Symbols: Building blocks of language
eg { a, b, c, 0, 1, 2, 3 }

Alphabet: Denoted by Σ. Alphabets are finite set of symbols.

String: Sequence of alphabets

Language: Collection of strings. Language can be finite or infinite

$\Sigma^0 = \varepsilon$ = strings of length 0 (finite language)

Automata is a machine/model to check if a string is part of language or not.

Power of Σ: Minimum power of sigma can be 0

Σ^n = set of all strings of length n

Σ^0 = set of all strings of length 0, also denoted by ε (null string)

Σ^* is called Kleene Closure: Set of all strings of all length possible over given alphabets = Infinite Language

Σ^+ = Positive closure = All sequence of strings except null string = $\Sigma^* - \Sigma^0$

Grammar: Standard way of representing language

A grammar G is defined as quadruple.

$G = \{ V, T, P, S \}$

V → Variable (denoted by capital letters)

T → Terminal (denoted by small letters)

P → Production Rule

S → Start Symbol

eg. $S \to aSb / \varepsilon$

We are given 2 production rules aSb and ε. We can start from either aSb or ε and substitute value of variable S with aSb or ε

Variable → S

Terminal → a, b

Production rule → aSb, ε

Start → aSb or ε

Ans) ε, aSb, aaSbb, aaaSbbb
ab, aabb, aaabbb
and so on

⇒ ε, ab, a^2b^2, a^3b^3, ..., a^nb^n $(n \geq 0)$

$L = a^n b^n$, $n \geq 0$

Ques) Can we generate abab from above grammar

Ans) No, because abab is not of form $a^n b^n$, $n \geq 0$

Ques)
$S \to SS$
$S \to aSb$
$S \to bSa$
$S \to \varepsilon$

Ans) ε, SS, aSbaSb, abab
aSbbSa, abba
abSaba aSbb, abaaabb
⇒ ε, abab, abba, ababaabb, ...
⇒ L: $n_a = n_b$ [no of a = no of b]

Chomsky classification:

a) Type 0 (Unrestricted Grammar):
It is of the form $X \to Y$ (X derives Y)

where $X \in$ Atleast one variable and
$Y \in (V+T)^*$ [any combination of variable and terminal]

eg. $S \to a$, $aSb \to abSb$ are Type 0 grammar but $a \to Sb$ is not [no variable on left side]

Language derived from Type 0 grammar is called Recursive Enumerable Language. Automata from Type 0 grammar is called Turing Machine.

Representation: $G(\{S\}, \{a, b\}, \{S \to a, aS \to abS\}, \{S\})$
Variable, Terminal, Production Rules, Start

b) Type 1 (Context Sensitive Grammar)
Corresponding Language: Context Sensitive Language
Corresponding Automata: Linear Bounded Automata

Type 1 = Type 0 + Length of left side <= Length of right side

$\therefore$ Type 0 + $|X| \leq |Y|$

eg. $aSb \to b$ is not Type 1 because length of left is 3 and length of right is 1
$Sab \to abab$ is Type 1 because $3 \leq 4$

Exceptions:

(a) $S \to \epsilon$ is Type 1 grammar only if start variable derives null (ϵ)
$S \to AB$, $A \to a$, $B \to \epsilon$ is not Type 1 grammar because B is deriving null

(b) If start variable is deriving null then we can't use start variable on right side
eg. $S \to \epsilon$, $S \to aS$ is not Type 1 grammar

(c) Type 2 (Content Free Grammar)
Corresponding Language: Content Free Language
Corresponding machine: Push Down Automata

$X \to Y$

Type 2 = Type 1 + $X \in$ Single Variable
i.e only one variable on left side

$S \to aAb$ ✓

$aS \to ab$ ✗

Exception:

(a) $S \to AB$, $A \to G$, $B \to \epsilon$ is allowed in CFG i.e any variable can derive null (ϵ).

(d) Type 3 (Regular Grammar)
Corresponding language: Regular Language
Corresponding machine: Finite Automata

$X \to Y$

Type 3 = Type 2 + only one variable on right side either on extreme left or in extreme right

eg. (i) $S \to aaS$ ✓
(ii) $S \to Saa$ ✓
(iii) $S \to aSa$ ✗
(iv) $S \to Saab$ [on left]] combination of left and right not allowed
$S \to aaaS$ [on right]]
Left and right simultaneously not allowed
(v) $S \to Saabb$] ✓ Allowed
$S \to aaa$]

Finite Automata: Finite automata is an abstract computing device. It is a mathematical model of system with discrete inputs, outputs, states and set of transitions from state to state that occurs on input symbols from alphabet Σ.

Representation:-

1) Graphical (Transition diagram or transition table)
2) Tabular (Transition table)
3) Mathematical (Transition function or mapping function)

Formal definition of finite automata

A finite automata is a 5-tuple

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

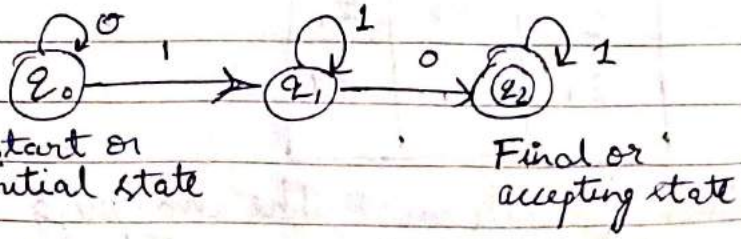
where

Q: is a finite set called states
Σ: is a finite set called alphabets
$\delta: Q \times \Sigma \to Q$ is the transition function
$q_0 \in Q$ is the start state also called initial state
$F \subseteq Q$ is the final state or accepting state.

1) Transition Diagram (Transition Graph):

It is a finite, directed and labelled graph consisting nodes and edges
Each node represents state of machine
Each edge represents, transition from one state to another

eg.

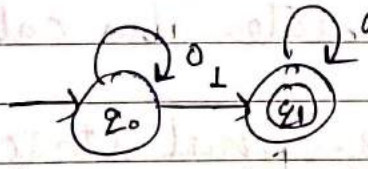


{0, 1} are inputs
q_0 - initial state
q_1 - intermediate state
q_2 - final state

2) Transition Table : It is basically a tabular representation of the transition function that takes two arguments (a state and a symbol) and returns a value (the next state.
→ Rows corresponds to state
→ Columns correspond to input symbols
→ Entries correspond to next state
→ The start state is marked with an arrow (→)
→ The accept state is marked with star (*)

$$\delta = Q \times \Sigma \to Q$$

eg. Given transition diagram.



draw transition table

Ans) q_0 → initial state
q_1 → final state

$q_0 \times 0 \to q_0$ $\quad$ $q_1 \times 0 \to q_1$
$q_0 \times 1 \to q_1$ $\quad$ $q_1 \times 1 \to q_1$

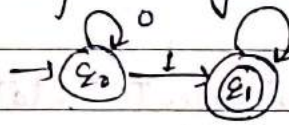
	0	1
q_0	q_0	q_1
q_1	q_1	q_1

3) Transition function : The mapping function or transition function is denoted by δ

Two parameters are passed to this transition function
(i) Current state (ii) Input Symbol
The transition function returns a state (next state)
δ(Current state, current-input symbol) = next state

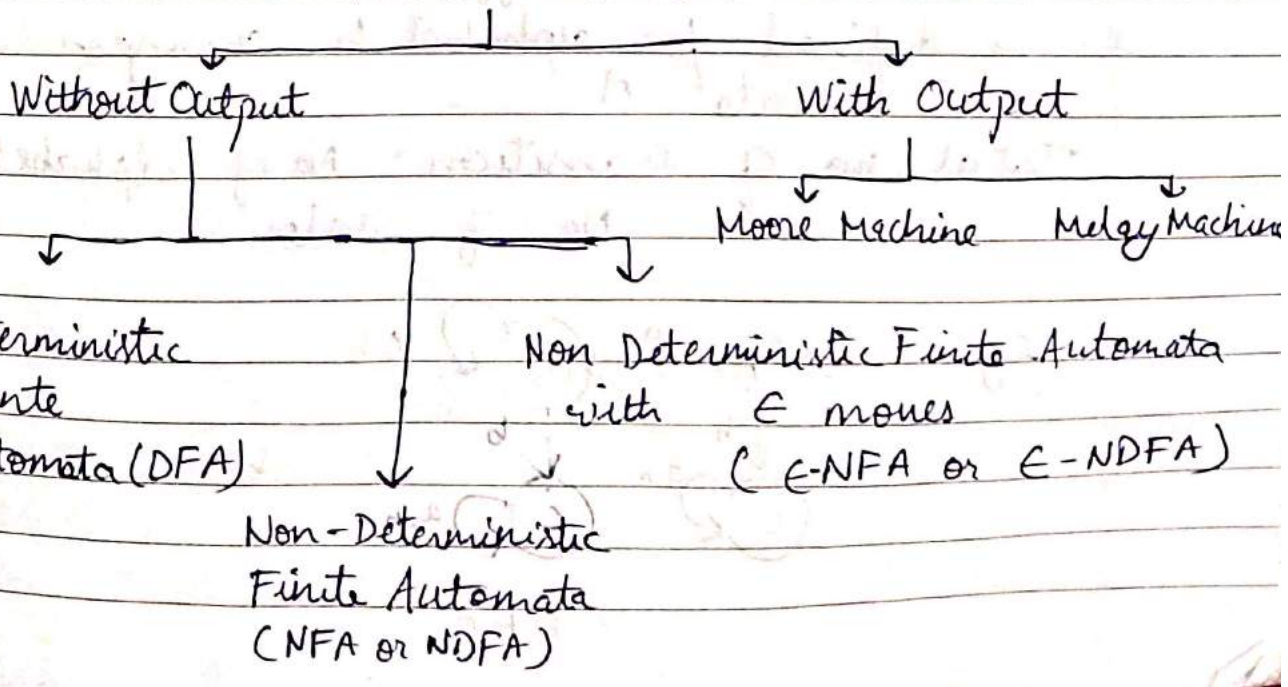
eg. $\delta(q_0, a) = q_0$ $\delta(q_1, a0) = q_1$
$\delta(q_0, 1) = q_1$ $\delta(q_1, 1) = q_1$

Corresponding transition diagram



Applications of finite automata
1) Used in compiler design
2) In switching theory
3) In design and analysis of complex s/w and h/w
4) To prove correctness of the program
5) To design finite state machines like Melay and Moore machine
6) It is base for formal languages and these formal languages are useful for programming language

Types of Finite Automata



Deterministic Finite Automata (DFA)

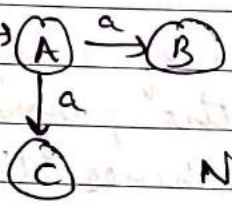
$M(Q, \Sigma, q_0, F, \delta)$

Q : Non empty finite set of states
Σ : Input Alphabet
q_0 : Initial state ($q_0 \in Q$)
F : Set of final states ($F \subseteq Q$)
δ : Transition function

$$Q \times \Sigma \to Q$$

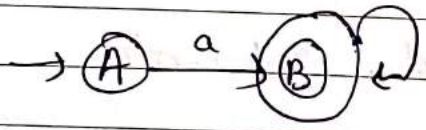
Deterministic means different states can not be obtained by applying same input symbol to given state

eg.



Non Deterministic

DFA must be complete means transition should be defined for each input alphabet

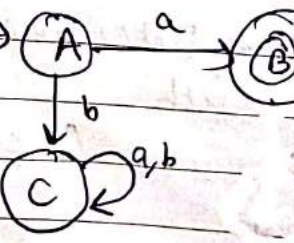


×

Dead Configuration because transition is defined for alphabet b corresponding to state A

Total no of transition = No of alphabet × No of states

eg.



✓

DFA

Non Deterministic Finite Automata (NFA)

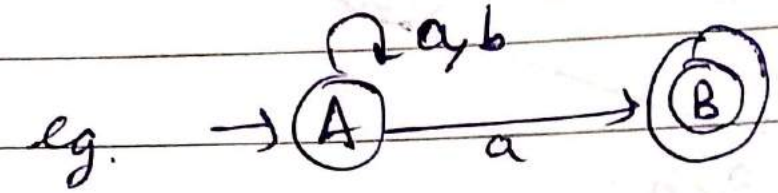
$M(Q, \Sigma, q_0, F, \delta)$

Q: Non empty finite set of states

Σ: Input alphabet

q_0 : Initial state
F : Set of final states
δ : Transition function

eg.



* $Q \times \Sigma \rightarrow 2^Q$

ϵ - NFA

$M(Q, \Sigma, q_0, F, \delta)$

Q: Non empty finite set of states

Σ: Input alphabet

q_0: Initial state

F: Set of final states

δ: Transition function

$$Q \times \Sigma \cup \{\epsilon\} \rightarrow 2^Q$$

Regular Expression

Let Σ be a given alphabet

1. ϕ, ϵ and $a \in \Sigma$ are regular expression (primitive regular expression)
2. If r_1 and r_2 are regular expression, then
   a) $r_1 + r_2$ (Union)
   b) $r_1 . r_2$ (Concatenation)
   c) r_1^* (Kleen Star)
   d) (r_1) are also regular expressions

classmate
Date
Page

Limitations of Finite state Automata

- Limited memory
- Strings without comparison
- Linear Power

Applications of Finite state Automata

- Word processor program
- Digital logic design
- Lexical analyzer
- Switching circuit design
- Text editor
- Software for scanning large bodies of text
- Software with finite state like vending machine
- Game design

Date
Page

Pushdown Automata

$M(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

Q: Non empty finite set of states
Σ: Input alphabet
Γ: Stack alphabet
δ: Transition function
q_0: Initial state
Z_0: Stack start symbol (Bottom of the stack)
F: Set of final states

Transition Function

$Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow Q \times \Gamma^*$ (DDPA)

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

Pumping Lemma (For Regular Languages)

>> Pumping Lemma is used to prove that a Language is NOT REGULAR

>> It cannot be used to prove that a Language is Regular

If A is a Regular Language, then A has a Pumping Length 'P' such that any string 'S' where $|S| \geq P$ may be divided into 3 parts $S = x\,y\,z$ such that the following conditions must be true:

(1) $x\,y^i z \in A$ for every $i \geq 0$

(2) $|y| > 0$

(3) $|xy| \leq P$

Pumping Lemma (For Context Free Languages)

Pumping Lemma (for CFL) is used to prove that a language is NOT Context Free

If A is a Context Free Language, then, A has a Pumping Length 'P' such that any string 'S', where $|S| \geq P$ may be divided into 5 pieces $S = uvxyz$ such that the following conditions must be true:

(1) $u\,v^i x\,y^i z$ is in A for every $i \geq 0$

(2) $|v\,y| > 0$

(3) $|v\,x\,y| \leq P$

Closure properties of regular language

1) Union
2) Concatination
3) Kleen Star
4) Complement
5) Intersection
6) Difference
7) Reversal

Proof:

1) Union: Let $L_1, L_2 \in$ {set of regular language}

$L_1 \cup L_2 = L_3$

$r_1 + r_2 = r_3$

r_1 = regular expression corresponding to L_1

r_2 = " " " " L_2

Since there exist regular expression for L_3 i.e. r_3. Hence $L_3 \in$ {set of regular language}

(2) Concatination:

$L_1 . L_2 = L_3$

$r_1 . r_2 = r_3$

Hence $L_3 \in$ {set of regular language}

(3) Kleen* star:

$(L_1)^* = L_2$

$(r_1)^* = r_2$

Hence $L_2 \in$ {set of regular language}

(4) Complement

$\overline{L_1} = L_2$

Design DFA for L_1.

Do complement of DFA by converting Final State to Non final and vice versa

$L_1 \Rightarrow$ DFA $\Rightarrow \overline{\text{DFA}}$

Corresponding language for $\overline{\text{DFA}}$ is L_2

Since we can design DFA for L_2. Hence L_2 is also regular language

(5) $L_1 \cap L_2 = L_3$

$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$

Since union and complement of regular language is also regular.

Hence $L_1 \cap L_2$ is also regular.

(6) $L_1 - L_2 = L_3$

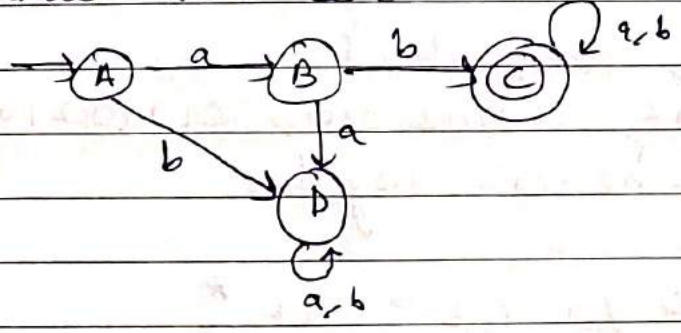
$L_1 - L_2 = L_1 \cap \overline{L_2}$

Since complement and intersection of regular language is also regular. Hence L_3 is regular.

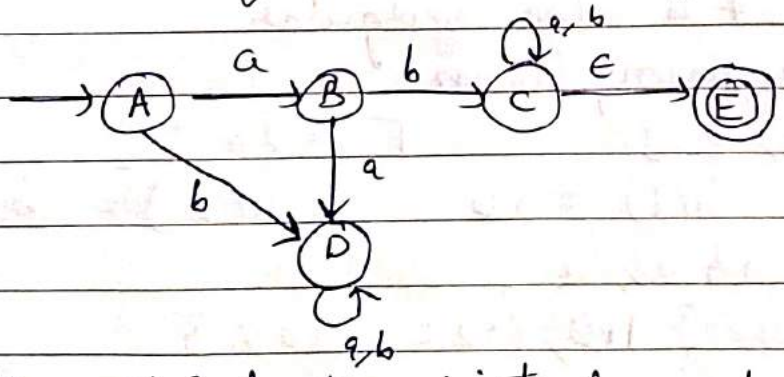
(7) Reversal:

Let L = starting with ab over $\Sigma\{a, b\}$.

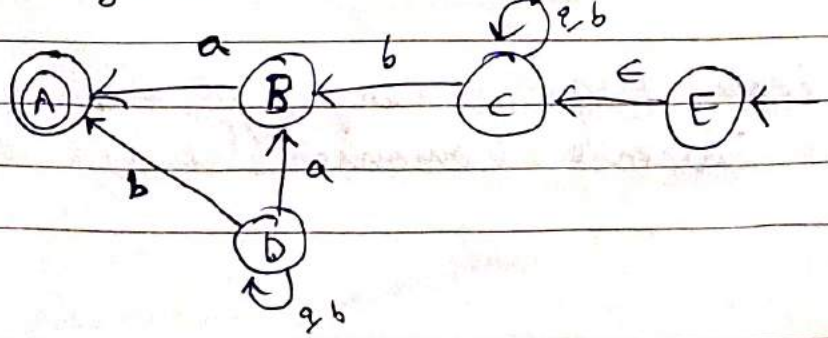
Finite Automata:



Convert final to non final with ϵ moves



Now final to initial and vice versa and change direction of arrows



Language corresponding to this finite automata is L^R

Hence regular language are closure under reversal.

(8) Homomorphism:

Suppose Σ and Γ (tou) are alphabets. Then a function $h: \Sigma \to \Gamma^*$ is called homomorphism

1. $h(\varepsilon) = \varepsilon$
2. $h(xyz) = h(x) . h(y) . h(z)$

eg. $\Sigma = \{a, b\}$ $\Gamma = \{0, 1, 2\}$

$h(a) = 010$

$h(b) = 102$

$\Rightarrow h(abba) = h(a).h(b)h(b)h(a)$

$= 010\ 102\ 102\ 010$

Let $L_1 = \{ab, aba, abb\}$

$\Rightarrow L_2 = \{010102, 010102010, 010102102\}$ is also finite, hence regular.

Let RE for $L_3 = ab^*$

then RE for $L_4 = 010(102)^*$

hence L_4 is also regular.

(9) Inverse Homomorphism:

$\Sigma = \{0, 1, 2\}$ $\Gamma = \{a, b\}$

$h(0) = a$ $h(1) = ab$ $h(2) = ba$

(1) $L_1 = \{ababa\}$

$h^{-1}(L_1) = \{110, 022, 102\}$

$$h(h^{-1}(L)) \subseteq L$$

Hence regular language are closure under inverse homomorphism.

(9) Substitution: Every symbol of a language is replaced by other language.

$\Sigma = \{0, 1\}$ $\quad$ $\Gamma = \{a, b\}$

$f(0) = a^*$ $\quad$ $0 \Rightarrow a^n \mid n \geq 0$

$f(1) = bb^*$ $\quad$ $1 \rightarrow bb^n \mid n \geq 0$

$b^n \mid n = 1$

(1) $L = 0 + 1^*$

$f(L) = a^* + (bb^*)^*$

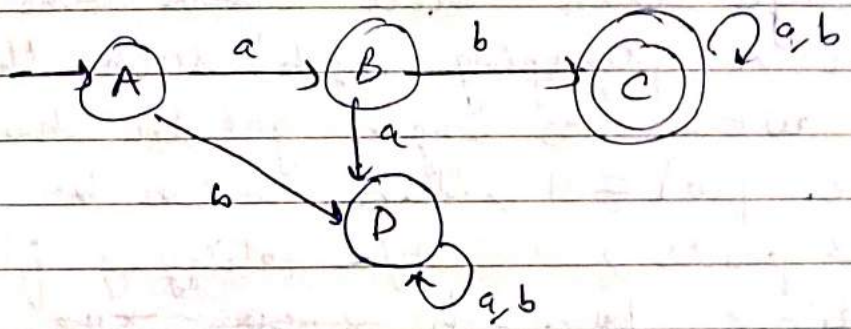
Hence we get regular expression after substitution. So regular language are closure under substitution.

(10) INIT

eg. GATE

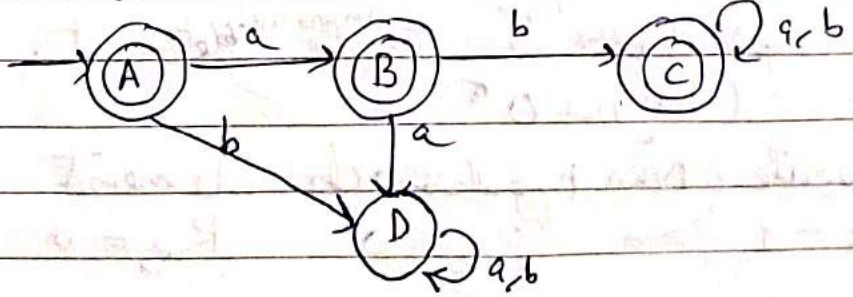
INIT will be { ∈, G, GA, GAT, GATE }

(1) L = set of all strings starting with ab



DFA for L

By converting all the non final states except dead state in DFA of L we get DFA of INIT(L).



DFA for INIT(L)

Hence regular language are closure under INIT.

(11) Quotient :

$L_1/L_2 = \{ y \mid xy \in L_1 \text{ for some } x \in L_2 \}$ Left

$L_1/L_2 = \{ x \mid xy \in L_1 \text{ for some } y \in L_2 \}$ Right

eg.

$L_1 = \{01, 001, 101, 0111, 1101\}$

$L_2 = \{01\}$

$L_1/L_2 = \{\epsilon, 11\}$ Left

$L_1/L_2 = \{\epsilon, 0, 1, 11\}$ Right

Regular language are closure under quotient.

(12) Infinite Union and Subset :

Regular language are not closure under infinite union and subset.

Closure Property of CFL

1) Union

$L_1 \cup L_2 = CFL$

Let $L_1 = a^n b^n \mid n \geq 1$

$\Rightarrow S_1 \rightarrow a\ S_1\ b \mid \epsilon$

and $L_2 = c^n d^n \mid n \geq 1$

$\rightarrow S_2 \rightarrow c\ S_2\ d \mid \epsilon$

$S \rightarrow S_1 \mid S_2$

$S_1 \rightarrow a\ S_1\ b \mid \epsilon$

$S_2 \rightarrow c\ S_2\ d \mid \epsilon$

Hence union is closure under CFL

2) Concatenation

$S \rightarrow S_1 S_2$

$S_1 \rightarrow a\ S_1\ b \mid \epsilon$

$S_2 \rightarrow c\ S_2\ d \mid \epsilon$

$L_1 = a^n b^n \mid n \geq 0$

$L_2 = c^n d^n \mid n \geq 0$

$L = a^n b^n c^m d^m \mid n, m \geq 0$

Hence intersection is closure under CFL

3) Kleene Closure

Let $L_1 = a^n b^n \mid n \geq 0$

$S_1 \rightarrow a\ S_1\ b \mid \epsilon$

$S \rightarrow S_1 S \mid \epsilon$

Hence Kleene closure is closure under CFL

4) Intersection

Intersection is not closure under CFL

$L_1 = a^n b^n c^m \mid n, m \geq 0$

$L_2 = a^m b^n c^n \mid m, n \geq 0$

$L_1 \cap L_2 = a^n b^n c^n \mid n \geq 0$

$L_1 = \{ \epsilon, ab, abc, aabb, aabbc, aabbcc, aaabbb, a^3b^3c, a^3b^3c^2, a^3b^3c^3, \ldots \}$

$L_2 = \{ \epsilon, bc, abc, bbcc, abbcc, a^2b^2c^2, b^3c^3, ab^3c^3, a^2b^3c^3, a^3b^3c^3, \ldots \}$

$L = L_1 \cap L_2 = \{ \epsilon, abc, a^2b^2c^2, a^3b^3c^3, \ldots \}$

$\Rightarrow L = a^n b^n c^n \mid n \geq 0$

L_1 and L_2 are CFL, but

L is not CFL

Hence, intersection is not closure under CFL

5) Complement

Let L_1 and L_2 be two CFL

$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$

Let's assume that complement is closure under CFL, then

$\Rightarrow \overline{L_1}$ and $\overline{L_2}$ are also ... CFL

We know that union is closure under CFL

so $\overline{L_1} \cup \overline{L_2}$ is also CFL

$\Rightarrow \overline{\overline{L_1} \cup \overline{L_2}}$ is also CFL

$\Rightarrow L_1 \cap L_2$ is CFL

Which is FALSE

Our assumption is WRONG

Hence, complement is not closure under CFL.

Turing Machine

Standard Definition

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

Q : Non empty finite set of states

Σ : Input Alphabet

δ : Transition Function

Γ : Tape Alphabet

q_0 : Initial State, $q_0 \in Q$

B : Special Symbol called blank

F : Set of Final States $F \subseteq Q$

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

eg. $(q_0, a) \rightarrow (q_1, X, R)$

- Deterministic Turing Machine : When a tape alphabet is applied on a state we get only one transition
- Non Deterministic Turing Machine : When a tape alphabet is applied on a state we can get one or more transition corresponding to that tape alphabet.

Turing Machine $\rightarrow$ Acceptor TM

$\rightarrow$ Transducer TM

Transducer : No requirement of final state

Acceptor : Final state is needed

Variations of Turing Machine

1) Turing Machine with stay option

→ $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$

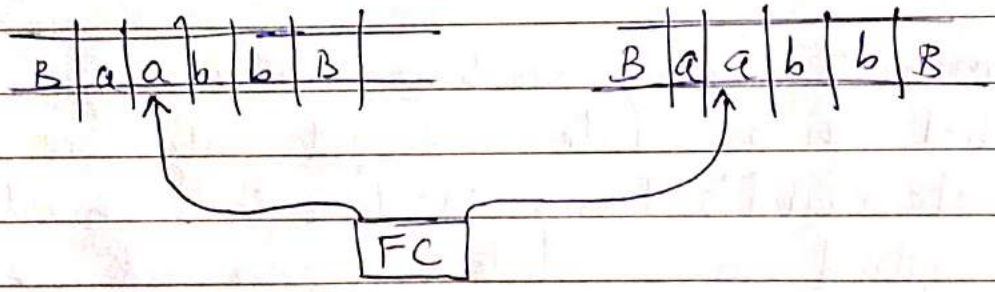
→ Accepts same number of languages as standard turing machine.

→ Power of TM with stay = Power of standard TM

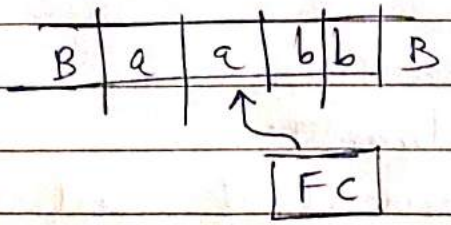
2) Turing Machine with semi infinite tape
→ It has finite number of blanks on one side of tape
→ eg. | a | a | B | B | B . . . .
→ Same power as that of standard TM

3) Offline Turing Machine
→ We modify the input in separate file
→ Same power as that of standard TM.

4) Multitape TM
→ It can have multiple tapes corresponding to a finite control

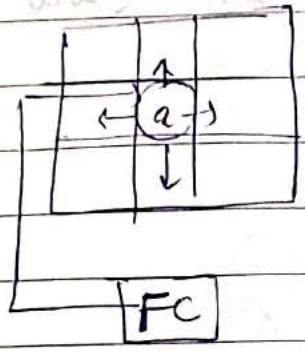


Multitape TM



→ Same power as that of standard TM
→ More efficient than standard TM (takes less time to compute)
→ Complexity of Multitape TM = $O(n)$
Complexity of standard TM = $O(n^2)$

5) Multidimensional TM
→ $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, U, D\}$



→ Same power as that of standard TM.

6) Jumping TM

→ $\varphi \times \Gamma \rightarrow \varphi \times \Gamma \times \{L, R\} \times \{n\}$

where n is number of cells jumped.

→ Accepts same number of language as standard TM so same power

7) Non Deterministic TM

→ $\varphi \times \Gamma \rightarrow 2^{\varphi \times \Gamma \times \{L, R\}}$

→ Accepts same number of language as standard TM, so same power

→ Power of deterministic TM = Power of non deterministic TM

Recursive Enumerable language: If we can design Turing Machine for a language then that language is called Recursive Enumerable Language.

Halting Turing Machine : Turing Machine that doesn't go into infinite loop.

Recursive Language: Language corresponding to Halting Turing Machine is called recursive language.

Reaursive Language $\subset$ Recursive Enumerable Language

Church Turing Thesis

1. Anything that can be done on 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 is more powerful than the Turing Machine Model.

Multitape Turing Machine

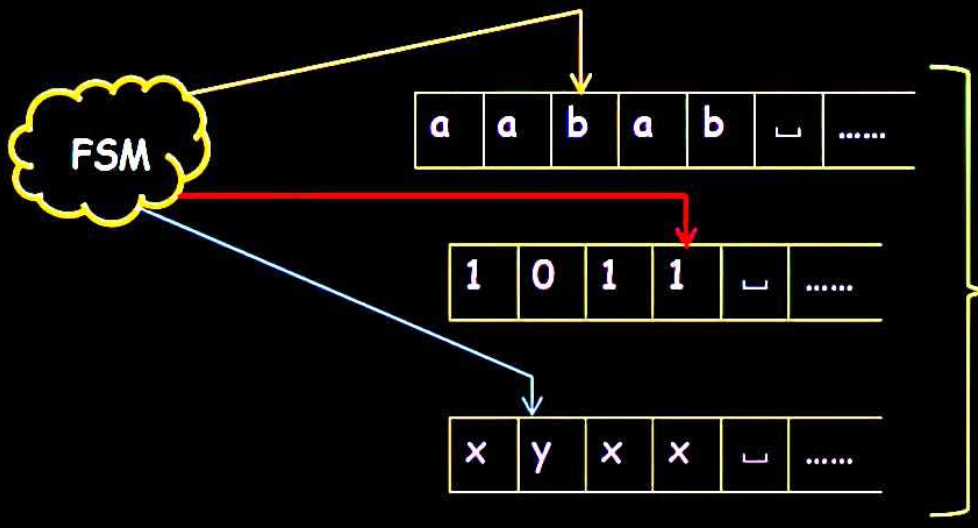
Theorem: Every Multitape Turing Machine has an equivalent Single Tape Turing Machine

Proof

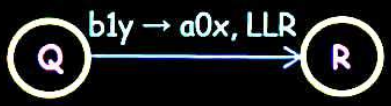
Given a Multitape Turing Machine show how to build a single tape Turing Machine

- Need to store all tapes on a single tape
  Show data representation
- Each tape has a tape head
  Show how to store that info
- Need to transform a move in the Multitape TM into one or moves in the Single Tape TM

Multitape Turing Machine

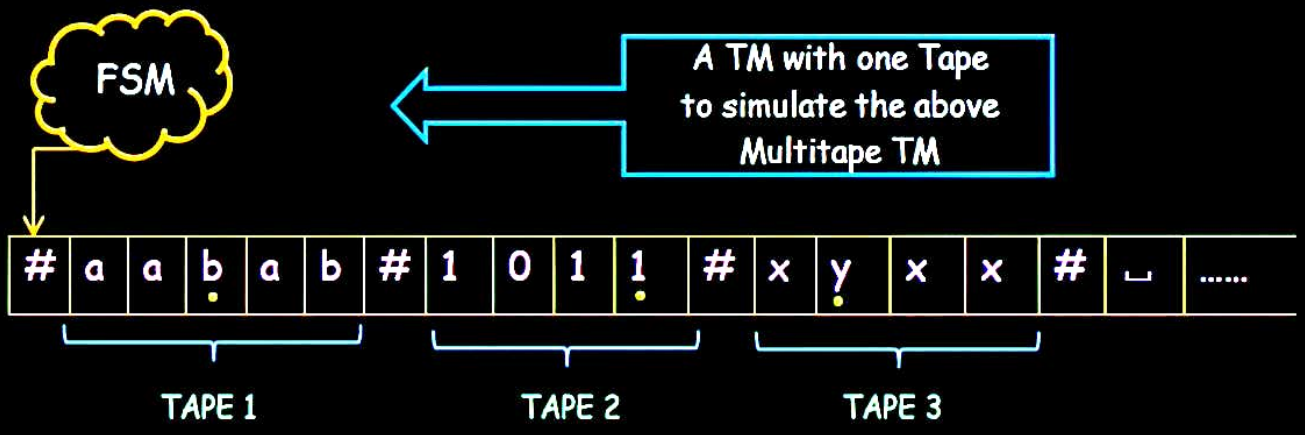


Several Tapes .
Each has its own
TAPE HEAD



An Example Machine with
K=3 Tapes

Single Tape Turing Machine



- Add "dots" to show where Head "K" is
- To simulate a transition from state Q, we must scan our Tape to see which symbols are under the K Tape Heads
- Once we determine this and are ready to MAKE the transition, we must scan across the tape again to update the cells and move the dots
- Whenever one head moves off the right end, we must shift our tape so we can insert a ␣

Nondeterminism in Turing Machine (Part-2)

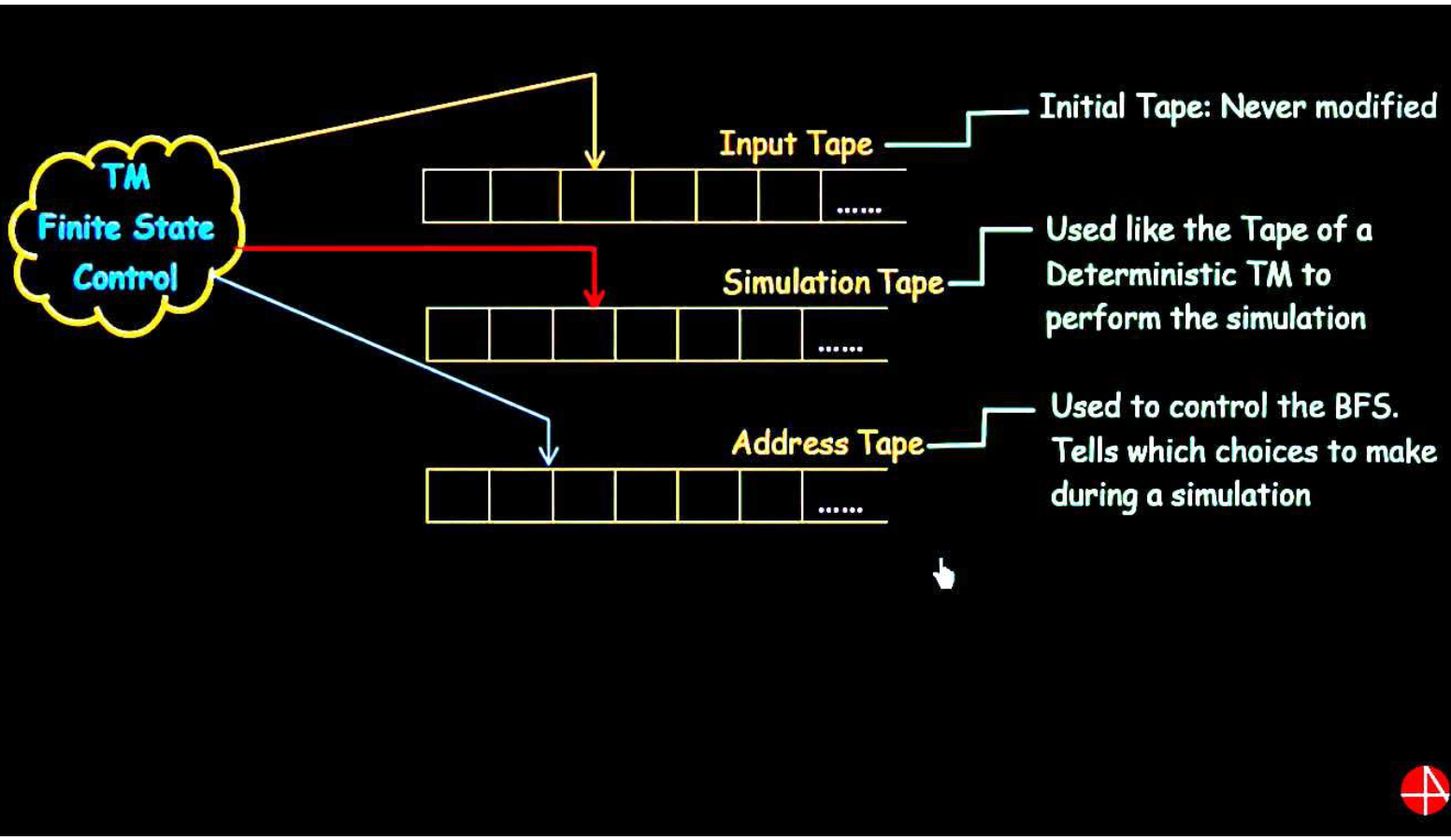
Theorem: Every NondeterministicTM has an equivalent Deterministic TM

Proof:

- Given a Nondeterministic TM (N) show how to construct an equivalent Deterministic TM (D)
- If N accepts on any branch, the D will Accept
- If N halts on every branch without any ACCEPT, then D will Halt and Reject.

Approach:

- Simulate N
- Simulate all branches of computation
- Search for any way N can Accept


TM
Finite State
Control
Input Tape
.....
Initial Tape: Never modified
Simulation Tape
.....
Used like the Tape of a
Deterministic TM to
perform the simulation
Address Tape
.....
Used to control the BFS.
Tells which choices to make
during a simulation

Algorithm:

Initially: TAPE 1 contains the Input
TAPE 2 and TAPE 3 are empty

- Copy TAPE 1 to TAPE 2
- Run the Simulation
- Use TAPE 2 as "The Tape"
- When choices occur (i.e. when Nondeterministic branch points are encountered) consult TAPE 3
- TAPE 3 contains a Path. Each number tells which choice to make
- Run the Simulation all the way down the branch as far as the address/ path goes (or the computation dies)
- Try the next branch
- Increment the address on TAPE 3
- REPEAT

If ACCEPT is ever encountered,
Halt and Accept
If all branches Reject or die out,
then Halt and Reject

The Universal Turing Machine

The Language

$A_{TM} = \{ \langle M,w \rangle \mid M \text{ is a Turing Machine and } M \text{ accepts } w \}$

is Turing Recognizable

Given the description of a TM and some input, can we determine whether the machine accepts it?

- Just Simulate/ Run the TM on the input

M Accepts w: Our Algorithm will Halt & Accept
M Rejects w: Our Algorithm will Halt & Reject.
M Loops on w: Our Algorithm will not Halt.

The Universal Turing Machine

Input: M = the description of some TM
w = an input string for M

Action: - Simulate M
- Behave just like M would (may accept, reject or loop)

The UTM is a recognizer (but not a decider) for

A_{TM} = { <M,w> | M is a TM and M accepts w}

Decidability and Undecidability

Recursive Language:

- A language 'L' is said to be recursive if there exists a Turing machine which will accept all the strings in 'L' and reject all the strings not in 'L'.
- The Turing machine will halt every time and give an answer (accepted or rejected) for each and every string input.

Recursively Enumerable Language:

- A language 'L' is said to be a recursively enumerable language if there exists a Turing machine which will accept (and therefore halt) for all the input strings which are in 'L'.
- But may or may not halt for all input strings which are not in 'L'.

Decidable Language:

A language 'L' is decidable if it is a recursive language. All decidable languages are recursive languages and vice-versa.

Partially Decidable Language:

A language 'L' is partially decidable if 'L' is a recursively enumerable language.

Undecidable Language:

- A language is undecidable if it is not decidable.
- An undecidable language may sometimes be partially decidable but not decidable.
- If a language is not even partially decidable, then there exists no Turing machine for that language

Recursive Language	TM will always Halt
Recursively Enumerable Language	TM will halt sometimes & may not halt sometimes
Decidable Language	Recursive Language
Partially Decidable Language	Recursively Enumerable Language
UNDECIDABLE	No TM for that language

The Halting Problem

Given a Program, WILL IT HALT ?

Given a Turing Machine, will it halt when run on some particular given input string?

Given some program written in some language (Java/C/ etc.) will it ever get into an infinite loop or will it always terminate?

Answer:

- In General we can't always know.
- The best we can do is run the program and see whether it halts.
- For many programs we can see that it will always halt or sometimes loop

BUT FOR PROGRAMS IN GENERAL THE QUESTION IS UNDECIDABLE.

Undecidability of the Halting Problem

Given a Program, WILL IT HALT ?

Can we design a machine which if given a program can find out or decide if that program will always halt or not halt on a particular input?

Let us assume that we can:

H (P, I)

- Halt
- Not Halt

This allows us to write another Program:

```
C ( X )
  if { H (X, X) == Halt }
       Loop Forever;
  else
       Return;
```

Let us assume that we can:

H (P, I)

- Halt
- Not Halt

This allows us to write another Program:

```
C ( X )
  if { H (X, X) == Halt }
       Loop Forever;
  else
       Return;
```

If we run 'C' on itself:

C (C)

- H (C,C) == Halt → Not Halt
- H (C,C) == Not Halt → Halt

Recursion Theorem

Let T be some turing machine that computes some function t. Then there will always exist another turing m/c R that does the same thing as t when t is applied to a description of itself.

Reducibility (A technique of proving undecidability)

We reduce Hard problem to easier problem. The solution of easier problem then can be used to solve the harder problem

Theorem: P is undecidable

Proof: 1) Assume P is decidable

2) Reduce acceptance problem for turing machine, A_{TM} (a HARD PROBLEM) into P (an EASY PROBLEM).

3) Use the solution of P to solve A_{TM}.
   - Use decidability of P to decide A_{TM}.
   - Build a TM to decide A_{TM} using the TM to decide P as subroutine.

BUT WE KNOW THAT DECIDER FOR A_{TM} CANNOT EXIST

Hence, P is Undecidable