. Regular languages are not closed under **infinite UNION** and **infinite INTERSECTION**
5. If L is DCFL then so are **MIN(L)** and **MAX(L)**.
6. Complement of **non-regular** is always non-regular
7. If L be a DCFL and R is a regular language then **L/R** is DCFL.
8. DCFL U CFL = CFL
9. If something is closed under UNION and COMPLEMENT then it will be surely closed under INTERSECTION.
10. Do Not blindly follow this table if intersection of two CFL is Null then it is CFL which contradict this table. Similar cases for other languages may happen.

Operations	REG	CFL
INIT	✓	✓
L/a	✓	✓
CYCLE	✓	✓
MIN	✓	X
MAX	✓	X
HALF	✓	X
ALT	✓	X







Let L be a language

1. **HALF(L)** = {x | for some y such that |x| = |y| and xy ∈ L }
2. **MIN(L)** = { w | w is in L and no proper prefix of w is in L}
3. **MAX(L)** = {w | w is in L and for no x other than epsilon wx is in L}
4. **INIT(L)** = {w | for some x, wx is in L}
5. **CYCLE(L)** = { w | we can write w as w=xy such that yx is in L}
6. **ALT(L, M)** is regular provided that L and M are regular languages.
7. **SHUFFLE(L, L')** is a CFL if L is CFL and L' is regular.
8. **SUFFIX(L) =** { y | xy ∈ L for some string x }, **CFL is closed under SUFFIX operation.**
9. **NOPREFIX(L)** = {w ∈ A | and no prefix of w is member of A }
10. **NOEXTEND(L) =** { w ∈ A | w is not proper prefix of any string in A }
11. **DROP-OUT(L)** let A be any language, define DROP-OUT(L) to be the language containing all strings that can be obtained by **removing one symbol from a string in L.**
12. Regular languages are closed under NOPREFIX, NOEXTEND, and DROP-OUT operations.
13. **Ri ght Quot i ent (L1/L2)** = {x|xy s uch t hat x∈L1 and y∈L2 i s not possible t hen ∈} L1={carrot} and L2={t,ot}. Then L1/L2 is {carro,carr}
14. **Lef t Quot i ent (L1/L2)** = {y|xy s uch t hat x∈L1 and y∈L2 i s not possible t hen ∈} Here, they have taken x as Suffix rather than taking it as Prefix (Left Quotient).

Note : These closure properties hold for both left and right quotients.
The quotient of a regular language with any other language is regular.
The quotient of a context free language with a regular language is context free.
The quotient of two context free languages can be any recursively enumerable language.
The quotient of two recursively enumerable languages is recursively enumerable.

Operations	REG	DCFL	CFL	CSL	REC	RE	Comments
w ∈ L(G)	✓	✓	✓	✓	✓	X	Membership property.
L(G)=Φ	✓	✓	✓	X	X	X	Emptiness Property
L(G)=Σ*	✓	✓	X	X	X	X	Language accepts everything?
L(G1)⊆L(G2)	✓	X	X	X	X	X	Is L(G1) subset of L(G2)?
L(G1)=L(G2)	✓	✓	X	X	X	X	Are both languages equal?
L(G1)∩L(G2)=Φ	✓	X	X	X	X	X	Disjointness Property.
L(G) is regular?	✓	X	X	X	X	X	G generates Regular language.
L(G) is finite?	✓	✓	✓	X	X	X	G generates finite language?
Ambiguity	✓	X	X	X	X	X	Is the given grammar ambiguous?

Decidability Table

1. Membership, (Me Feared)
2. Emptiness,
3. Finiteness,
4. Equivalence,
5. Ambiguity,
6. Regularity,
7. Everything,
8. Disjointedness

All are decidable for Regular languages.
DCFL - 12347
CFL - 123
CSL and REC - 1
RE - none

Non-Trivial Property :
A property which is satisfied by all elements of a set or NO element of a set is called a TRIVIAL property. Any other property is called non-trivial. If P is a non-trivial property of a set S, then some elements of S will satisfy the property and some will violate the property.

- ***Why non trivial stuff*** :

No one cares for trivial stuffs. One real life example of a set and a property is as follows:
● Consider the set of all GATE 2019 applicants
● Whether an applicant has an application number - is a trivial question because everyone will be having it
● Whether an applicant location is Bangalore is not a trivial question
● For any non-trivial property P, we will have a Pyes instance and a Pno instance.

Note : Remember one thing always that the language accepted by the TM is recursively enumerable. So whenever you find TM means the L is always RE.
Example : Consider some non-trivial properties,
"Whether a given RE set has string starting with 1." we write this as
{<M> | Turing machine that accepts any string starting with 1}.

Lets see **Rice's theorem Part 1** :
Any non-trivial property of Recursively Enumerable set is UNDECIDABLE.
i.e., The following problems are undecidable:
1. If a given recursively enumerable language has a particular string
2. If a given recursively enumerable language is finite/regular/context-free
3. If number of strings in the given recursively enumerable language is 10 (or any other number).
But you can design one algorithm in polynomial time that will answer the question so this will become decidable. For example : Given a graph G, does it have a cycle? Is this problem decidable? - Yes of course. Just do a DFS and detect presence of backedge.

● Undecidable - for some inputs we are not able to say "yes"/"no"
● Semi-decidable - for all inputs with answer "yes" we can say "yes"
● Decidable - for all inputs we can answer "yes" or "no".
● So, some problems can be undecidable but still be semi-decidable
● All decidable problems are semi-decidable too.
● Halting problem is an undecidable problem which is semi-decidable

Semidecidability :
if we can give a procedure which works for "yes" case, the problem becomes semi-decidable and its corresponding language becomes recursively enumerable.
Lets see **Rice's theorem Part 2** :
"Any non-monotonic property of recursively enumerable set is not even semi-decidable"
Compared to Part 1, we have ● Non-trivial becoming non-monotonic ● Undecidable becoming not even semi-decidable.

Non-trivial condition requires that we have Lyes and Lno. Non-monotonic condition requires an additional condition that Lyes SUBSET Lno (is it proper subset?) We can take ANY possible Lyes and Lno to ensure the above property. Lyes should be proper subset of Lno.

L = { | L(M) is finite}
Lyes = Φ (Any other finite language will also do)
Lno = Σ^* , L yes $\subset$ L no ($\Phi \subset \Sigma^*$) - non - monotonic
$\therefore$ L(M) is not Turing recognizable (not RE)

Properties :
If a property of RE set is non-trivial, it is undecidable
If a property of RE set is trivial, it is decidable (Trivial means it)
If a property of RE set is non-monotonic, it is not even semi-decidable
If a property of RE set is not non-monotonic, it may/may not be semi-decidable

Some trivial properties :
L = { <M>| L(M) is recursively enumerable}
L = { <M>| L(M) is countable}
L = { <M>| L(M) can be accepted by a Turing machine with even number of states} L = { <M>| There exist a deterministic Turing machine which accepts L(M)}

How to know which language is recursive and recursively enumerable ?
Answer :
When we have a TM which always halts on "yes" case of a problem, that language is recursively enumerable.
1. If this TM halts for all "no" cases, then its language becomes recursive.
2. If for some "no" strings the TM does not halt, then the language becomes recursively enumerable but not recursive.
3. TM for a RE language either rejects "no" strings or goes into an infinite loop.
So, When a problem is decideble then the language describing is REC.
You can use non-monotonic property to determine whether a language is **not even RE.**
While making yes and no cases you should not try to cover all the cases that satisfy yes propety only one relation or case is enough to show whether Lyes is proper subset of Lno.

Note :

1) Ambiguous grammar and its dissambiguous version will accept the same language the only difference is there are more way to derive string in ambiguous version.
2) Any palindrome language is non regular, means we can not make finite automata for such languages.
3) (r* s*)* = (r + s)*
4) All regular languages are context free, but converse is not true. That means union of regular and CFL is regular which means it is also CFL.
5) Difference between DPDA and NPDA is DPDA always accepts unambiguous languages, always have one choice on per input unlike NPDA who have capability to enter two state on same output, is proper subset of NPDA.
6) Always try from small example to big In language acceptation.
7) A language can be recognizable if the TM either terminates and rejects the string or doesn't terminate at all. This means that the TM continues with the computing when the provided input doesn't lie in the language. Recursive enumerable languages are also known as recognizable languages.
8) Whereas, the language is decidable if and only if there is a machine which accepts the string when the provided input lies in that language and rejects the string when provided input doesn't lie in that language. Recursive language is also called Turing decidable

Turing-recognizable languages:

- TM halts in an accepting configuration if w is in the language.
- TM may halt in a rejecting configuration or go on indefinitely if w is not in the language.

Turing-decidable languages:

- TM halts in an accepting configuration if w is in the language.
- TM halts in a rejecting configuration if w is not in the language.

10) Reversal of FA can be obtained by just switching final state to initial state and vice versa. Without touching other state and flipping the arrows to other side.
11) In case of grammar checking from production rules apply table wise approach for example first write string then below that line apply production rule and so on.
12) In case of minimization of DFA you can not combine anystate with final state. You can combine initial state with simple state but not with final state.
13) Page number 20, 21 pg monu thakur is important don't forget to see RE and CSL column.
14) A language for which every grammar is ambiguous is called inherently ambiguous language. So if language is regular or DCFL it will surely can be written in unambiguous grammar. So to check whether the given language is not inherently ambi. We make grammar if can be accepted by DPDA then it is not IA.
15) Every infinite regular language L has a subset S which is undecidable.
Every infinite regular language L has a subset S which is unrecognizable.
Every infinite language L has a subset S which is undecidable.
Every infinite language L has a subset S which is unrecognizable.
16) if $L(A) \subseteq L(B)$ then $L(A) \cap L^c(B) = \emptyset$
17) If L* is decidable then L cannot be decidable because alphabet of L may contains strings which are undecidable. But the closer of it would be all languages which is regular.
18) "**All words accepted by FA A contains one more 'a' than b**" is different from "**The automata A accpets all words over sigma that contains one more 'a' than 'b'**". Second sentence may contains strings which is not accepted by first sentence.
19) Every Turing recognizable set is included in a Turing recoganizable set. As every turing recognizable set is subset of sigma closure. And sigma closure in turn is decidable.

_____ TURING MACHINE _____

From my notes:-

1. HALT-TM = {<M, w> | M is a TM and M halts on input w} – **Undecidable, RE**
2. E-TM = {<M> | M is a TM and L(M) = Φ} – **Undecidable, NOT RE**
3. EN-TM = {<M> | M is a TM and L(M) != Φ} – **Undecidable, RE**
4. REGULAR-TM = {<M> | M is a TM and L(M) is a regular language} – **Undecidable, NOT RE**
5. REGULAR-TM = {<M> | M is a TM and L(M) is a REC}, **Undecidable, NOT RE**
6. REGULAR-TM = {<M> | M is a TM and L(M) is a NOT REC}, **Undecidable, NOT RE**
7. EQ-TM = {<M1, M2> | M1 and M2 are TMs and L(M1) = L(M2)} – **Undecidable, NOT RE**
8. A-LBA = {<M, w> | M is an LBA that accepts string w} – **Decidable**
9. E-LBA = {<M> | M is an LBA where L(M) = Φ} – **Undecidable, NOT RE**
10. All-CFG = {<G>|G is a context free grammar and L(G) = Σ∗} **Undecidable, NOT RE**
11. T = {<M > | M is a TM that accepts w^r whenever it accepts w} **undecidable, RE?**
12. A TM ever writes a blank symbol over a non-blank symbol during the course of its computation. **Undecidable, NOT RE**
13. L3 = {< G> | G is ambiguous} **RE**, while L3' is **NOT RE**.

Note:-

1. TM >> LBA(FA + 2 counter) > NPDA(NFA + 1 counter) > DPDA (DFA + 1 counter) > NFA = DFA
2. Any TM with m symbols and n states can be simulated using **4mn + n** states by other TM.
3. If A <=p B (A is polynomial reducible to B, if A is NOT RE then B is NOT RE too.
4. A <=p A', If A is Turing recognizable, then A is decidable.

Some Decidable/ Undecidable Problems

a. L(M) has **at least** 10 strings – **RE**
b. L(M) has **at most** 10 strings – **NOT RE**
c. L = {M | M is a TM that accepts a string of length 2014} – **RE**
   There are finite number of strings of length 2014, if we can execute multiple instances of TM in parallel, if any string is accepted we can stop.
d. L(M) is recognized by a TM having even number of states. **Decidable (trivial property)**
e. If $A \leq_m B$ and B is a regular language, does that imply that A is a regular language? **(NO)**
f. if a language A is in RE and $A \leq_m \overline{A}$, then A is recursive. **(TRUE)**
g. L={⟨M,w⟩|M does not modify the tape on input w} - **decidable**
h. an arbitrary TM ever prints a specific letter – **Undecidable**

Mapping Reduction :

• A mapping reduction $A \leq_m B$ (or $A \leq_P B$) is an algorithm (respectively, polytime algorithm) that can transform any instance of decision problem A into an instance of decision problem B, in such a way that the answer correspondence property holds.
• Answer correspondence property: The answer (yes/no) to any A problem instance must be the same as the answer to the corresponding B problem instance to which the reduction transforms it.
What does that mean : $A \leq_m B$ means "A problems are no harder to solve than B problems.", $A \leq_m B$ means "Being able to solve any B problem $\Rightarrow$ being able to solve any A problem"
It also mean B -> A and not(A) => not(B)

Formal languages and countability :

1) The set of all formal languages is countably infinite. or The number of strings in a language is countably infinite.

proof : - Divide the strings of the languages into subsets based on their lenght; i.e., put all strings of same length together, etc.

- Within each set, put the strings in lexiccographical order
- Merge the subsets, preserving their order
- Now put strings into one to one correspondance with the counting numbers.

2) The set of all languages is the power set of all strings. The power set of a countably infinite set is uncountably infinite, so the set of all binary languages is uncountably infinite.

3) Set of all regular languages over sigma which is the set of words is countable

Proof : The set of all regular languages is a subset of the set of all recursively enumerable languages. And a subset of a countable set is always countable. This is because all the elements of the countable set can be written in a specific order and each of that element can be checked for its membership in the other set.

4) Set of all languages over sigma accpeted by Turing machines is countably infinite because each TM can be represented by a binary string and each binary string can be obtained in a proper order and checked whether it's a TM or not.

5) Set of all non-regular language over sigma is uncountable.

6) Cantor's theorem says that if S is any set then $|S|<|P(S)|$ where P(S) is power set of S. Form this we can say that if S is countably infinite set then P(S) is uncountable.

Now, we know every language is countable. If we have any infinite language L, then it means that we have uncountable many subsets of L but we know that set of RE languages is countable so, due to this we have a subset of L which is not RE. Not RE means undecidable. So, every infinite language L has a subset S which is undecidable. Take contrapositive, if we have a language L which doesn't have an undecidable subset then it means that L is finite.