

# Finite Automata and Regular Languages

## FUNDAMENTALS

**Alphabet:** alphabet is a finite non-empty set of symbols

$\Sigma$  denotes an alphabet

**String:**

- a string over an alphabet 'A' is a finite ordered sequence of symbols from 'A'. The length of the string is the number of symbols in string with repetitions counted
- an empty string denoted by ' $\epsilon$ ', is the (unique) string of length zero.
- If S and T are sets of strings then  $ST = \{xy \mid x \in S \text{ and } y \in T\}$
- Given an alphabet A,
  - $A^0 = \{\epsilon\}$
  - $A^{n+1} = A.A^n$
  - ...
  - $A^* = \bigcup_{n=0}^{\infty} A^n$

## *LANGUAGES*

- A language 'L' over  $\Sigma$  is any finite or infinite set of strings over  $\Sigma$ .
- The elements in L are strings – finite sequences of symbols
- A language which does not contain any elements is called an 'empty language'
- Empty language resembles empty set  $\Rightarrow \{\} = \Phi \neq \epsilon$
- A language L over an alphabet A is subset of  $A^*$  i.e.  $L \subseteq A^*$
- An empty language is a language that does not accept any strings including  $\epsilon$ . ( $\rightarrow \emptyset$ )
- A language which only accepts  $\epsilon$  ( $\rightarrow \{\epsilon\}$  (concentric))

## *OPERATIONS*

Operations on strings

1. **Concatenation:** Combines two strings by putting one after the other (a.b)
  - Concatenation of empty string with any other string gives the string itself
2. **Substring:** If 'w' is a string, then 'v' is a substring of 'w' if there exists string x and y such that  $w = xvy$ . 'x' is called the prefix and 'y' is called the suffix of w.
3. **Reversal:** if 'w' is a string, then  $w^R$  is reversal of string spelled backwards.
  - $x = (x^R)^R$
  - $(xz)^R = z^R.x^R$
4. **Kleen star operation:** Let 'w' be a string  $w^*$  is set of strings obtained by applying any number of concatenations of w with itself, including empty string.
  - Example:  $a^* = \{\epsilon, a, aa, aaa, \dots\}$

Operations on Languages

1. **Union:** Given some alphabet  $\Sigma$ , for any two languages,  $L_1, L_2$  over  $\Sigma$ , the union  $L_1 \cup L_2$  of  $L_1$  and  $L_2$  is the language  $L_1 \cup L_2 = \{w \in \Sigma^* \mid w \in L_1 \text{ or } w \in L_2\}$

2. **Intersection:** Given some alphabet  $\Sigma$ , for any two languages,  $L_1, L_2$  over  $\Sigma$ , the intersection  $L_1 \cap L_2$  of  $L_1$  and  $L_2$  is the language  $L_1 \cap L_2 = \{w \in \Sigma^* \mid w \in L_1 \text{ and } w \in L_2\}$
3. **Difference / Relative Complement:** Given some alphabet  $\Sigma$ , for any two languages,  $L_1, L_2$  over  $\Sigma$ , the difference  $L_1 - L_2$  of  $L_1$  and  $L_2$  is the language  $L_1 - L_2 = \{w \in \Sigma^* \mid w \in L_1 \text{ and } w \text{ does not belong to } L_2\}$
4. **Concatenation:** Given some alphabet  $\Sigma$ , for any two languages,  $L_1, L_2$  over  $\Sigma$ , the concatenation  $L_1 L_2$  of  $L_1$  and  $L_2$  is the language  $L_1 L_2 = \{w \in \Sigma^* \mid \exists u \in L_1, \exists v \in L_2 \text{ and } w = uv\}$ 
  - **Properties**
  - $L\emptyset = \emptyset = \emptyset L$
  - $L\{\epsilon\} = L = \{\epsilon\}L$
  - $(L_1 \cup \{\epsilon\})L_2 = L_1L_2 \cup L_2$
  - $L_1(L_2 \cup \{\epsilon\}) = L_1L_2 \cup L_1$
  - $L^nL = LL^n = L^{n+1}$ 
    - $L_1L_2 \neq L_2L_1$
5. **Kleen \* Closure ( $L^*$ ):** Given an alphabet  $\Sigma$ , for any language  $L$  over  $\Sigma$ , the  $*$  closure  $L^*$  of  $L$  is language,  $L^* = \bigcup_{n \geq 0} L^n$
6. **Kleen + Closure ( $L^+$ ):** Given an alphabet  $\Sigma$ , for any language  $L$  over  $\Sigma$ , the kleen  $+$  closure  $L^+$  of  $L$  is language,  $L^+ = \bigcup_{n \geq 1} L^n$

#### Properties

- $\emptyset^* = \{\epsilon\}$
- $L^+ = L^*L$
- $(L^*)^* = L^*$
- $L^*L^* = L^*$

## FINITE STATE MACHINES

FSM is the simplest computational model of limited memory computers. It is designed to solve decision problems, i.e, to decide whether the given input satisfies certain conditions. The next state and output of FSM is a function of input and the current state.

### Types of FSM

- Mealy machine
- Moore machine

### Finite Automata

- FA is a state machine that comprehensively captures all possible states and transitions that a machine can take while responding to a stream (sequence) of input symbols.
- FA is recognizer of 'regular languages'

### State Machine

- Finite state machines
  - Mealy machine
  - Moore machine
- Finite automata
  - DFA
  - NFA

- epsilon – NFA

## Types of Finite Automata

### 1. Deterministic Finite Automata

- DFA can exist in only one state at a time
- DFA is defined by 5-tuple :  $\{Q, \Sigma, q_0, F, \delta\}$ 
  - $Q \rightarrow$  Finite number of states (elements)
  - $\Sigma \rightarrow$  Finite set of symbols (alphabets)
  - $q_0 \rightarrow$  Start/Initial state
  - $F \rightarrow$  Set of final states
  - $\delta \rightarrow$  Transition function, which is a mapping between
    - $\delta: Q \times \Sigma \rightarrow Q$
- How to use DFA

### Transition Diagram

State machines are represented by directed graphs called transition (state) diagrams.

Vertices : states

Arcs : transition

Double concentric circles: Final states

### Transition Table

Transition function can be represented by tables.

*Note: minimum number of states for k-divisibility is k-states*

### 2. Non-deterministic Finite Automata

- The machine can exist in multiple states at the same time
- Each transition function maps to a set of states
- NFA is defined by 5-tuple:  $\{Q, \Sigma, q_0, F, \delta\}$ 
  - $Q \rightarrow$  Finite number of states (elements)
  - $\Sigma \rightarrow$  Finite set of symbols (alphabets)
  - $q_0 \rightarrow$  Start/Initial state
  - $F \rightarrow$  Set of final states
  - $\delta \rightarrow$  Transition function, which is a mapping between
    - $\delta: Q \times \Sigma \rightarrow 2^Q$
- How to use NFA

## Difference between NFA and DFA

DFA	NFA
1. All transitions are deterministic i.e., each transition leads to exactly one state	1. Transitions could be non-deterministic i.e., a transition could lead to a subset of states
2. For each state, the transition on all possible	2. For each state, not all symbols necessarily

symbols should be defined	have to be defined
3. Accepts input if the last state is in F	3. Accepts input if one of the last states is in F.
4. Practical implementation is feasible	4. Practical implementation has to be deterministic. It needs conversion from NFA to DFA

### Relation between DFA and NFA

1. A language L is accepted by a DFA if and only if it is accepted by NFA
2. DFA is a special case of NFA
3. Every language accepted by NFA is also accepted by a DFA.  $D_f = N_f$

### NFA WITH $\epsilon$ – MOVES

- $\epsilon$ -transitions in finite automata allows a state to jump to another state without consuming any input symbol

#### Conversion and equivalence:

$\epsilon$  – NFA  $\rightarrow$  NFA  $\rightarrow$  DFA

#### NFA without $\epsilon$ – moves:

- Two FA,  $N_\epsilon$  and  $N$  are said to be equivalent, if  $L(N_\epsilon) = L(N)$  i.e., any language described by some  $N_\epsilon$  there is a  $N$  that accepts the same language.
- For  $N_\epsilon = \{Q, \Sigma, q_0, F, \delta\}$  and  $N = \{Q, \Sigma', q_0, F', \delta'\}$ , find
- $\delta'(q, a) = \epsilon$  – closure ( $\delta(\epsilon$  – closure( $q, a)$ )
- $F' = \{F \cup \{q_0\}\}$ , if  $\epsilon$  – closure ( $q_0$ ) contains a member of  $F$ , otherwise
- *Note: when transforming  $N_\epsilon$  to  $N$ , only transitions are required to be changes and states remains same.*

### CONVERSION OF NFA TO DFA

Let NFA be defined as  $N = \{Q_N, \Sigma, q_0, F_N, \delta_N\}$

The equivalent DFA,  $D = \{Q_D, \Sigma, q_0, F_D, \delta_D\}$

**Step 1:**  $Q_D = 2^{Q_N}$ , i.e.,  $Q_D$  is set of all subsets of  $Q$ , i.e., it is power set of  $Q_N$

**Step 2:**  $F_D$  is set of subsets  $S$  of  $Q_N$  such that  $S \cap F_N \neq \emptyset$ , i.e.,  $F_D$  is all sets of  $N$ 's states that include once accepting state of  $N$ .

**Step 3:** For each set,  $S \subseteq Q_N$  and for each input symbol  $a$  in

$\Sigma$ :  $\delta_D(S, a) = \bigcup_{P \in S} \delta_N(P, a)$

That is to compute  $\delta_D(S, a)$ , look at all states  $P$  in  $S$ , see what states  $N$  goes to starting from  $P$  on input  $a$ , and take union of all those states.

*Note: For any NFA,  $N$  with ' $n$ ' states, the corresponding DFA can have  $2^n$  states.*

### MINIMIZATION OF DFA

Minimization of DFA means that we are minimizing the representation of a FA to its least form. Here basically we replace multiple states with a single state without disturbing the representation. In a DFA states  $p$  and  $q$  are equivalent when:

$\delta(p, w) \in F \Rightarrow \delta(q, w) \in F$  and  $\delta(p, w) \notin F \Rightarrow \delta(q, w) \notin F$

if  $\text{len}(w) = 0$  and  $p$  and  $q$  follow the above property:  $p$  and  $q$  are 0-equivalent

...

if  $\text{len}(w) = n$  and  $p$  and  $q$  follow the above property:  $p$  and  $q$  are  $n$ -equivalent

Therefore, instead of having two states, we can combine them into one state and decrease the number of states in the final answer.

We are using Partitioning Method here.

Step1: Identify the start and final state

Step2: If there is any state which is unreachable from the initial state, delete it.

Step3: draw the state transition table

Step4: Find the 0-equivalent sets. Separate non-final states from final states

Step5: Find the 1-equivalent sets. Separate the non equivalent from equivalent

Step5: Keep finding the  $n$ -equivalent sets until the next equivalent calculation is identical to the previous one.

Remove the dead states

## **EQUIVALENCE BETWEEN NFA AND DFA**

There is a DFA for any NFA, i.e.,

$L(D) = L(N)$ .

### **Construction:**

- In DFA or NFA, whenever an arrow is followed, there is a set of possible states. This set of states is a subset of  $Q$ .
- Track the information about subsets of states that can be reached from the initial state after following arrows.
- Consider each subset of states of NFA as a state of DFA and every subset of states containing a final state as a final state of DFA.

### **Equivalence of Finite Automata**

- Two automata  $A$  and  $B$  are said to be equivalent if both accept exactly the same set of input strings
- If two automata  $M_1$  and  $M_2$  are equivalent then
  - If there is a path from the start state of  $M_1$  to a final state of  $M_1$  labeled as  $a_1, a_2 \dots a_k$  then there is a path from the start state of  $M_2$  to the final state of  $M_2$  labeled as  $a_1, a_2, \dots a_k$
  - If there is a path from the start state of  $M_2$  to a final state of  $M_1$  labeled as  $b_1, b_2 \dots b_k$  then there is a path from the start state of  $M_1$  to the final state of  $M_2$  labeled as  $b_1, b_2, \dots b_k$

**Union:** The union of languages  $L$  and  $M$  is the set of strings that are in both  $L$  and  $M$

**Concatenation:** The concatenation of languages  $L$  and  $M$  is the set of strings that can be formed by taking any string in  $L$  and concatenating it with any string in  $M$ .

### **Closure, star or Kleen star of Language $L$ :**

Kleen star is denoted as  $L^*$ . It represents the set of strings that can be formed by taking any number of strings from  $L$  with repetition and concatenating them. It is a Unary operator.

$L_0$  is the set; we can make selecting zero strings from  $L$ .

$L_0 = \{\epsilon\}$

$L_1$  is the language consisting of selecting one string from  $L$ .

$L_2$  is the language consisting of concatenations selecting two strings from  $L$ .

...

$L^*$  is the union of  $L_0, L_1, \dots$  Linf

Ex:  $L = \{0, 10\}$

$L^* = \{0, 00, 000, 10, 010, \dots\}$

### Intersection:

Let two DFAs  $M_1$  and  $M_2$  accept the languages  $L_1$  and  $L_2$ .

$M_1 = \{Q_1, \Sigma, q_0^1, F_1, \delta_1\}$

$M_2 = \{Q_2, \Sigma, q_0^2, F_2, \delta_2\}$

The intersection of  $M_1$  and  $M_2$  can be given as

$M = \{Q, \Sigma, q_0, F, \delta\}$

$Q$  = Pairs of states, one from  $M_1$  and one from  $M_2$  i.e.,

$Q = \{(q_1, q_2) \mid q_1 \in Q_1 \text{ and } q_2 \in Q_2\}$

$Q = Q_1 \times Q_2$

$q_0 = (q_0^1, q_0^2)$

$\delta((q_i^1, q_j^2), x) = (\delta_1(q_i^1, x), \delta_2(q_j^2, x))$

$F = \{(q_1, q_2) \mid q_1 \in F \text{ and } q_2 \in F\}$

### Union:

Let two DFAs  $M_1$  and  $M_2$  accept the languages  $L_1$  and  $L_2$ .

$M_1 = \{Q_1, \Sigma, q_0^1, F_1, \delta_1\}$

$M_2 = \{Q_2, \Sigma, q_0^2, F_2, \delta_2\}$

The union of  $M_1$  and  $M_2$  can be given as

$M = \{Q, \Sigma, q_0, F, \delta\}$

$Q$  = Pairs of states, one from  $M_1$  and one from  $M_2$  i.e.,

$Q = \{(q_1, q_2) \mid q_1 \in Q_1 \text{ and } q_2 \in Q_2\}$

$Q = Q_1 \times Q_2$

$q_0 = (q_0^1, q_0^2)$

$\delta((q_i^1, q_j^2), x) = (\delta_1(q_i^1, x), \delta_2(q_j^2, x))$

$F = \{(q_1, q_2) \mid q_1 \in F \text{ or } q_2 \in F\}$

### Difference:

The difference of  $L_1$  and  $L_2$  can be given as:

$L_1 - L_2$  with  $M = \{Q, \Sigma, q_0, F, \delta\}$

$Q = Q_1 \times Q_2$

$\delta((q_i^1, q_j^2), x) = (\delta_1(q_i^1, x), \delta_2(q_j^2, x))$

$F = \{(q_1, q_2) \mid q_1 \in F \text{ and } q_2 \text{ does not } \in F\}$

### Reversing a DFA:

- $M$  is a DFA which recognizes the language  $L$
- $M^R$  will accept the language  $L^R$
- To construct  $M^R$ 
  - Reverse all transitions
  - Turn the start state to the final state
  - Turn the final states to the start state
  - Merge states and modify the FA such that the resultant contains a single start state.

## MEALY AND MOORE MACHINES

### Moore Machine

A Moore machine is a finite state machine, where outputs are determined by current state alone. A moore machine associates an output symbol with each state and each time a state is entered, an output is obtained simultaneously. So, first output always occurs as soon as the machine starts.

Moore machine is defined by 6-tuple:

$\{Q_1, \Sigma, \delta_1, q_0, \Delta, \lambda\}$

- $Q \rightarrow$  Finite number of states
- $\Sigma \rightarrow$  Finite set of input symbol
- $\Delta \rightarrow$  It is an output alphabet
- $\delta \rightarrow$  Transition function,  $Q \times \Sigma \rightarrow Q$  (state function)
- $\lambda \rightarrow$  Output function,  $Q \rightarrow \Delta$  (machine function)
- $q_0 \rightarrow$  Initial state of machine

*Note: The output symbol at a given time depends only on present state of moore machine.*

### Mealy Machine

- A Mealy machine is a FSM, where outputs are determined by current state and input.
- It associates an output symbol with each transition and the output depends on current input
- Mealy machine is defined as 6-tuple  $\{Q_1, \Sigma, \delta_1, q_0, \Delta, \lambda\}$  where,
  - $Q \rightarrow$  Finite number of states
  - $\Sigma \rightarrow$  Finite set of input symbol
  - $\Delta \rightarrow$  It is an output alphabet (finite set of output symbols)
  - $\delta \rightarrow$  Transition function,  $Q \times \Sigma \rightarrow Q$  (state function)
  - $\lambda \rightarrow$  Output function,  $Q \rightarrow \Delta$  (machine function)
  - $q_0 \rightarrow$  Initial state of machine ( $q_0 \in Q$ )

*Note: In Moore machine, for input string of length  $n$ , the output sequence consists of  $(n+1)$  symbols  
In Mealy machine, for input sequence of length  $n$ , the output sequence also consists of ' $n$ ' symbols*

### Equivalence of Moore and Mealy machine

#### (a) Mealy machine equivalent to Moore Machine

If  $M1 = \{Q, \Sigma, \delta, q_0, \Delta, \lambda\}$  is a Moore machine, then there is a Mealy machine equivalent to  $M1$ .

Proof: Let  $M2 = \{Q_1, \Sigma, \delta, q_0, \Delta, \lambda^1\}$  and define  $\lambda^1(q, a)$  to be  $\lambda(\delta(q, a))$  for all states  $q$  and input symbol ' $a$ '. Then,  $M1$  and  $M2$  enter the same sequence of state on the same input, and with each transition  $M2$  emits the o/p that  $M1$  associates with the state entered.

Steps:

1. Always start with the initial state

Note: during the conversion of mealy to moore, the number of states may increase. If mealy machine has N states and m outputs, the number of states in the equivalent moore machine may become  $N \times m$

### (b) Moore machine equivalent to Mealy machine

The number of states in the conversion of moore machine to mealy machine will remain same.

## REGULAR LANGUAGES

### Introduction

The examples that we have seen; set of all binary numbers divisible by 2, 3 etc are nothing but regular languages.

FA are **acceptors** of regular languages. They are some mathematical representation to which if you give a string, they will accept it or not. FA (with and without output) are acceptors of regular languages. Acceptors accept “strings” from a language and tell us if the string is accepted or not.

The **generators** of regular languages are regular grammars. They generate the language. They are of two types (Right Linear Grammar and Left Linear Grammar).

A **representator** is a mathematical expression that represents a Regular Language. The RL are mathematically represented by expressions called Regular Expressions.

Let's talk about RE. They are the representations of the languages that are exactly accepted by the FA. Or if we take any language which is accepted by FA, then we can represent it using Regular expressions.

### Operations on Regular Expressions

1. Union (+)
2. Concatenation ( $\bullet$ )
3. Kleene Closure (\*)

Examples:

Regular Expression	Regular Language
$\emptyset$	{ }
$\epsilon$	{ $\epsilon$ }
a	{ a }
$a^*$	{ $\epsilon$ , a, aa, aaa, ... }
$a^+$	$a.a^*/a^*.a/\{a, aa, aaa, \dots\}$
$(a+b)^*$	{ $\epsilon$ , a, b, aa, ab, ba, bb, aaa, aab, aba, ... } (set of all strings possible over (a,b))

Points to remember:

1. If the language is finite, there is a finite automata and regular expression that accept and represent that language.



## Identities of RE

1.  $\emptyset + R = R + \emptyset = R$
2.  $\emptyset.R = R.\emptyset = \emptyset$
3.  $\varepsilon.R = R.\varepsilon = R$
4.  $\varepsilon^* = \varepsilon$
5.  $\emptyset^* = \varepsilon$
6.  $\varepsilon + RR^* = R^*R + \varepsilon = R^*$
7.  $(a + b)^* = (a^* + b^*)^*$   
 $= (a^*b^*)^*$   
 $= (a^* + b)^* = (a + b^*)^* = a^*(ba^*)^* = b^*(ab^*)^*$

The set of regular languages over an alphabet  $\Sigma$  is defined recursively as below. Any language belonging to this set is a regular language over  $\Sigma$ .

## Definition of set of regular languages

- **Basis clause:**  $\emptyset$ ,  $\{\varepsilon\}$ ,  $\{a\}$  for any symbol  $a \in \Sigma$ , are regular languages.
- **Inductive clause:** If  $L_r$  and  $L_s$  are regular language, then  $L_r \cup L_s$ ,  $L_r \bullet L_s$ ,  $L_r^*$  are regular languages.
- **External clause:** Nothing is a regular language, unless it is obtained from above two clauses.

**Regular Language:** Any language represented by regular expression(s) is called a regular language.

Ex: the regular expression  $a^*$  denotes a language which has  $\{\varepsilon, a, aa, aaa, \dots\}$

## Regular Expression

- Regular expressions are used to denote regular languages
- The set of regular expressions over an alphabet  $\Sigma$  is defined recursively as below. Any element of that set is a regular expression.
- **Basis clause:**  $\emptyset$ ,  $\varepsilon$ ,  $a$  are regular expressions corresponding to  $\emptyset$ ,  $\{\varepsilon\}$ ,  $\{a\}$  respectively where  $a$  is an element of  $\Sigma$ .
- **Inductive clause:** If  $r$  and  $s$  are regular expressions corresponding to languages  $L_r$  and  $L_s$  then  $(r+s)$ ,  $(rs)$  and  $(r^*)$  are regular expressions corresponding to the languages  $L_r \cup L_s$ ,  $L_r \bullet L_s$ ,  $L_r^*$  respectively.
- **External clause:** Nothing is a regular expression unless it is obtained from the above two clauses.

**Closure property of RE:** The iteration or closure of a regular expression  $R$ , written as  $R^*$  is also a regular expression.

**CONVERSION OF RE TO FA:** Just look at the RE given to you and bitch just make that FA.

**CONVERSION OF FA TO RE:** The methods used is state elimination method.

Step 1: The initial state should not have any incoming edge. If it is there, create a new initial state.

Step 2: Final state should not have outgoing edges. If it is there, create a new final state. There should be only one final state

Step 3: Eliminate the states other than initial and final states.

## IS THE LANGUAGE REGULAR OR NOT?

To check if the language is regular or not.

1. If language is finite, it is a regular language
2. If the language is infinite, the FA should have a loop and inside the loop there should be a pattern. If there is no such pattern then no FA will be possible to recognize that language. This is what pumping lemma says.

If an infinite language has to be accepted by a finite automata then there should be a loop inside the finite automata. Therefore Pumping Lemma is a negative test. This means that pumping lemma says that the language is not regular if you don't find a pattern but it doesn't say that the language IS regular if you find a pattern. Pumping lemma doesn't guarantee that. Therefore, if you fail the pumping lemma, the language is NOT REGULAR but if you pass it you can't say IT IS REGULAR.

To find if a given language is regular or not, make a RE or a FA out of the language, if it is possible to get, language is regular. In some cases you may need to find that the language does not take infinite memory.

### **CLOSURE PROPERTIES OF REGULAR SETS**

1. **Union:** If L and M are regular languages, LUM is regular language closed under union.
2. **Concatenation and Kleene Closure:** If L and M are regular languages, LM is regular language and  $L^*$  is also regular.
3. **Intersection:** L intersection M is regular, if L and M are regular languages
4. **Difference:**  $L - M$  contains strings in L but not in M, where L and M are regular languages
5. **Complementation:** The complement of language L is  $\Sigma^* - L$ .  
Since  $\Sigma^*$  is surely regular, the complement of a regular language is always regular. Where  $\Sigma^*$  is a universal language.
6. **Homomorphism:** If L is a regular language, h is homomorphism on its alphabet then  $h(L) = \{h(w) \mid w \text{ is in } L\}$  is also a regular language.

### **GRAMMAR**

A grammar is generally represented by  $\{V, T, P, S\}$  where

V = set of all vertices

T = set of all terminals

P = set of all productions

S = start symbol

Ex. A grammar is given as:

$S \rightarrow aSB$

$S \rightarrow aB$

$B \rightarrow b$

for this  $V = \{S, B\}$ ,  $T = \{a, b\}$ ,  $P = \{\text{production formulas given above}\}$ , S = start symbol. So if this is the grammar, let's find out what is the language generated by this grammar.

Getting a string from this grammar is called *derivation*.

$S \Rightarrow aSB$  (sentential/sequential form) (in every step, only one variable will be substituted by the production formula.)

If you start replacing the leftmost symbol first, that is called leftmost derivation. If you start replacing the rightmost symbol first, it is called rightmost derivation.

Let's do the leftmost derivation first.

$S \Rightarrow aaBB$  (sentential/sequential form)

$S \Rightarrow aabB$  (sentential/sequential form)

$S \Rightarrow aabb$  (derivation)

**Derivation Tree:** The yield of the parse/derivation tree is gonna be the string.

## TYPES OF GRAMMARS

Chomsky gave four different grammars;

### Type 3: Regular Grammars

A Regular grammar is the generator of regular languages. These regular languages are accepted by Finite Automata. The Regular languages generated can be represented by Regular Expressions. The Type 3 or Regular Grammar is a Grammar of the form:

$$A \rightarrow \alpha B / \beta \text{ (RIGHT LINEAR GRAMMAR)}$$
$$A, B \in V$$
$$\alpha, \beta \in T$$

or

$$A \rightarrow B\alpha / \beta \text{ (LEFT LINEAR GRAMMAR)}$$
$$A, B \in V$$
$$\alpha, \beta \in T$$

*To check whether the grammar is of Type 3: Give a look at the production formulas and check if all of the productions are either Right Linear or Left Linear.*

### Constructing Regular Grammar from FA

1. FA (L)  $\rightarrow$  Right Linear Grammar (L)  $\rightarrow$  Left Linear Grammar ( $L^R$ )

2. FA(L)  $\rightarrow$  FA( $L^R$ )  $\rightarrow$  Right Linear Grammar ( $L^R$ )  $\rightarrow$  Left Linear Grammar (L)

### Constructing an FA from the Regular Grammar

1. Constructing FA from Right linear grammar is straight forward

2. To construct FA from a left linear grammar

$$LLG(L) \rightarrow RLG(L^R) \rightarrow FA(L^R) \rightarrow FA(L^R)^R$$

### Type 2: Context Free Grammar

The Type 2 or Context Free Grammar is of the form:

$$A \rightarrow \alpha \text{ where } A \in V \text{ and } \alpha \in (V \cup T)^*$$

If a grammar is regular, it is always context free. It is called a context free grammar because (eg.)

$$A \rightarrow aAb / ab$$

If we derive:  $A \rightarrow aAb$

$$\rightarrow aabb$$

I can replace the A in the first production with anything without worrying about the context in which the vertex is present! In context sensitive grammar, I'll have to look at the context of A (aA, Ab, etc.) eg.  $aA \rightarrow aa$

The class of languages generated by context free grammar is called context free languages. The Automata that is the acceptor of context free grammar is the Push Down Automata

### Properties of Type 2 grammar: Context Free Grammar.

The classification of the Context Free Grammar:

1. Context Free Grammar: Ambiguous and non-ambiguous
2. Context Free Grammar: Deterministic and Non-Deterministic
3. Context Free Grammar: Left Recursive and Right Recursive

The entire CFG can be divided into the above mentioned categories. These categories are not mutually exclusive. A CFG can be deterministic and ambiguous or any other combination or combinations from the categories.

The entire discussion on CFG is done in Compiler design so Ravindra has skipped them here. Let's move on to the Properties of CFG.

**Interesting problem: Given a string  $w$ , and CFG  $G$ . Does  $w$  belong to language generated by  $G$ ?**

Derive the grammar in 1 steps, 2 steps, 3 steps ... if the string can be generated in  $n$ -steps then this algorithm will stop after  $n$ -steps. If the string does not belong to the language generated by  $G$ , you may get into an infinite loop. We can solve this by modifying the Grammar.

If we eliminate the epsilon-productions and unit-productions then we'll always increase the string length with every step in the derivation. Therefore after reaching the number of sentential forms which are greater than the  $|w|$  then we can discard all the sentential forms produced until now and conclude that the Grammar does not produce a language that accepts the string ' $w$ '.

In worst case we may have a grammar like this:

$A \rightarrow BC$  (production to a vertex increasing the length of the string)

$A \rightarrow a$  (terminal production)

In worst case the number of steps that we'll have to take to see if the string ' $w$ ' belongs to the language generated by the grammar  $G$  is  $2|w|$ . First  $|w|$  for the number of productions to reach  $|w|$  length and then the putting of terminal will take  $|w|$  steps. (For a grammar which does not have unit or epsilon production). Let's see the algorithms.

In first step:  $P$  productions, or  $P$  sentential forms to examine

In second step:  $P^2$  sentential forms, square because I have to replace the variables with  $P$  possible productions....

.

.

.

$P^{2^w}$

So this algorithm which does not have any epsilon or unit productions is going to take  $O(P^{2^w + 1})$  in order to find out the membership of a string  $w$  in a language generated by a given grammar.

$P + P^2 + P^3 + \dots + P^{2^w}$

So we're not going to be using some exponential algorithm.

***Note: whatever algorithm you're going to use, make sure that your context free grammar does not have epsilon productions and unit productions and should not contain useless symbols.***

One such algorithm is CYK algorithm. Using this you can find out the membership of a string in a language generated by Grammar  $G$  in the order  $O(|w|^3)$ . Before we go to the CYK algorithm, let's see how to eliminate the epsilon and unit productions from the grammar.

### Elimination of $\epsilon$ – productions

Here we are going to eliminate the epsilon productions from the grammar. A question arises: Can we eliminate all the epsilon productions?

Ans: If  $L(G)$  has epsilon in it, then NO. It needs to have at least one production which generated epsilon. This means that if the language does not produce have  $\epsilon$  in it, then you can eliminate all  $\epsilon$ -productions from the grammar. Otherwise  $S \rightarrow \epsilon$  should be a production.

Let's take an example.:

$S \rightarrow aSb/aAb$

$A \rightarrow \epsilon$

**Step 1:** Find out all the null productions (Here it is  $A \rightarrow \epsilon$ )

**Step 2:** Find out all the nullable variables.

Nullable variables are:

if a variable can directly generate  $\epsilon$  ( $A \Rightarrow \epsilon$ ) or

if a variable can after some derivations can generate  $\epsilon$  like ( $A \Rightarrow () \Rightarrow () \Rightarrow \epsilon$ )

That variable is called nullable.

In this example:  $A$  is nullable

**Step 3:** Go to the right hand side of every production and wherever that nullable variable is present, write that production with and without the nullable variable.

$S \rightarrow aSb/aAb/ab$

$S \rightarrow \epsilon$  (now you can eliminate this)

$S \rightarrow aSb/aAb/ab$ . But there is no  $A$  variable involved, so final grammar becomes.

**$S \rightarrow aSb/ab$**

### Example 2:

$S \rightarrow AB$

$A \rightarrow aAA/\epsilon$

$B \rightarrow bBB/\epsilon$

Step 1: Nullables are  $\{A, B, S\}$ .

This means that the start symbol is generating  $\epsilon$ . Therefore,  $\epsilon$  belongs to  $L(G)$ . In this language  $\epsilon$  is present so all the productions may not be eliminated which have epsilon.

Step 2: Let's rewrite the grammar with and without all the nullables

$S \rightarrow AB/B/A/\epsilon$

$A \rightarrow aAA/aA/a$

$B \rightarrow bBB/bB/b$

This is the language which we have derived without the epsilon productions.  $S$  will have an epsilon production because the language has an epsilon in it.

**Example 3:**

$S \rightarrow AbaC$   
 $A \rightarrow BC$   
 $B \rightarrow b/\epsilon$   
 $C \rightarrow D/\epsilon$   
 $D \rightarrow d$

Step 1: Nullables are  $\{C, B, A\}$

Answer:

$S \rightarrow AbaC/baC/Aba/ba$   
 $A \rightarrow BC/B/C$

(here we are not going to include  $\epsilon$  in place of BC because A is not a start symbol. If LHS would have been start symbol that means the  $\epsilon$  is in the language).

$B \rightarrow b$   
 $C \rightarrow D$   
 $D \rightarrow d$

Now when you look at the examples above, eliminating  $\epsilon$  productions are going to add unit productions. Now for the purpose of finding if string 'w' belongs to the language generated by the Grammar G, we need to count the meaningful production steps. For eg.

$A \rightarrow B$   
 $B \rightarrow C$   
 $C \rightarrow d$   
 (3 steps)

is equivalent to saying  $A \rightarrow d$  (1 step)

The length of the string or the number of terminals are also not increasing.

**Elimination of unit – productions**

Let's see how to remove unit productions.

**Example 1:**

$S \rightarrow Aa/B$   
 $B \rightarrow A/bb$   
 $A \rightarrow a/bc/B$

*Note: The language generated by the transformations on the Grammar should not affect the language generated by the grammar. Even if you delete the unit productions, the final language generated should not be affected.*

Step 1: Write the grammar without the unit productions

$S \rightarrow Aa$   
 $B \rightarrow bb$   
 $A \rightarrow a/bc$

Step 2: What are the unit productions, and what the elimination of them have effect, add the effect to the grammar without unit productions

a.  $S \rightarrow B$  in turn  $B \rightarrow bb$  (this be missed if I delete B in the language)

$S \rightarrow B \rightarrow A \rightarrow a, bc$

This implies I need to add  $bb, a, bc$  to  $S \Rightarrow Aa$

Therefore  $S \Rightarrow Aa/bb/a/bc$

b.  $B \rightarrow A$

New B:  $B \rightarrow bb/a/bc$

c.  $A \rightarrow B$

New A:  $A \rightarrow a/bc/bb$

The final grammar becomes

**$S \rightarrow Aa/bb/a/bc$**

**$B \rightarrow bb/a/bc$**

**$A \rightarrow a/bc/bb$**

### Example 2:

$S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow C/b$

$C \rightarrow D$

$D \rightarrow E$

$E \rightarrow a$

$B \rightarrow C \rightarrow D \rightarrow E \rightarrow a \Rightarrow B$  could not reach 'a' if I delete it,  $B \rightarrow b/a$

$C \rightarrow D$ , could miss 'a'  $\Rightarrow C \rightarrow a$  is the final productions

$D \rightarrow E$  could miss 'a'  $\Rightarrow D \rightarrow a$  is the final production

Now  $S \rightarrow AB \Rightarrow C, D$  and  $R$  are not reachable, USELESS SYMBOLS, so let's see how to remove them.

### Elimination of useless symbols

Useful symbols:

1. A symbol 'A' in a grammar is useful if it is able to derive some terminal or string of terminals.
2. This symbol should be reachable from start state.

Test Derivability and Reachability to check for useless symbols.

### Example 1:

$S \rightarrow AB/a$

$A \rightarrow BC / b$

$B \rightarrow aB / C$

$C \rightarrow aC / B$

Step 1: Useful symbols:  $\{a, b, S, A\}$

Step 2: If any production is made of useful symbols, that becomes a useful symbol (left side). Here B and C are useless.

Step 3: Delete all the productions which have useless symbols as the LHS and also delete all symbols containing useless symbols on RHS.

Remaining:

S → a  
A → b

Step 4: check reachability from S to symbols

S → a

**Example 2:**

S → AB/AC  
A → aAb/bAa/a  
B → bbA/aaB/AB  
C → abcA/aDb  
D → bD/aC

Step 1: Useful symbols : {a, b, A, B, S},

Step 2: included in step 1 here

Step 3: eliminate C and D productions, change S production

Final productions before testing reachability:

S → AB  
A → aAb/bAa/a  
B → bbA/aaB/AB

A and B both are reachable from S so the final grammar becomes

**S → AB**  
**A → aAb/bAa/a**  
**B → bbA/aaB/AB**

**Example 3:**

S → ABC/BaB  
A → aA/BaC/aaa  
B → bBb/a  
C → CA/AC

Step 1: Useful symbols: {a, b, B, A, S}

Step 2: in step one included

Step 3: eliminate C as the production, change S and A

Final useful grammar becomes;

S → BaB  
A → aA/aaa  
B → bBb/a

A is not reachable from S alone or through B, so removing A.

Final useful grammar becomes:



$S \rightarrow BaB$   
 $B \rightarrow bBb/a$

## PUSH DOWN AUTOMATA

In order to accept Context Free Languages, we need a machine called Push Down Automata. PDA is nothing but a FA to which a memory element is added and that memory element is stack. A PDA is defined as a sept-tuple  $\{Q, \Sigma, \delta, q_0, Z_0, F, \Gamma\}$  where.

$Q$ : finite set of states

$\Sigma$ : input alphabet

$\delta$ : Transition function

$q_0$ : Initial State

$Z_0$ : Bottom of the stack

$F$ : set of final states

$\Gamma$ : stack alphabet

If the PDA is deterministic:

$\delta: Q \times (\Sigma \cup \epsilon) \times \Gamma \rightarrow Q \times \Gamma^*$  (Automata is in state and sees a symbol or empty symbol and the top of the stack, then it goes to a state in  $Q$  and pushes a symbol in the stack)

If PDA is non-deterministic

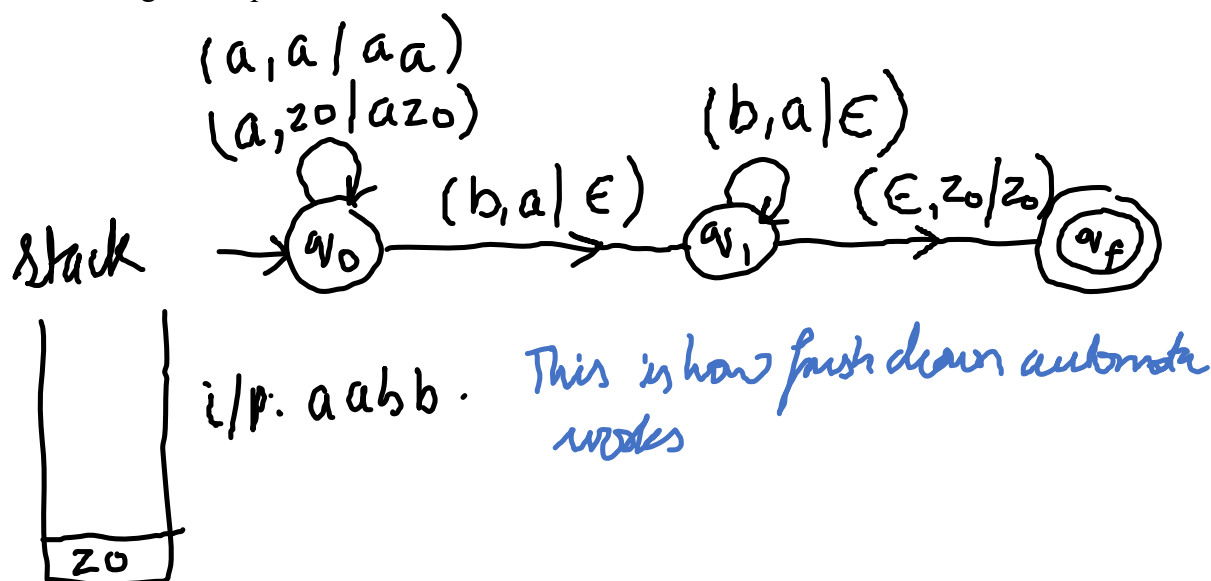
$\delta: Q \times (\Sigma \cup \epsilon) \times \Gamma \rightarrow 2^{Q \times \Gamma^*}$  (This means if you are in some state and see an input or empty symbol and see the top of the stack, you decide to go to more than one state and push more than one symbol to the top of the stack, then it is non-determinism)

Let's take examples to understand.

**Example 1:** Let us say we have language  $a^n b^n \mid n \geq 1$

Here you have to see all  $a$ 's first and then see all  $b$ 's and count  $a$  against  $b$ .

If I have  $aabb$ , then initially  $z_0$  is in the stack. Push ' $a$ ' as you see in the input ' $a$ ' of the string and pop ' $a$ ' as you see input ' $b$ ' of the string. At the end when  $\epsilon$  is reached and the top of stack is  $z_0$ , then the string is accepted. Let's see the PDA for this.



Another way to represent this PDA is using transition function.

Transition function based:

$$\delta(q_0, a, z_0) = (q_0, az_0)$$

$$\delta(q_0, a, a) = (q_0, aa)$$

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

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

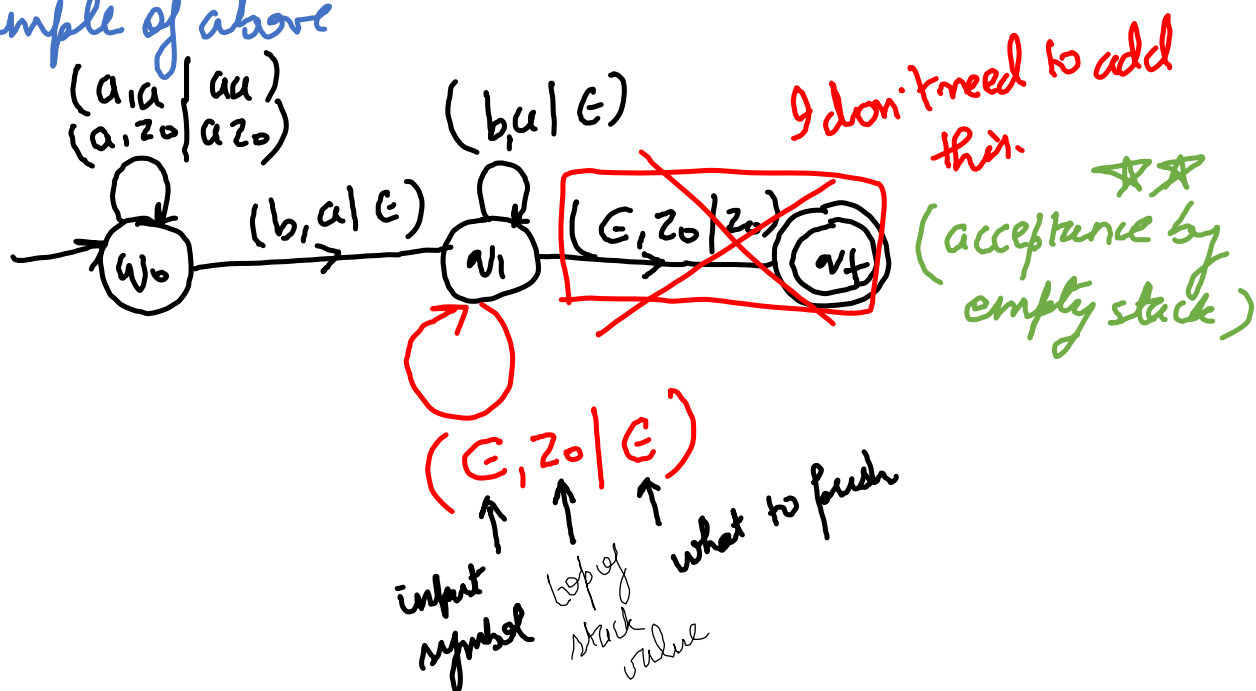
$$\delta(\epsilon, b, z_0) = (q_f, z_0)$$

Push Down Automata acceptance can be of two types:

1. Acceptance by final state
2. Acceptance by empty stack

Both these PDAs are equivalent in power. Either deterministic or non-deterministic, both type of PDAs is equivalent in power. Let's see an example

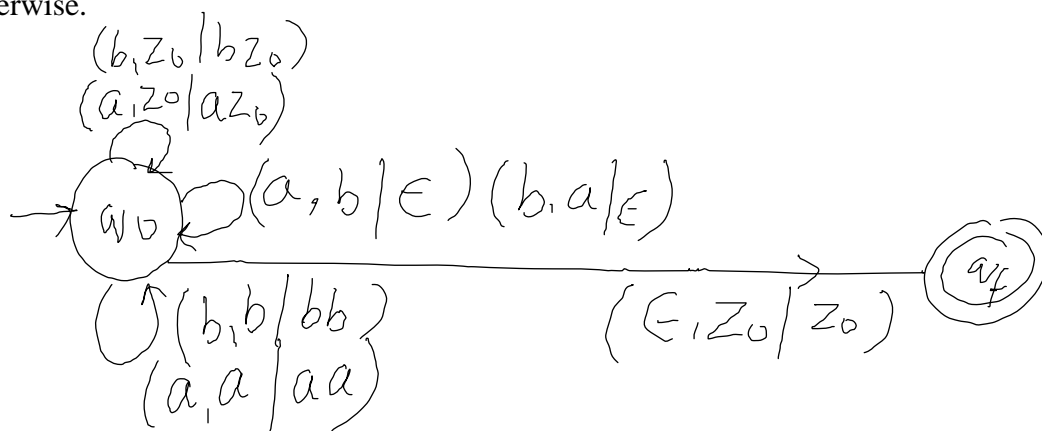
*Example of above*



**Example 2:**  $L = \{w \mid n_a(w) = n_b(w)\}$

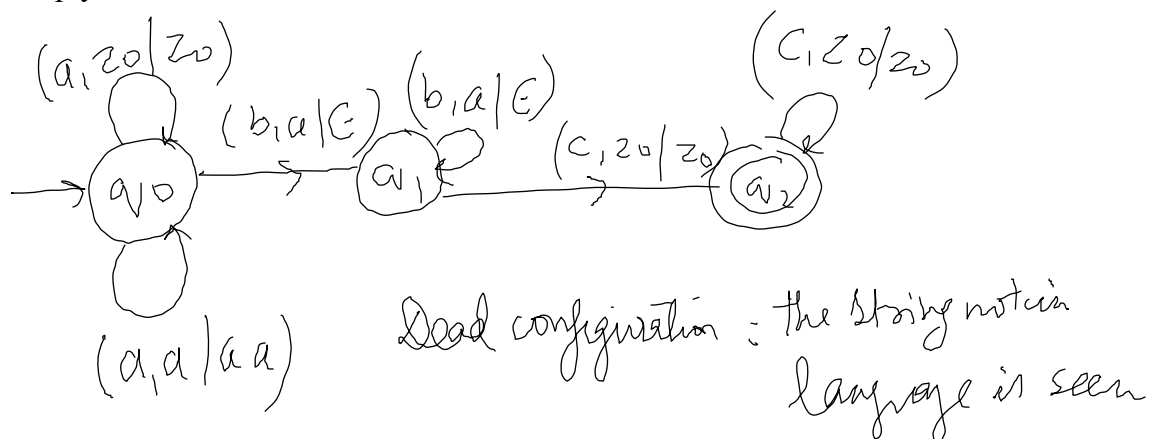
This is like counting all a's and then match them against all b's.

Solution: push if top of stack is empty or the same symbol, pop the stack if the other symbol is seen otherwise.



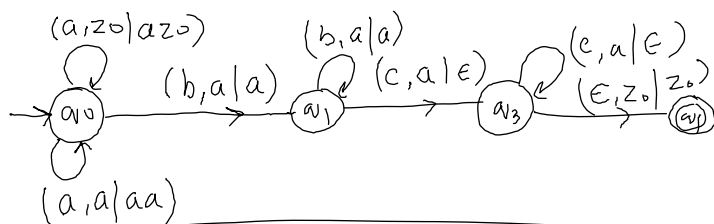
**Example 3:**  $L = \{a^n b^n c^m \mid n, m \geq 1\}$

Here a and b should be of same number and c could be any other number. Compare a and b and then simply see c's

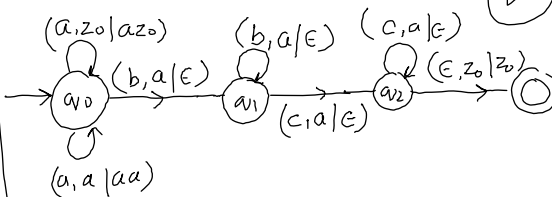


When the input is in the language, it will be accepted by the PDA. If the input is not seen in the language, then it will be considered as dead configuration by the PDA and the PDA will halt in that state. This is halting. This will make PDA stop in a non-final state and it will not be accepted by PDA. If the input is not in the language, the PDA will definitely halt in one of the intermediate state. It will not go to the final state or go in an infinite loop. HALTING will be must.

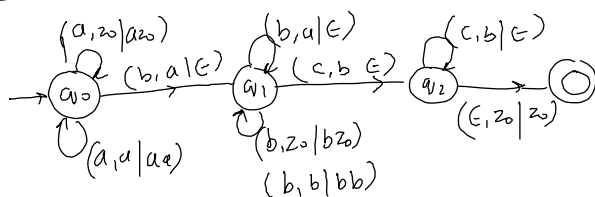
Example:  $L = \{a^n b^m c^n \mid n, m \geq 1\}$  ✓



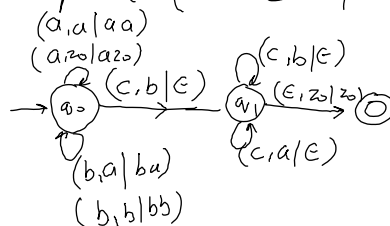
Example:  $L = \{a^{m+n} b^m c^n \mid m, n \geq 1\}$  ✓



Example:  $L = \{a^n b^{m+n} c^m \mid n, m \geq 1\}$



Example:  $L = \{a^n b^m c^{n+m} \mid n, m \geq 1\}$



Some other examples are:

$L = \{a^n b^n c^m d^m \mid n, m \geq 1\}$

$L = \{a^n b^m c^m d^n \mid n, m \geq 1\}$

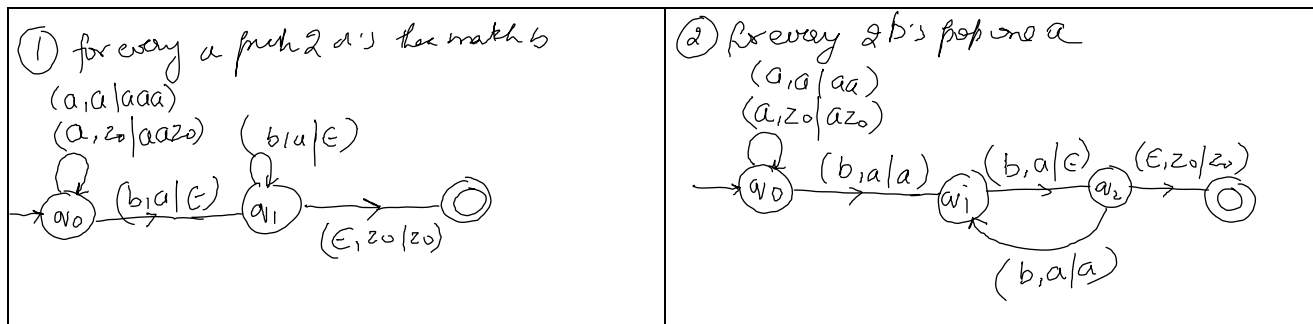
$L = \{a^n b^m c^n d^m \mid n, m \geq 1\}$  (this is not a context free language because PDA will halt).

Another Example:  $L = \{a^n b^{2n} \mid n \geq 1\}$

Here we can make a PDA in two ways.

1. For every a push 2 a's and then match the number of b's
2. For every 2 b's pop one a

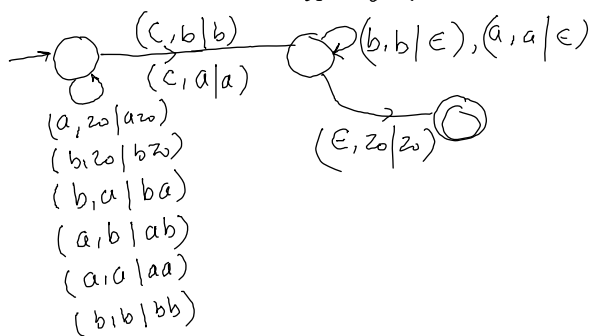
Let's see both of them:



Another example of non- Context free language:  $L = \{a^n b^n c^n \mid n \geq 1\}$

Example:  $L = \{wcw^R, w \text{ belongs to } \{a, b\}^+\}$

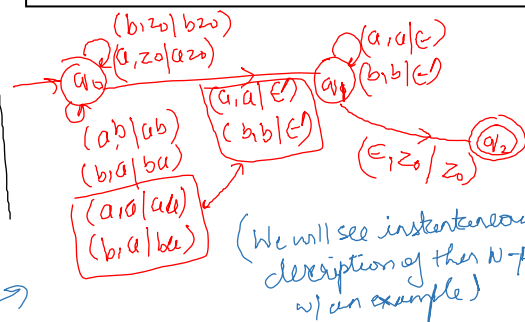
we need to see w then centre and then compare  $w^R$  to w.  
 words can be: abcba, bacab, aacaa, bbcb, etc



Example:  $L = \{w w^R, w \in \{a, b\}^+\}$

Thus is an example of ND-PDA

There is no central element. So one part of the PDA will assume that a given element is centre, other will not



eg. aaaa (instantaneous description of the ND-PDA)

$(q_0, aaaa, z_0)$

$(q_0, aaaa, az_0)$

no-centre | centre

$(q_0, aa, aaz_0)$

$(q_1, aa, z_0)$   
X (dead)

no | yes-centre

$(q_0, a, aaaaaz_0)$  |  $(q_1, a, az_0)$

centre | no

$(q_1, \epsilon, z_0)$

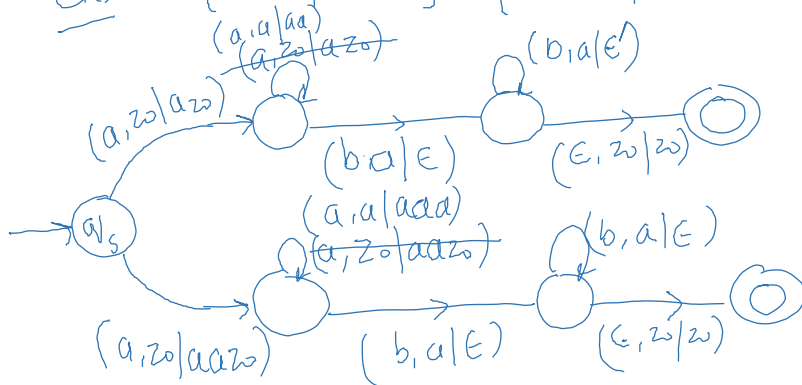
$(q_1, \epsilon, aaaaaz_0)$  |  $(q_1, \epsilon, az_0)$   
X dead | X dead

**This example tells us that there are languages  
That NPDA can accept which DPDA can't. So,  
Their powers are different.**

final

Let's take one more example.

ex:  $L = \{a^n b^n \mid n \geq 1\} \cup \{a^n b^{2n} \mid n \geq 1\}$

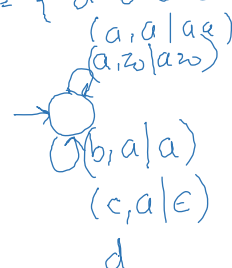


ex:  $L = \{a^i b^j c^k d^l \mid i=k \text{ or } j=l\}$

let's segregate it.

$L_1 = \{a^i b^j c^k d^l \mid i=k\}$

$L_2 = \{a^i b^j c^k d^l \mid j=l\}$



Let's see some examples of some languages and determine if they are regular or not or context free or not.

$a^{m+n} b^n c^m \mid m, n \geq 1$	not regular because of infinite strings & comparison. can give PDA for it so D-CFL
$a^m b^{m+n} c^n \mid m, n \geq 1$	"
$a^m b^n c^{m+n} \mid m, n \geq 1$	"
$a^m b^m c^n d^n \mid m, n \geq 1$	"
$a^m b^n c^m d^n \mid m, n \geq 1$	no PDA for it, so not CFL.
$a^m b^n c^n d^m \mid m, n \geq 1$	CFL not RL because PDA can't be made
$a^m b^i c^m d^k \mid m, n \geq 1$	yes PDA so CFL.
$a^m b^n \mid m > n$	Deterministic PDA is there, so D-CFL
$a^n b^{2n} \mid n \geq 1$	yes CFL
$a^n b^{n^2} \mid n \geq 1$	yes CFL (take eg. of $a^n b^{2n}$ )
$a^n b^{2^n} \mid n \geq 1$	$2^n$ notation AP so not possible PDA; <u>XCFL</u>
$ww^R \mid w \in (a,b)^*$	yes PDA so CFL (it has N PDA so CFL)
$ww \mid w \in (a,b)^*$	not CFL because comparison w/ stack bottom not possible
$a^n b^n c^m \mid n > m$	m will not have anything to compare against. PDA not possible so not CFL
$a^n b^n c^n d^n \mid n < 10^{10}$	finite language, $\therefore$ RL, D-CFL & so CFL too
$a^n b^{2n} c^{3n} \mid n \geq 1$	not CFL.
$x \in y \mid x, y \in (0,1)^*$	it is RL $\Rightarrow$ D-CFL & CFL
$xx^R \mid x \in (a,b)^*,  x  = l$	$2^l$ strings possible, finite, so RL & thus CFL

$www^R \mid w \in (a,b)^*$	not a context free lang.
$a^n b^{3^n} \mid n \geq 1$	not a context free language
$a^m b^n \mid m \neq n$	CFL POA possible.
$a^m b^n \mid m = 2n+1$	CFL, $m$ in AP: 1, 3, 5, 7...
$a^i b^j \mid i \neq 2j+1$	CF, check remaining symbol presence
$a^{n^2} \mid n \geq 1$	$n^2$ not in AP, so no CFL looks not possible
$a^{2^n} \mid n \geq 1$	$\times$ CFL (no AP)
$a^{n!} \mid n \geq 1$	$\times$ CFL (no AP)
$a^m \mid m \text{ is a prime}$	$\times$ CFL (no AP)
$a^k \mid k \text{ is even}$	RL, CFL
$a^i b^j c^k \mid i > j > k$	no CFL
$a^i b^j c^k \mid j = i+k$	CF, DCFL
$a^i b^j c^k d^l \mid i=k \text{ or } j=l$	NDCFL
$a^i b^j c^k d^l \mid i=k \& j=l$	$\times$ CFL
$a^m b^l c^k d^n \mid m, l, k, n \geq 1$	RL $\Rightarrow$ CFL
$a^n b^m \mid n, m \geq 1$	RL $\therefore$ CFL
$\{a^3, a^8, a^{13}, \dots\}$	AP: RL $\&$ CFL
$\{a^{2n+1} \mid n \geq 1\}$	RL, CFL
$\{a^{n^n} \mid n \geq 1\}$	$\times$ RL, $\times$ CFL
$\{w \mid w \in \{a,b\}^*,  w  \geq 100\}$	RL, CFL
$\{w \mid w \in \{a,b,c\}^*, n_a(w) = n_b(w) = n_c(w)\}$	no CFL
$\{w \mid w \in \{a,b\}^*, n_a(w) \geq n_b(w) + 1\}$	CFL

# Turing Machines

Until right now we have seen PDA or FA. The main difference between PDA, FA and Turing machines is that PDA and FA move in only one direction while turning machine can move in both the directions. Turning machine can read and write a symbol on the tape.

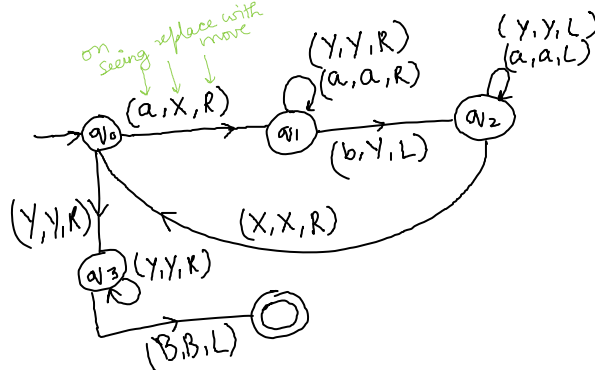
Turning machine:

1. It can move left and right
2. It can do both read and write operations on the tape
3. It may not have all the configurations from a state and still be deterministic
4. If a string is not accepted by a turing machine, the machine will halt in a state other than the final state
5. If a string is accepted by a turing machine, the machine will halt in the final state
6. Turing machine is a hypothetical infinite tape with cells.

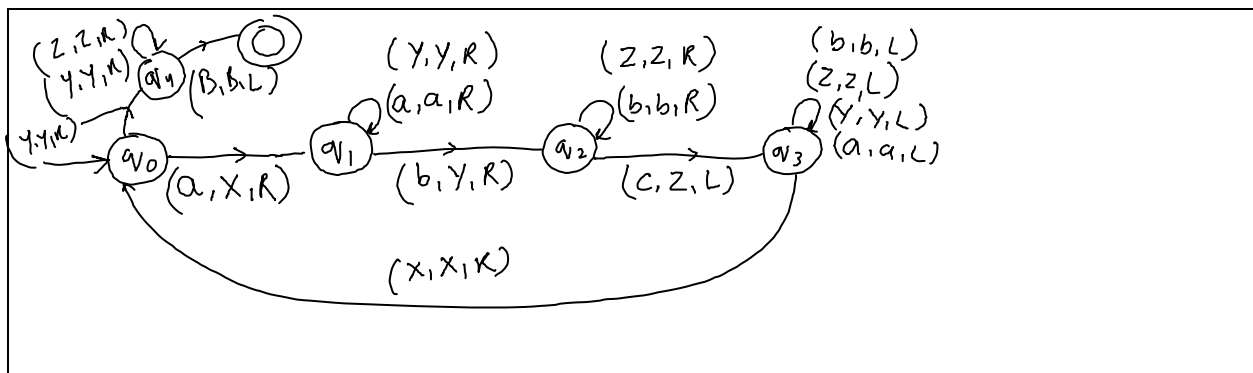
Let's see an example:

Check whether the following string is accepted or not: B = Blank

... B	B	a	a	a	b	b	b	B	B ...
-------	---	---	---	---	---	---	---	---	-------



Let's see one more example:  $L = \{a^n b^n c^n \mid n \geq 1\}$ , make a turning machine for this one.



## Turning machine: A transducer.

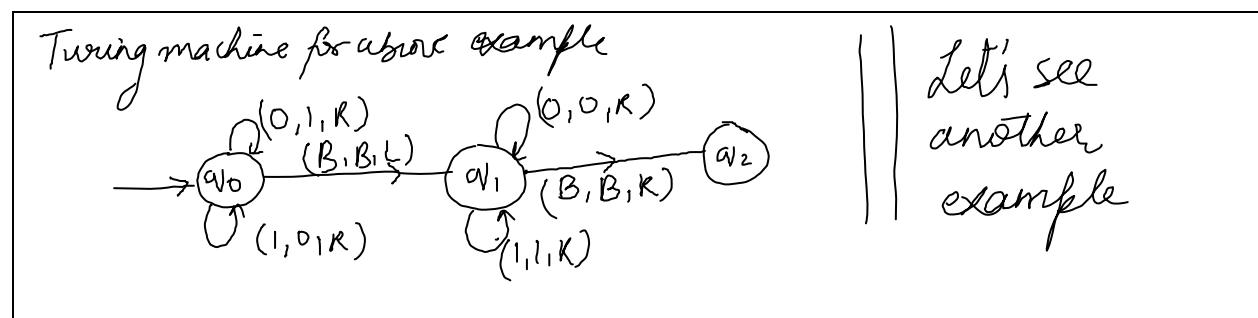
FA and PDA are considered as acceptors. This means that when we send a symbol to these machines, they will tell us if they accept or reject the string or whether the input is present or absent in the language. On the other hand:

Turning machine can act as an acceptor and as a transducer. A transducer converts an input to an output. So turning machines can convert a given input to give an output too. Because turning machine has the capacity to read from the tape and write on it too, it has the capacity to change the symbols on the tape, meaning changing the input to some other output. Let's take an example.

### Example 1: TM to find one's complement of a binary string.

*Note: It is a courtesy to leave the output at the start of the final modified string. Accha nahi lagta ki Blank pe chhod diya. Lol.*

Here the turning machine is not an acceptor of string, therefore there is no need of a final state. It can halt whenever the work has been done.



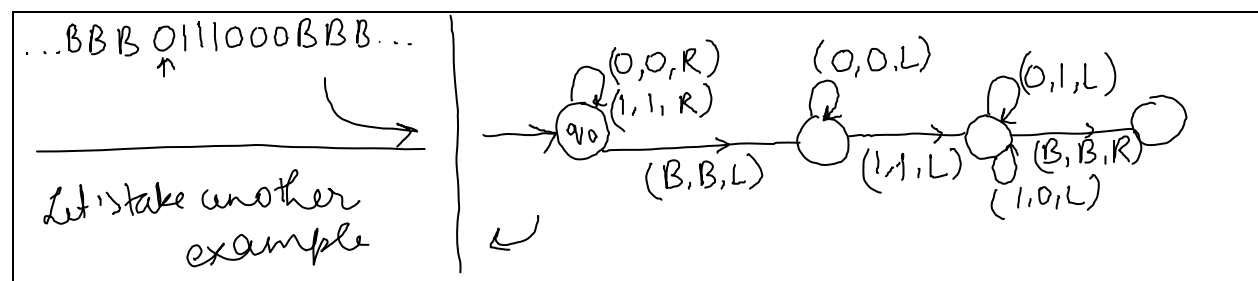
### Example 2: Turing machine to find the 2's complement of a given binary string

Logic: From right, if zeros keep them as zeros, 1<sup>st</sup> one leave it as it is, 1's complement the remaining symbols on the left of 1<sup>st</sup> 1.

String : 0111000

One's : 1000111

Two's: 1001000



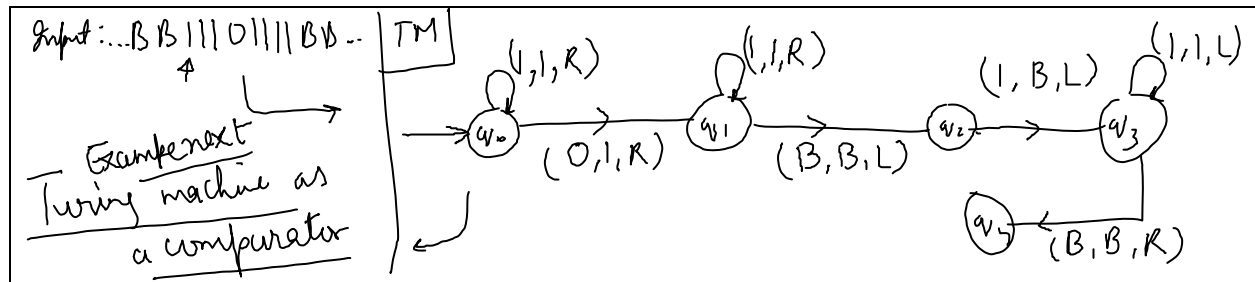


### Example 3: Turning machine as an adder

Let's use unary representation of the number.

$3 = 111$ ,  $4 = 1111$ ,  $5 = 11111$  and so on.

String will be given like: 11101111 (for adding 3 and 4) and we want in the end 1111111 (7) as the answer. Let's do this.

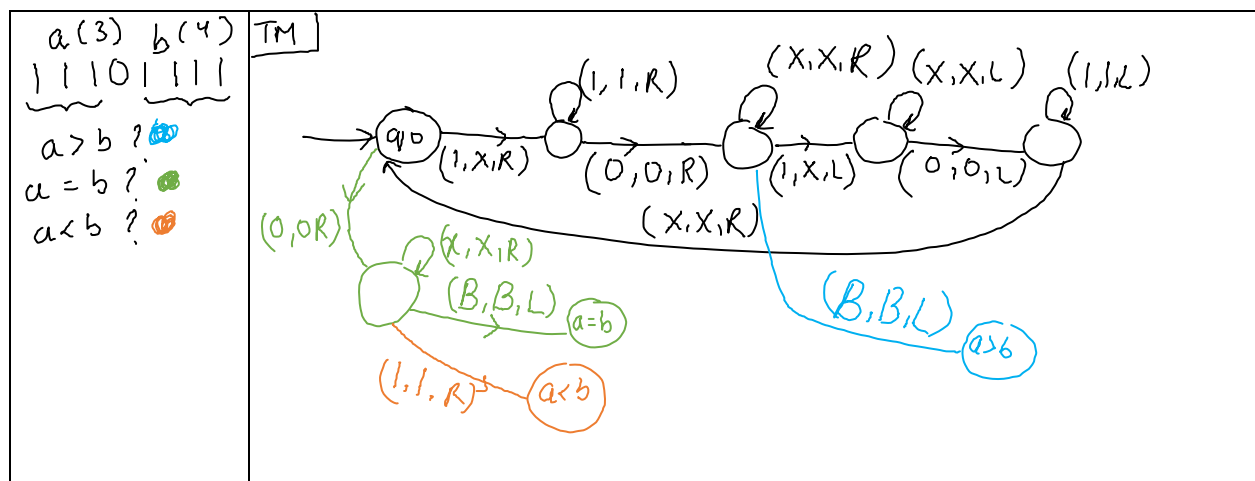


### Example 4: Turning machine as a comparator

$a=111$  (number 3),  $b=1111$  (number 4). The turning machine should give me three things.

1. Whether a and b are equal (should halt in a state that indicates a is equal to b)
2. Whether a is greater than b (should halt in a state that indicates  $a > b$ )
3. Whether a is smaller than b (should halt in a state that indicates  $a < b$ )

Therefore, depending on the input our turning machine should always halt in one of the three states. Let's see how can we do that.



*Note: addition and comparison are the basis operations for any mathematical operation. So by having a cascade of Turing machines you can represent functions like log, exponents, subtraction, division, multiplication by just addition and comparison operations. Therefore, Turing machine is mathematically complete. This means that given any mathematical function,*

*Turing machine will be able to compute it because it is able to do addition and comparison. So Turing machine can perform any mathematical function.*

## STANDARD TURING MACHINE

Let's define the standard Turing machine. "Standard" because there have been various modifications to Turing machines. However, all of them are equivalent to each other.

A standard Turing machine is a sept-tuple defined as

$$M = \{ Q, \Sigma, \Gamma, \delta, q_0, B, F \}$$

$Q$ : Set of states                       $\delta$ : transition function                       $F \subseteq Q$ : set of final states

$\Sigma$ : input alphabet                       $q_0 \in Q$ : initial state

$\Gamma$ : tape alphabet                       $B \in \Gamma$ : used to represent blank

$\delta: (Q \times \Gamma) \rightarrow (Q \times \Gamma \times \{L, R\})$  [this is a partial function, because there are dead configurations, so for some of the states and inputs there is no determinism. We don't define for every symbol and every state]

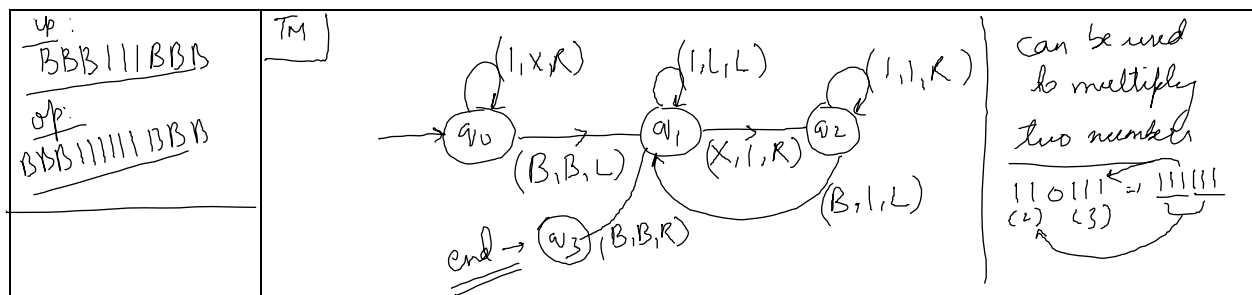
$\Gamma$  is the tape alphabet which consists of all the symbols that can be written on a tape. If the input alphabet is  $\{a, b\}$  then let's say tape alphabet =  $\{a, b, X, Y, B\}$ . Input alphabet is a subset of tape alphabet.

If you want to visualize Turing machines, then take FA add two stacks to it (with one stack it is PDA). If there are two stacks, I can use that machine with a stack or a queue and can do anything. That is a Turing machine. Those two stacks can be merged and seen as a tape.

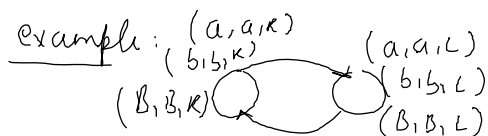
Notes:

1. Tape is unbounded, so any number of left and right moves are possible
2. It is deterministic, i.e., at most one move for each configuration. Looking at the same configuration you will never see two transitions in a Turing machine.
3. No special input or output file. Input and output remain on the tape.

### Example 5: Turing machine as a copier.



**Non-halting Turing machine:** A Turing machine which doesn't halt. It is almost like being in an infinite loop. This kind of behavior is not possible in unidirectional automata like FA or PDA.



This is called halting problem in Turing machines.

### Turing Thesis

Any computation that can be carried out by mechanical means can be performed by a Turing machine. This means Turing machine can solve every problem solved by the said mechanical means.

Some arguments why Turing thesis is accepted as definition of mechanical computation or computer:

1. Anything that can be done by existing digital computer can also be done by Turing machine.
2. No one has yet been able to suggest a problem, solvable by what we intuitively consider an algorithm, for which a Turing machine program cannot be written.
3. Alternative models have been proposed for mechanical computation but none of them are more powerful than the Turing machine model.

### Some modifications to standard Turing machines:

Power of TM: Number of languages a TM can accept.

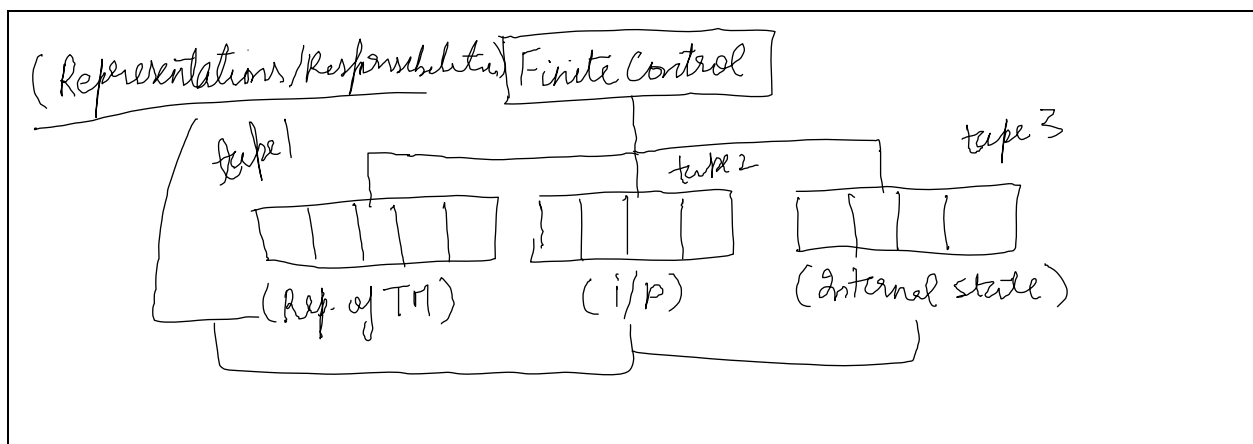
It is seen that the number of languages accepted by a standard TM and modified ones will be same. But, the time taken to accept a language may be different. Which means the number of steps required may increase or decrease.

But changes don't increase the power. Here we are discussing the TM modifications which do not affect the power of the TM.

1. **TM with stay option:** The TM need not move to the left or right, it can also stay in a particular state.  $\delta: (Q \times \Gamma) \rightarrow (Q \times \Gamma \times \{L, R, S\})$
2. **TM with semi-infinite tape:** The tape is not infinite on both the side.
3. **Offline TM:** Input in separate file which is read-only. So finite control of TM can read the symbol from the input and write on the TM tape. If modifications required in the input, copy the input on the tape and then do the modifications on the tape.
4. **Multi-tape TM:**  $\delta: (Q \times \Gamma^n) \rightarrow (Q \times \Gamma^n \times \{L, R\}^n) \mid n = \text{number of tapes we have.}$
5. **Jumping TM:** Instead of moving one step ahead it can go any number of steps to the right.  $\delta: (Q \times \Gamma) \rightarrow (Q \times \Gamma \times \{L, R\} \times \{n\}) \mid n = \text{steps}$
6. **Non-erasing TM:** Standard TM can change the input to a blank. In non-erasing TM you can't do that.
7. **Always writing TM:** In standard TM we have liberty to leave the symbol as it is. Here you are supposed to definitely change the symbol machine has read to some other symbol.
8. **Multidimensional TM:**  $\delta: (Q \times \Gamma) \rightarrow (Q \times \Gamma \times \{L, R, U, D\})$ . In our tape, instead of having 2 dimensions you can also go up and down which is respectively U and D.
9. **Multihead TM:** The finite control can look at various places simultaneously.
10. **Automata with a queue:** There is a finite automata with queue.
11. **TM with only 3 states**
12. **Multitape Tm with stay option and at most 2 states**
13. **Non-deterministic TM:**  $\delta: (2^{Q \times \Gamma}) \rightarrow (2^{Q \times \Gamma} \times \{L, R\})$
14. **A NPDA with two independent stacks:**  $\delta: (Q \times (\Sigma \cup \epsilon) \times \Gamma \times \Gamma) \rightarrow 2^{Q \times \Gamma^* \times \Gamma^*}$

## UNIVERSAL TURING MACHINE

A universal Turing machine has a finite control but it is going to have 3 tapes. So it is a multi-tape Turing machine. Let's see what these three tapes do.



The first tape is the entire representation of the Turing machine which UTM is trying to accommodate.

The second tape is the input that the Turing machine should work on.

The third tape is the manager of the internal states.

**Example: Let's say that you want to add two numbers, 2 and 3.**

The input will be 110111 and that will be saved on the second tape of the universal Turing machine. The representation of the Turing machine which will only add two numbers (not the UTM) will be saved in the first tape. The internal state governed by the transition function will be saved in the third tape. The first symbol 1 (from 110111) will be read from the second tape. The third tape will have the initial state  $q_0$  so according to the transition function  $\delta(q_0, 1) = (q_2, x, R)$ , the symbol  $x$  will be written in the second tape and the third tape will have the state  $q_2$  now. This transition is governed in the first tape which holds the representation of the Turing machine which would add two numbers.

The representation of the Turing machine that does one thing has to be saved in tape 1. This sounds like a mystery, so how are we going to do that?

We assume that the Turing machine has

$Q = \{q_0, q_1, q_2, q_3, q_4, \dots\}$  (set of all states)

$\Gamma = \{a_0, a_1, a_2, a_3, a_4, \dots\}$  (tape alphabet)

Then you have to encode all the symbols, let's say the encoding is as follows:

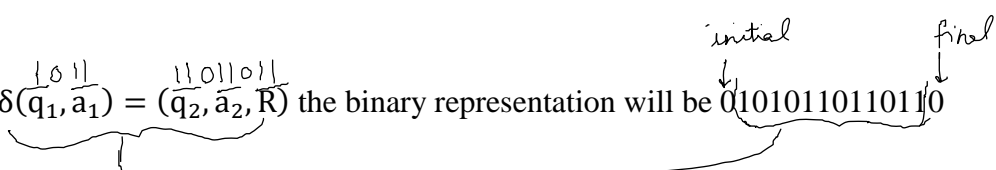
$q_1 \rightarrow 1, q_2 \rightarrow 11, q_3 \rightarrow 111, q_4 \rightarrow 1111$ , similarly

$a_1 \rightarrow 1, a_2 \rightarrow 11, a_3 \rightarrow 111, a_4 \rightarrow 1111$

$R \rightarrow 11, L \rightarrow 1$

'0'  $\rightarrow$  separator

So for a transition like  $\delta(q_1, a_1) = (q_2, a_2, R)$  the binary representation will be 01010110110110



Therefore, every Turing machine can be encoded as a string of 0's and 1's. (very important). These strings of 0's and 1's for all transitions possible will be kept in the Tape 1.

So if I'm going to represent a Turing machine in terms of 0's and 1's then I can say that any possible Turing machine will be an element of:  $\Sigma^*, \Sigma \in \{0,1\}$ . But this does not mean that every element of this infinite set is going to be a Turing machine. Please know the following statements.

*Statement 1: Every Turing machine can be represented as a string of 0's and 1's*

*Statement 2: Not every string of 0's and 1's is a Turing machine.*

## LINEAR BOUNDED AUTOMATA

We know that Turing machines have been modified. All of these Turing machines that we saw were equivalent to each other, i.e., the languages accepted by all of the modified Turing machines is same. Let's take some more modifications.

1. A non-deterministic Turing machine with a tape that acts as a stack: this will be equivalent to the PDA.
2. A Turing machine with a finite tape: This will act as our Finite Automata.

Here, we are essentially reducing the power of a TM by putting some limitations on the tape it can operate on. Let's see one more modification:

*A Turing machine with the size of the tape which is equal to the size of the input.*

Now we are putting a boundary on the tape size. This particular result of putting a boundary gives birth to a new machine which is called *Linear Bounded Automata*.

A linear bounded automaton (LBA) is powerful than FA and PDA. Remember the example that  $L = \{a^n b^n c^n \mid n \geq 0\}$  is a language which can be accepted by LBA but not a PDA. For LBA to be more powerful than the PDA, it should accept all the languages accepted by PDA. This can be proved by making the tape of LBA as the stack of PDA and a PDA will not require the stack size more than the input to accept a string. Therefore, every PDA can have an equivalent LBA but the inverse is not true. Thus, LBA is powerful than PDA.

The boundary indicators  $<$  and  $>$  in the LBA tape are called endmarkers.

So the power dynamics are:

- $FA < PDA < LBA < TM$

The power of the deterministic and non-deterministic machines in:

- FA will be equal
- PDA will not be equal
- LBA *we don't know as no one has found it out yet.*
- TM will be equal

Let's see some general examples of languages which can be accepted by a LBA but not by a PDA.

$$L = \{a^n b^n c^n d^n, n \geq 1\}$$

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

$$L = \{a^n : n = m^2, m \geq 1\}$$

$$L = \{a^n : n \text{ is a prime}\}$$

$$L = \{a^n: n \text{ is not a prime}\}$$

$$L = \{ww: w \in \{a, b\}^+\}$$

$$L = \{w^n; w \in \{a, z\}^+, n \geq 1\}$$

$$L = \{www^R: w \in \{a, b\}^+\}$$

These are some standard examples of languages which are accepted by LBA apart from all the languages which are accepted by a PDA. So what is the set of all languages accepted by Turing machines?

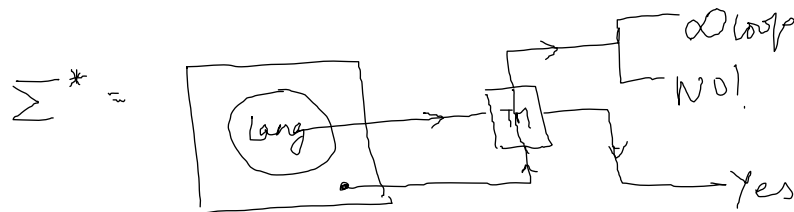
## LANGUAGE ACCEPTED BY TURING MACHINES

We saw Regular Languages are accepted by FA and Context Free Languages are accepted by PDA. The languages accepted by a TM are called *Recursively Enumerable Languages*.

Since languages accepted by PDA are also accepted by TM, therefore CFLs are Recursively Enumerable.

Turing machine cannot accept  $\epsilon$  but a PDA does. So when I say CFLs are a subset of RELs (recursively enumerable languages) then I am talking about the ones which don't have  $\epsilon$ . Also  $\epsilon$  don't make difference.

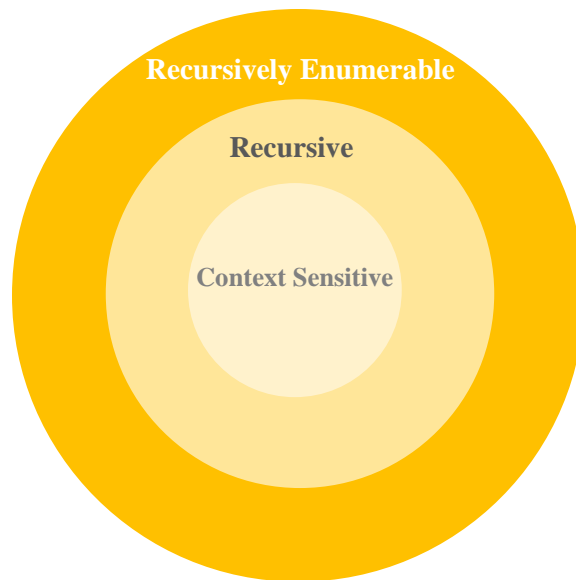
Now when a string is the language the Turing machine will accept, say Yes and halt in a finite state. If the string is not in the language the TM will say No or *it will enter an infinite loop*.



So whenever the TM enters an infinite loop, we don't know whether to wait or to halt the Turing machine. Now, I don't want such a dilemma. I want a TM that says YES or NO! So, if I restrict a Turing Machine in such a way that it says only Yes or No, such a Turing machine is called **HALTING TURING MACHINE**. Therefore, a HTM will always halt no matter the language given to it.

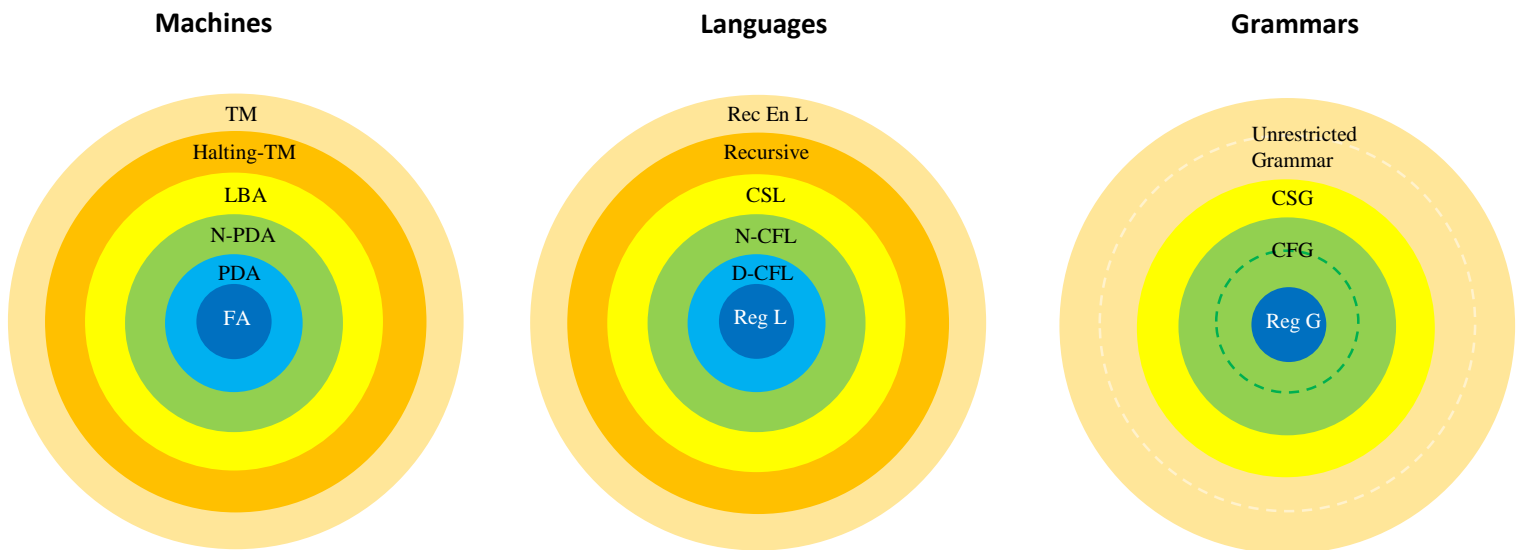
*Note: The languages accepted by **Turing Machines** are called **Recursively Enumerable Languages** and the languages accepted by **Halting Turing Machines** are called **Recursive Languages**. Recursive languages are a subset of Recursively Enumerable Languages*

*Moreover, the languages accepted by **Linear Bounded Automata** are called **Context Sensitive Languages**. LBA is a halting Turing machine. The relationship between these languages is:*



Now let's see what different types of machines we have seen, what are the languages associated with them and what are the grammars associated with them.

## THE BIG PICTURE



Let's see them one by one and learn about their properties. By the way:

- Unrestricted grammars are also called Type 0 Grammars
- Context-sensitive grammars are also called Type 1 Grammars
- Context-free grammars are also called Type 2 Grammars
- Regular grammars are also called Type 3 Grammars



## UNRESTRICTED GRAMMAR

The languages corresponding to TM are Recursively Enumerable Languages and the generator of this language is the Unrestricted Grammar.

A grammar is called unrestricted if all the productions are of the form:

$$u \rightarrow v, u \in (V \cup T)^+, v \in (V \cup T)^*$$

This means the LHS of the production rule can have anything except  $\epsilon$  meaning the null string should not derive any string. The right hand side is completely unrestricted.

Example: What language does the following unrestricted grammar generates?

$$S \rightarrow S_1 B$$

$$S_1 \rightarrow a S_1 b$$

$$b B \rightarrow b b b B$$

$$a S_1 b \rightarrow a a$$

$$B \rightarrow \lambda$$

## CONTEXT SENSITIVE GRAMMAR

A grammar is said to be context sensitive if all the production formulas are of the form:

$$x \rightarrow y: x, y \in (V \cup T)^+$$

and

$$|x| \leq |y|$$

This means that the LHS and RHS can't have  $\epsilon$  and there will never be a contraction from LHS to RHS. This means at every step of the derivation the string length either remains same or increases but *never decreases*.

Now, it is called context-sensitive (let's understand with example):

In context free we see:  $A \rightarrow a$  (here the variable A can be replaced by a terminal 'a' independent of the 'context' or extra information around A)

In context sensitive:  $aAb \rightarrow aab$  (here we are not replacing the variable A independently to 'a'. The A will only be replaced with 'a' terminal *iff* A occurs with 'a' and 'b' as prefix and suffix respectively so  $A \rightarrow a$  is not a production rule here)

Example: What is the language generated by the following CSG.

$$S \rightarrow abc/aAbc$$

$$Ab \rightarrow bA$$

derivation : (1)  $S \Rightarrow \boxed{abc} \rightarrow \underline{a'b'c'}$

(2)  $S \Rightarrow aAbc$

$\Rightarrow abAc$

$\Rightarrow abBbcc$

(3)  $S \Rightarrow aAbc$

$\Rightarrow abAc$

$\Rightarrow abBbcc$

$\Rightarrow aBbbcc$

$\Rightarrow \dots$

$Ac \rightarrow Bbcc$   
 $bB \rightarrow Bb$   
 $aB \rightarrow aa/aaA$

Language:  $\{a^n b^n c^n \mid n \geq 1\}$

$\Rightarrow aB b b c c$   
 $\Rightarrow aaA b b c c$   
 $\Rightarrow aa b b A c c$   
 $\Rightarrow aa b b B b c c c$   
 $\Rightarrow aa b B b b c c c$   
 $\Rightarrow aa B b b b c c c$   
 $\Rightarrow aa a b b b c c c$   
 $= a^3 b^3 c^3$

## IMPORTANT THEOREM ON RECURSIVE AND RE LANGUAGES

Theorem:

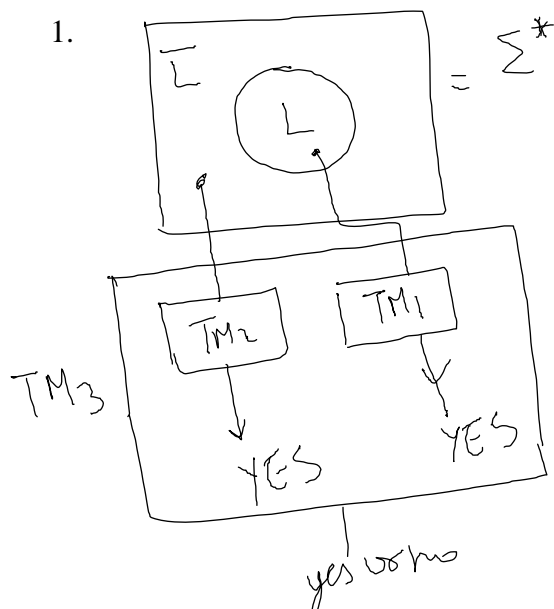
1. If a language  $L$  and its complement  $L^c$  are both recursively enumerable, then both languages are recursive.
2. If  $L$  is recursive, then  $L^c$  is also recursive and consequently both are recursively enumerable.

The difference between TM and HTM we already know. The *membership algorithm* also asks the same: *Given a string and a language, whether the string belongs to a language or not?* If there is an algorithm that can answer the above question that is called a membership algorithm.

So, *for Recursively Enumerable languages, the membership algorithm does not exist.*

But, *for the Recursive Languages, the membership algorithm exists and answers the membership question with a YES or NO!*

Let's understand the theorems.

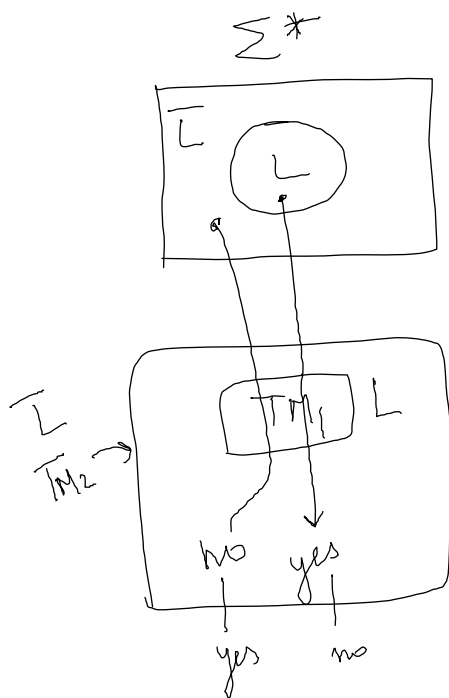


The theorem says that if the language  $L$  and its complement  $L^c$  both are recursively enumerable. This means that the language  $L$  and  $L^c$ , both can be represented by two separate Turing machines. Now if a language is accepted by a Turing machine it is always going to halt.

Let's create a Turing machine consisting of both TM1 and TM2 which accept  $L$  and  $L^c$  languages respectively (TM3). Now this Turing machine can take any string from the  $\Sigma^*$  and come to a YES or NO outcome with a halt.

Thus  $L$  and  $L^c$  are both recursive.

2. Let's take a look at the second point of the theorem.



The second statement says that *if  $L$  is recursive, then  $L^c$  is also recursive*. This means that there is a HTL for  $L$  and if we create a HTL which complements the output of TM1, then this TM2(diagram) will represent an HTL for the language  $L^c$ .

Thus *consequently both are recursively enumerable* because RE just wants a Turing machine, it doesn't care if it halts or not.

# Countability, Computability and Decidability

We saw the machines, languages and grammar and we can also see that each smaller language is a proper subset of the circle that encapsulates it. To see if CSL is a proper subset of Recursive language, we have mathematical proofs but no concrete example of a language. Same goes for when we need to show that Recursive language is a proper subset of Recursively Enumerable Languages.

Now a general question we may ask: Is Turing Machine capable of accepting any language? Or for any language is there a Turing machine accepting it?

This is tough to answer directly, but we are going to use Set Theory for an indirect proof.

*We are going to take a set of all the languages and another set of all the Turing machines. Both will be infinite, but we are going to prove that the set of TMs will have a smaller cardinality than the set of languages. Therefore, not all of the languages will have an associated TM.*

**Set → Finite**

→ **Infinite → Countable**

→ **un-countable**

**Countable sets:** A set 'S' is said to be countable if all the elements of the set can be put in one-to-one correspondence with the set of natural numbers. The natural number corresponding to the elements of the set in question is called *index* of the element.

**Uncountable sets:** A set is uncountable, if it is infinite and not countable.

Example of countable: Set of all even numbers, set of all odd numbers.

Example of uncountable: Set of all real numbers

## Alternative definition of countability

A set is said to be countable if there exists an enumeration method using which all the elements of the set can be generated and for any particular element, it takes only finite number of steps to generate it. The finite number of steps taken to generate the element can be as its index and hence mapping into natural numbers.

Let's take some examples:

**Example 1:**  $\frac{p}{q}$ :  $p, q \in \mathbb{Z}_f$  or set of all quotients  $p/q$  where  $p$  and  $q$  belong to set of all positive integers.

Set:  $\{1/1, 1/3, 1/5, 1/6, \dots\}$  now we need to find enumeration method.

If we take  $1/1, 1/2, 1/3, 1/4 \dots$  etc then  $2/1$  will never be able to reach. Wrong enumeration

Reciprocal will also be invalid enumeration.

Go in the increasing order of  $p$  and  $q$  and arrange them.

1/1, 1/2, 2/1, 1/3, 2/2, 3/1, ... then we can give index.

*Therefore, this set is countable.*

**Example 2:**  $\Sigma^*$ :  $\Sigma = \{a, b\}$  or set of all the strings over a finite alphabet.

$\Sigma = \{\epsilon, a, b, aa, bb, aabb, \dots\}$

Find the enumeration method: Follow the dictionary order? Wrong enumeration. 'b' will never come

Enumeration method: Arrange in increasing length.

*So this set is countable as an element can be reached in finite number of steps.*

1. *Note: every infinite set of strings from finite set of alphabets is countable.*
2. *Note: This method of arranging in alphabetical order sorting in the increasing string length sorting is called **Proper Order**.*

**Property: Every subset of a countable set is either finite or countable.**

*Now we know that  $\Sigma^*$  is countable. A language is a subset of  $\Sigma^*$ . Therefore, every language is countable. It can be finite or countable.*

**Example 3:** Set of all Turing machines are countable. (Important)

Let's take  $\Sigma = \{0,1\}$  then  $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, \dots\}$  and we know that every Turing machine can be encoded as a string of 0's and 1's (Universal Turing Machine)

1. Now  $\Sigma^*$  is countable.
2. Therefore, let's say set of all Turing Machines is S. Then  $S \subseteq \Sigma^*$
3. Every subset of countable set either finite or countable.

Therefore, using 1, 2, and 3, the set of all TMs S is countable.

**Implications of the fact that set of all Turing machines is countable**

1. Since the set of Turing machines are countable, the set of all Recursively Enumerable Languages is also countable.
2. Using 1 all Recursive Languages are countable.
3. Using 2: all Context Sensitive Languages are countable
4. Using 3: all Context Free Languages are countable.
5. Using 4: all Regular Languages are countable.

Using 2, 3, 4, 5 above we can say that the set of all LBA, set of all PDA (deterministic and non-deterministic), and set of all FA (deterministic and non-deterministic) are countable.

**Diagonalization method to prove that set of all languages are uncountable:** We have  $\Sigma = \{a, b\}$ ,  $\Sigma^*$  is countable.  $\Sigma^*$  is the set of all possible strings on 'a' and 'b'. Subset of  $\Sigma^*$  is a language. Now the power set of  $\Sigma^*$  is the set of all the subsets of  $\Sigma^*$  which will be  $2^{\Sigma^*}$ . Let's see if  $2^{\Sigma^*}$  is countable or not.

**Statement:** If a set 'S' is countably infinite,  $2^S$  is uncountable.

We have to prove above statement.

$$\Sigma^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$$

$$L = \{a, aa, bb\} \text{ Representation in } \Sigma^* = \{0, 1, 0, 1, 0, 0, 1, 0, \dots\}$$

So, I can represent every language as a string of 0's and 1's from  $\Sigma^*$ .

**Assumption :** Let us assume that  $2^{\Sigma^*}$  is countable.

**Work:** If any set S is countable, it has an index and depending on that index I can put all the elements in a table which looks like this.

Index	Representation of an element in $\Sigma^*$
1	0100101...
2	1010010...
3	0101010...
4	0101010...
5	1010101...
6	0111011...
7	1001010...

Now if an element belongs to  $2^{\Sigma^*}$  then that element should be in a table like above corresponding to some index value in  $\mathbb{N}$ .

Now we have to look at the diagonal values of the representation of elements in the above table (marked in red). The values are: 0001000...

Diagonal Values: 0001000

Now we need to complement the values: 1110111...

Now these complemented values will differ with all of the languages in at least one bit. These bit (0 and 1) are taken from the list of languages. The diagonal elements when complemented will give a new language which will always be different from all the languages in this table. Therefore we can't reach this new language through an enumeration or can't give a natural numbered one-to-one correspondence to it.

Thus, our assumption is wrong and the set  $2^{\Sigma^*}$  is not countable.

**Verdict:** Set  $2^{\Sigma^*}$  is not countable or uncountable infinite set.

Now we already saw that the set of all Turing machines is countable. This means that the cardinality of that set is less than the cardinality of the set  $2^{\Sigma^*}$ .

*Therefore, for all the languages possible, there will not be a one-to-one Turing machine map possible. This means that there must be some languages which are not Recursively Enumerable.*

Let's see some properties, may be asked (remember them):

1. If S1 and S2 are countable sets, then  $S1 \cup S2$  is also countable and  $S1 \times S2$  is also countable. (run the enumeration process of S1 and S2 alternatively to get enumeration for  $S1 \cup S2$ . Therefore, this statement is correct, this can be extended to finite number of sets S1 S2 S3..Sn).

2. The Cartesian product of finite number of countable sets is countable

We can enumerate on  $S_1 \times S_2$  using the indices values.

Ex:  $S_1 = \{a_1, a_2, a_3, \dots\}$  (it is countable (given) so there are indices like 1, 2, 3... for the elements) 1 2 3

$S_2 = \{b_1, b_2, b_3, \dots\}$  (it is countable (given) so there are indices like 1, 2, 3... for the elements) 1 2 3

Now  $S_1 \times S_2$  is countable when we have index value for each of the element of this new set. So I'm gonna add the indices and see which elements can come in this new set.

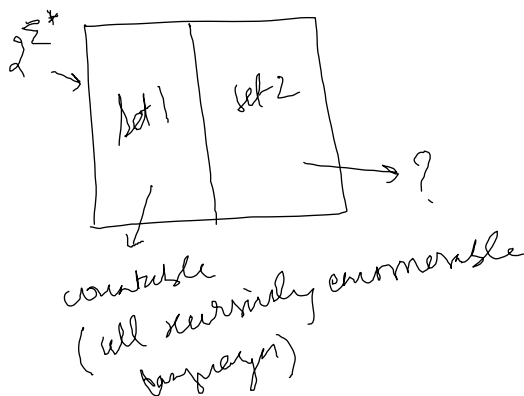
for a sum of 2 indices possible: (1, 1) : ordered pairs possible (a<sub>1</sub>, b<sub>1</sub>)

for a sum of 3 indices possible: (1,2), (2,1): ordered pairs possible: (a<sub>1</sub>, b<sub>2</sub>) (a<sub>2</sub>, b<sub>1</sub>) etc.

$S_1 \times S_2 = \{(a_1, b_1), (a_1, b_2), (b_2, a_1), \dots\}$  (which is countable)

3. The set of all languages that are not recursively enumerable is uncountable.

We know that set  $2^{\Sigma^*}$  is uncountable. Now let's divide this set into languages which are RE and languages which are not like below.



Now we know that Set 1 is countable because Turing machines are countable. Let us assume set 2 is countable. Then it means:

$$S_1 \cup S_2 = 2^{\Sigma^*}$$

This means  $2^{\Sigma^*}$  should be countable is  $S_1$  and  $S_2$  are both countable.

Therefore, our assumption is wrong.

$\Rightarrow S_2$  is not countable. (Proved)

# Computability and Decidability

Computability and decidability are same in the sense, both of them talk about if there exists an algorithm or not. Before going ahead, let's see what an algorithm is and what is the difference between an algorithm and a Turing machine.

An algorithm is a Turing machine that is supposed to halt. So for a problem if there is an algorithm it is as good as saying that for that problem there is a Turing machine that is supposed to halt. But if there is a Turing machine it doesn't mean that is an algorithm. A TM that is an algorithm is supposed to halt after finite number of steps. Therefore, only Halting TMs are considered as algorithms.

Now we know that TMs are countable. HTMs are subset of TMs. Therefore, HTMs are countable. And this Algorithms are also countable.

Now, set of all languages is uncountable, every problem can be compared to a language. This means that the number of problems will be uncountable and number of algorithms will be countable. Therefore, there are some problems for which there will be no algorithm.

**Computability and Decidability** are very identical but with a minor difference.

## Computability:

Let us take a function  $f(n) = n^2 + 1$ . The function will have a domain and a range. If I give this function some value 'n', it is going to give out a value by computing  $n^2 + 1$ .

If there is an algorithm to compute the function, such a function is called computable. An algorithm is a HTM we know that, so we can say:

If there exists a Turing Machine and we give an input (n) to its tape and it gives the output  $f(n)$  on the same tape and then it halts for every input 'n' given. Such a function is called computable.

**Definition:** If there is a function defined on a domain and there is a Turing Machine which will produce the output on the tape given the input on the tape and halts for every input chosen from the domain. Such a function is computable.

**Decidability:** Let us take a problem (a statement which has true or false as the answer).

Example : Problem: If 'n' is prime number

Domain: Set of all natural numbers (chosen by us)

Answer to be determined: Yes or No

*Note: Therefore, the only difference between Computability and Decidability is that we talk about the function in computability and we talk about the problem in Decidability.*

**Definition:** If there is a problem for which there exists an algorithm/halting Turing machine and this TM takes anything from the domain and decides if the problem has Yes or No as the answer followed by halting. Then that problem is decidable.



**Note:** A large domain can be collectively undecidable but an instance of the set will always be decidable. A common domain may be decidable or undecidable. Because there will be a Turing machine that will say Yes or No for a particular instance but may not find such decidability for a large domain.

**Example:** Problem: Given grammar 'G' is ambiguous

Domain: Set of all Context Free grammar.

This problem is undecidable. But if we take an instance of domain, let's say, Grammar 'G1' then it is a decidable problem.

**In TOC you will get questions on decidability. Let's see some of them. We use an indirect method to answer decidability. To say a problem is undecidable or decidable, we talk about reducibility.**

Let's say there is a problem  $P_1$  and there is a conversion algorithm or HTM that converts it to  $P_2$ . Now if  $P_2$  has an "algorithm" or halting Turing machine then  $P_1$  is decidable. This conversion of one problem to another problem is called *reducibility*.

1. Therefore, if  $P_2$  is decidable,  $P_1$  is decidable.
2. If it is already proved that problem  $P_1$  is undecidable, then the problem  $P_2$  must be undecidable.

Starting point for the theory of undecidability is Halting Problem.

Intuition: Take the halting problem -> reduce it to some problem  $P_1$   $P_2$   $P_3$  etc. All of them will be undecidable.

## **TURING MACHINE HALTING PROBLEM**

This is the most interesting problem in computer science. This is the basic point which is used to prove that other problems are undecidable.

**Statement:** *Given the description of a Turing machine 'm' and an input 'w', does 'm' when started with 'w' as its input eventually halts?*

The undecidability of the above statement is proved using the theorem below.

**Theorem:** *If the halting problem were decidable, then every Recursively Enumerable language would be Recursive. Therefore, the halting problem is undecidable.*

It means that if I take a TM 'm' and string 'w' and give this (m, w) to an algorithm which says that 'm' halts on 'w' (i.e. halting problem is decidable) then the TM 'm' on 'w' should give Yes or No when the membership algorithm is tested on 'm' and 'w'. This means that there will always be a concrete Yes or No answer from all of the TM in the set of all TMs. Therefore, all the languages which are Recursively Enumerable will become Recursive. But there exists a proof that atleast one Recursively Enumerable language is not recursive. A contradiction has occurred. So, the halting problem is not decidable.

## **SOME UNDECIDABLE PROBLEMS BASED ON TURING MACHINE HALTING PROBLEM**

Let's see some problems which can be converted from Halting Problem of Turing Machine. Based on that we can say that the problem is undecidable or not.

**Problem 1:** The state entry problem is given a Turing machine  $M$ , a state  $q \in Q$  and  $w \in \Sigma^*$ . Decide whether or not the state 'q' is ever entered when  $M$  is applied to 'w'.

*Proof:* Using this statement I can make the halting problem decidable, so this statement or problem is undecidable.

*Explanation:* Let's say we have a Turing machine. A TM halts on a dead configuration. So for every dead configuration state I can transition that state to a new state 'q' which will be the new dead configuration. Therefore, this problem becomes decidable. But if this is the case, then the general halting problem becomes decidable too. Therefore, this problem is *undecidable* by contradiction.

**Problem 2:** Given a TM 'M', whether or not  $M$  halts if started with a blank tape.

*Answer:* This problem is undecidable which can be proved by reducing the halting problem to this problem.

**Note:** Almost any problem related to Recursively Enumerable languages is undecidable. The reason is the unavailability of membership algorithm.

(We will see the undecidable RE based questions in a table later).

## POST CORRESPONDENCE PROBLEM

In context free grammars we have few problems like ambiguity problem which is undecidable. So, directly taking that halting problem and reducing it to see decidability of an ambiguity problem is somewhat complex. So, people have taken the halting problem and converted it to the *post correspondence problem*. This post correspondence problem is converted to ambiguity problem. Using this it is possible to show that the ambiguity problem of context free languages is undecidable. This becomes a kind of an intermediate step.

**Problem Statement:** Given two sequences of 'n' strings on some alphabet  $\Sigma$ , sat  $A = w_1 w_2 \dots w_n$  and  $B = v_1 v_2 \dots v_n$ , we say there exists a PC-solution for pair (A, B) if there is a non-empty sequence of integers  $i, j, \dots, k$ , such that

$$w_i w_j \dots w_k = v_i v_j \dots v_k$$

PC problem is to devise an algorithm that will tell us for ant (A,B), whether or not there exist a PC-solution.

THIS PROBLEM IS UNDECIDABLE and can be extended to ambiguity problem of context free grammars.

## COMPLEXITY CLASSES

We learnt that a problem is decidable if there exists an algorithm/Turing machine for it. Now, when I say "algorithm" we are not actually interested in all the algorithms. There will be some algorithms that will

not be applicable as programs in practice as they take lots of resources like CPU time and memory. We are interested in finding out the algorithm that is feasible.

We have learnt previously that modifications to Turing Machines does not increase the power. Therefore the set of languages accepted by them remains same. But, some modifications decrease the time complexity of the calculation. Example will be a Turing machine with one tape and a Multi-tape Turing machine when accepting a language:  $L = \{a^n b^n : n \geq 1\}$ . A TM with one tape will accept the language in  $O(n^2)$  and a multi-tape TM will accept the language in  $O(n)$ .

We know that all the problems can't have a Turing Machine. Within the set of problems that can be solved, we divide them into two classes depending on if the problem has a deterministic Turing machine or a Non-deterministic Turing machine as it's acceptor.

Deterministic Turing machine problems fall in **P-class** and non-deterministic Turing machine problems fall in **NP-class**. We are interested in P-class problems generally. If we have an NP-class problem, we don't use the algorithm in practice. Instead we use heuristics for that.

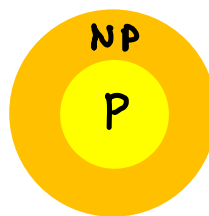
**P-class:** The set of all languages/problems that are accepted by some deterministic Turing machine in polynomial time.

This deterministic TM can use any number of tapes (any number of data structures) in a practical computer.

Polynomial time:  $O(n^k)$ , k can be any number

**NP-class:** The set of all languages/problems that are accepted by some non-deterministic Turing machine in polynomial time.

**Note:** *For every problem that is solved by deterministic Turing machine can be solved by non-deterministic Turing machine. Therefore, we can draw the following:*



Now the question comes, is  $P = NP$  (open question right now). We know that all the Turing Machines are equivalent in power. But when a non-deterministic Turing machine is converted to an equivalent deterministic then the language/problem will maybe accepted by it in *exponential* time instead of polynomial time. Both the machines are not calculating the same problem in same time.

## DECIDABILITY PROBLEMS TABLE

Problem	RL	DCFL	CFL	CSL	Rec	REL
Does $w \in L$ ? (membership problem)	D	D	D	D	D	UD
Is $L = \phi$ ? (Emptiness problem)	D	D	D	UD	UD	UD
Is $L_1 = L_2$ ? (Equality Problem)	D	UD	UD	UD	UD	UD
Is $L = \Sigma^*$ ? (Completeness problem)	D	UD	UD	UD	UD	UD
Is $L_1 \subseteq L_2$ (Subset problem)	D	UD	UD	UD	UD	UD
Is $L_1 \cap L_2 = \phi$	D	UD	UD	UD	UD	UD
Is $L$ finite or not? (finiteness problem)	D	D	D	UD	UD	UD
Is complement of 'L' a language of same type or not?	D	D	UD	D	D	UD
Is intersection of two languages of same type?	D	UD	UD	UD	UD	UD
Is $L$ regular language?	D	D	UD	UD	UD	UD

1. For regular languages everything is decidable
2. For Recursively enumerable languages everything is undecidable.
3. For recursive language membership and complement is decidable
4. Memorize CFL (DCFL and CSL are not asked).

## PROPERTIES OF CONTEXT FREE LANGUAGES

If the language is context free definitely there will be a context free grammar.

**Union:** If  $L_1$  and  $L_2$  are two context free languages, then their union  $L_1 \cup L_2$  is also a context free language.

If  $S_1$  is the start symbol for  $CFG_1$  which generates language  $L_1$  and  $S_2$  is the start symbol for  $CFG_2$  which generates language  $L_2$ , then the start symbol for context free grammar that generates  $L_1 \cup L_2$  will look like:

$$S \rightarrow S_1 / S_2$$

**Concatenation:** If  $L_1$  and  $L_2$  are two context free languages, then their concatenation  $L_1 \cdot L_2$  is also a context free language.

If  $S_1$  is the start symbol for  $CFG_1$  which generates language  $L_1$  and  $S_2$  is the start symbol for  $CFG_2$  which generates language  $L_2$ , then the start symbol for context free grammar that generates  $L_1 \cdot L_2$  will look like:

$$S \rightarrow S_1 \cdot S_2$$

**Closure:** If  $L_1$  is a context free language then  $L_1^*$  is also a context free language.

If  $S_1$  is the start symbol for  $CFG_1$  which generates language  $L_1$  then for language  $L_1^*$  the start symbol for context free grammar will look like:

$$S \rightarrow S_1 S / \varepsilon$$

**Therefore, context free languages are closed under union, concatenation and Kleene closure.**

**Intersection:** If  $L_1$  and  $L_2$  are two context free languages, then their intersection  $L_1 \cap L_2$  is NOT always a context free grammar.

Proof by counter example:

$$L_1 = \{a^n b^n c^m | n, m \geq 0\}$$

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

$$L_1 \cap L_2 = \{a^n b^n c^n | n \geq 0\}$$

Language  $L_1$  and  $L_2$  are context free but the language obtained through their intersection is not context free.

**Complement:** If  $L_1$  is a context free language then the its complement  $\overline{L_1}$  may not always be context free.

Proof by contradiction:

We know that language  $L_1 \cap L_2$  is not context free.

Now we can write:  $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$

Assume: Complement of a context free language is context free.

With this assumption  $\overline{L_1}$  and  $\overline{L_2}$  are context free and we know that union of two context free languages is context free, therefore,  $\overline{L_1} \cup \overline{L_2}$  should be context free always. But we know that  $L_1 \cap L_2$  is not context free always and  $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$

⇒ Our assumption is wrong.

⇒ Hence, context free languages are not closed under complement.

**Context free languages are not closed under intersection and complement.**

## DECIDABLE PROBLEMS ON CONTEXT FREE LANGUAGES

**Membership Problem (Decidable):** In context free languages membership problem is decidable. This means that given a string we can decide if it belongs to the language or not. CYK is the membership algorithm used.

**Emptiness Problem (Decidable):** Given a context free language  $L$ , is it empty or not?

How to decide: Take the CFG for the language and simplify it. In the simplified grammar if you found that the start symbol  $S$  is becoming useless, this means that we cannot generate anything and hence the language will be empty.

**Finiteness Problem (Decidable):** Given a context free language  $L$ , is it finite or not?

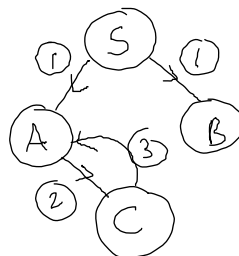
How to decide: Take the CFG for the language and simplify it and in the resulting grammar you try to draw the dependency graph.

Example: Let's say we got the following simplified language:

- ①  $S \rightarrow AB$
- ②  $A \rightarrow aC / a$
- ③  $C \rightarrow aA / b$
- ④  $B \rightarrow a$

The dependency graph will have the variables as nodes:

Dependency graph:



Now, if there is a cycle in dependency graph, then we can say that the language is infinite otherwise it is finite. This is how we are going to *decide* the Finiteness Problem in context free language.

## PROPERTIES OF REGULAR LANGUAGES

1. **Union:** Regular languages are closed under union.

Proof: Let's say two languages  $L_1$  and  $L_2$  are regular. Then their union  $L_1 \cup L_2$  is also regular as the regular expressions, say  $R_1$  and  $R_2$  for the languages  $L_1$  and  $L_2$  can take representation by  $R_1 + R_2$  for language  $L_1 \cup L_2$ .

2. **Concatenation:** Regular languages are closed under concatenation.

Proof: Let's say two languages  $L_1$  and  $L_2$  are regular. Then their concatenation  $L_1 \cdot L_2$  is also regular as the regular expressions, say  $R_1$  and  $R_2$  for the languages  $L_1$  and  $L_2$  can take representation  $R_1.R_2$  for the language  $L_1 \cdot L_2$ .

3. **Kleene Closure:** Regular languages are closed under Kleene Closure  $*$ .

Proof: Let's say a language  $L_1$  is regular. Then, its Kleene Closure  $L_1^*$  is also regular as the regular expression, say  $R_1$  for the language  $L_1$  can take representation  $R_1^*$  for the language  $L_1^*$  and  $R_1^*$  will also be a regular expression.

4. **Complement:** Regular languages are closed under complementation.

Proof: Let  $L_1$  be a regular language. Then  $\overline{L_1} = \Sigma^* - L_1$ . For  $L_1$  we will have a DFA and for  $\overline{L_1}$  we will have a complement of the DFA which is also a DFA. And since DFAs are acceptors of the Regular Languages, therefore the complement  $\overline{L_1}$  is also regular.

5. **Intersection:** Regular languages are closed under intersection.

Proof: We can write  $L_1 \cap L_2$  as  $\overline{\overline{L_1} \cup \overline{L_2}}$  and using the complement and union properties discussed above we can prove that  $L_1 \cap L_2$  is also regular. (*This is called proof by construction*).

6. **Difference:** Regular languages are closed under difference

Proof: I can again use proof by construction to say that  $L_1 - L_2 = L_1 \cap \overline{L_2}$  and use previous properties to conclude that  $L_1 - L_2$  is also a regular language.

7. **Reversal:** Regular languages are closed under reversal.

Proof: Take the DFA of the language  $L$ , reverse it. The reversed FA is going to take  $L^R$  as its language. Since we got a FA, the language it accepts is also regular.

8. **Homomorphism:** Regular languages are closed under homomorphism.

- A homomorphism is a way to compare two groups for structural similarities. It is a function between two groups which preserves the group structure in each group. It's a tool to compare two groups for similarities.

9. **Inverse Homomorphism:** Regular languages are closed under inverse homomorphism

The inverse of homomorphic image of a language  $L$  is:

$$h^{-1}(L) = \left\{ \frac{x}{h(x)} \right\} \text{ is in } L$$

For a string  $w$ ,  $h + (w) = \left\{ \frac{x}{h(x)} \in L \right\}$

10. **Right Quotient:** Regular languages are closed under right quotient.

Let  $L_1$  and  $L_2$  be languages on the same alphabet, then, the right quotient of  $L_1$  on  $L_2$  is defined as:

$$\frac{L_1}{L_2} = \{x : xy \in L_1 \text{ for some } y \in L_2\}$$

Example:  $L_1 = \{01,001,101,0001,1101\}$ ,  $L_2 = \{01\}$  then  $\frac{L_1}{L_2} = \{\epsilon, 0,1,00,11\}$

You don't directly perform the right quotient operation on the regular expressions. Make the language first and then divide every string in  $L_1$  with string in  $L_2$ .

11. **INIT operation:** Regular languages are closed under INIT operation.

Example:  $L = \{a, ab, aabb\}$

Init means what are all the prefixes of a string.

$\text{Init}(L) = \{\epsilon, a, \epsilon, a, ab, \epsilon, a, aa, aab, aabb\} = \{\epsilon, a, ab, aa, aab, aabb\}$

To say the regular languages are closed under INIT, you create a DFA and set every state as the final state except the dead state.

12. **Substitution:** Regular languages are closed under substitution.

It means that if for a language  $L$ , every symbol is replaced with *other language* (not a string of other language because that is homomorphism). Then the resulting language is also a Regular Language.

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

$f(a) = 0^*$

$f(b) = 01^*$

$L = a + b^*$

$f(L) = 0^* + (01^*)^*$

13. **Regular languages are NOT CLOSED under infinite union of Regular Languages. This means there exists at least one example that shows that Regular Languages are not closed under infinite union.**

Example:  $L_1 = \{a^1b^1\}$ , therefore  $L_1 \cup L_2 \cup L_3 \dots \Rightarrow L = \{a^n b^n : n \geq 1\}$

$L_2 = \{a^2b^2\}$

$L_3 = \{a^3b^3\}$

...

Language  $L$  is not regular.