Introduction to Automata Theory

Purpose and motivation

- What are the mathematical properties of computer hardware and software?
- What is a **computation** and what is an **algorithm**? Can we give rigorous mathematical definitions of these **notions**?
- What are the limitations of computers? Can "everything" be computed?

Purpose: Develop formal mathematical models of computation that reflect real-world computers.

Alan Turing (1912-1954)

- Father of Modern Computer Science
- English mathematician
- Studied abstract machines called **Turing machines** even before computers existed

Theory of Computation: A historical perspective

1930s	• Alan Turing studies Turing machines • Decidability • Halting problem
1940-1950s	• "Finite automata" machines studied • Noam Chomsky proposes the "Chomsky Hierarchy" for formal languages
1969	Cook introduces "intractable" problems or "NP-Hard" problems
1970-	Modern computer science: compilers, computational & complexity theory evolve

Theory of Computation, nowadays

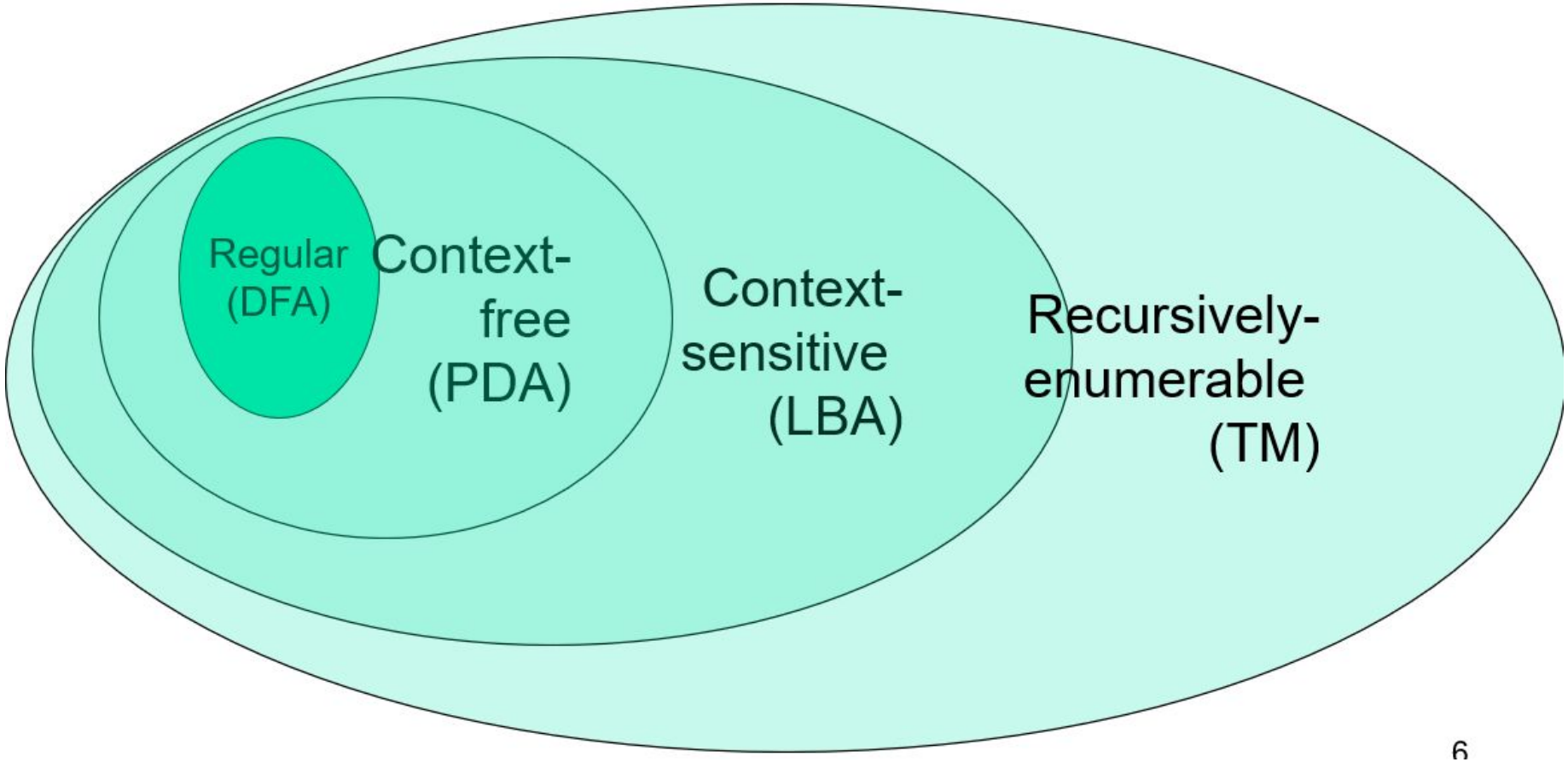
- Nowadays, the Theory of Computation can be divided into the following three areas:
- **Complexity theory**: What makes some problems computationally **hard** and other problems **easy**?
- **Computability theory**: Classify problems as being **solvable** or **unsolvable**.
- **Automata theory**: It deals with definitions and properties of different types of **computation models**.

Computation models

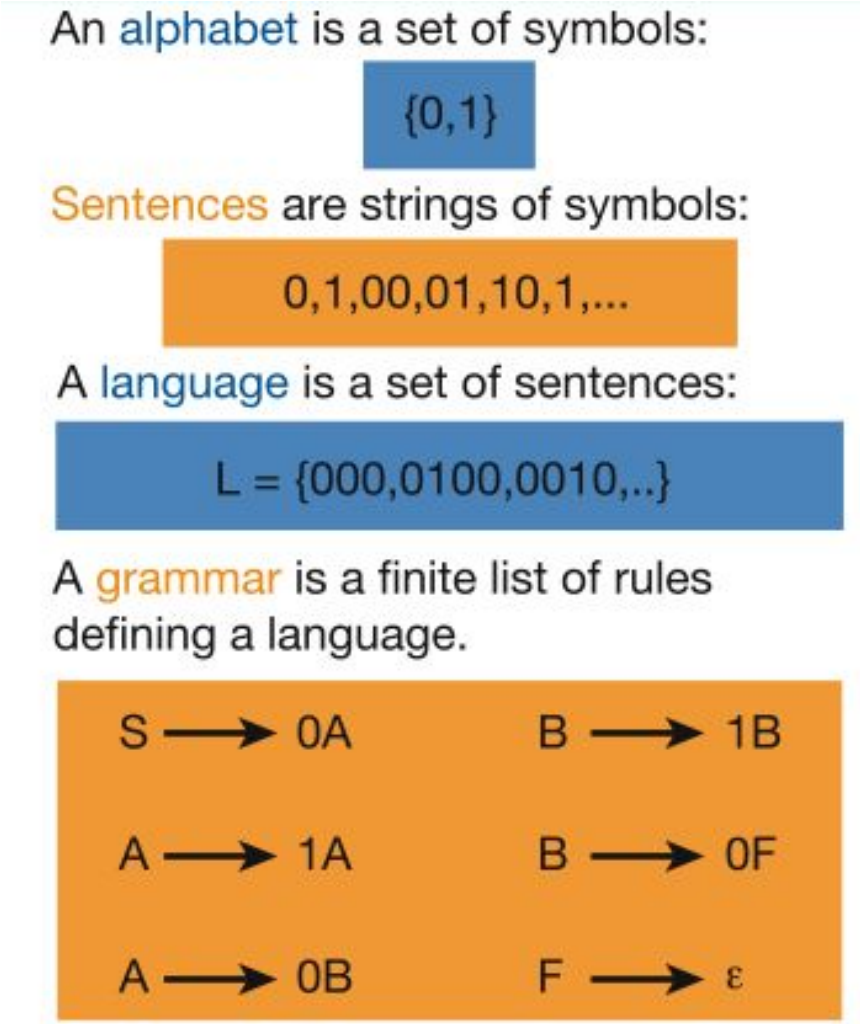
- **Finite Automata**: These are used in text processing, compilers, and hardware design.
- **Context-Free Grammars**: These are used to define programming languages and in Artificial Intelligence.
- **Turing Machines**: These form a simple abstract model of a “real” computer, such as our PC at home.

The Chomsky hierarchy

- A hierarchy of classes of formal languages



Languages & Grammars



- Languages: *"A language is a collection of sentences of finite length all constructed from a finite alphabet of symbols"*
- Grammars: *"A grammar can be regarded as a device that enumerates the sentences of a language"* - nothing more, nothing less

Reference: N. Chomsky, Information and Control, Vol 2, 1959

Rudimentary Elements of Automata Theory

Alphabet

An alphabet is a finite, non-empty set of symbols

- We use the symbol $\sum$ (sigma) to denote an alphabet
- Examples:
  - Binary: $\sum$ = {0,1}
  - All lower case letters: $\sum$ = {a, b, c, ..z}
  - Alphanumeric: $\sum$ = {a-z, A-Z, 0-9}
  - DNA molecule letters: $\sum$ = {a, c, g, t}
  - …

Strings

A string or word is a finite sequence of symbols chosen from $\sum$

- **Empty string is ε (or "epsilon")**

- Length of a string w, denoted by "**|w|**", is equal to the number of (non- ε) characters in the string
  - *E.g., x = 010100* $|x| = 6$
  - *x = 01 ε 0 ε 1 ε 00 ε* $|x| = ?$
  - *xy = concatenation* of two strings *x* and *y*

Powers of an alphabet

Let $\sum$ be an alphabet.

- $\sum^k$ = the set of all strings of length k
- $\sum^* = \sum^0 \cup \sum^1 \cup \sum^2 \cup \ldots$
- $\sum^+ = \sum^1 \cup \sum^2 \cup \sum^3 \cup \ldots$

Languages

L is a said to be a language over alphabet ∑, only if L ⊆ ∑*

- this is because ∑* is the set of all strings (of all possible length including 0) over the given alphabet ∑

Examples:

1. Let L be *the* language of all strings consisting of *n* 0's followed by *n* 1's:
   L = {ε, 01, 0011, 000111,…}
2. Let L be *the* language of all strings of with equal number of 0's and 1's:
   L = {ε, 01, 10, 0011, 1100, 0101, 1010, 1001,…}

Canonical ordering of strings in the language

Definition: Ø denotes the Empty language

- Let L = {ε}; Is L=Ø?

The membership problem

Given a string w $\in \sum^*$ and a language L over $\sum$, decide whether or not w $\in$ L.

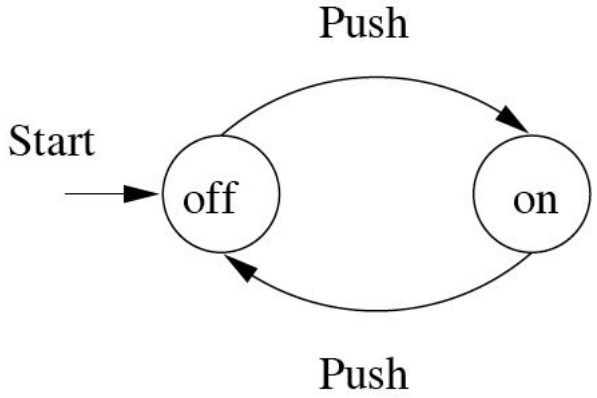
<u>Example:</u>

Let w = 100011

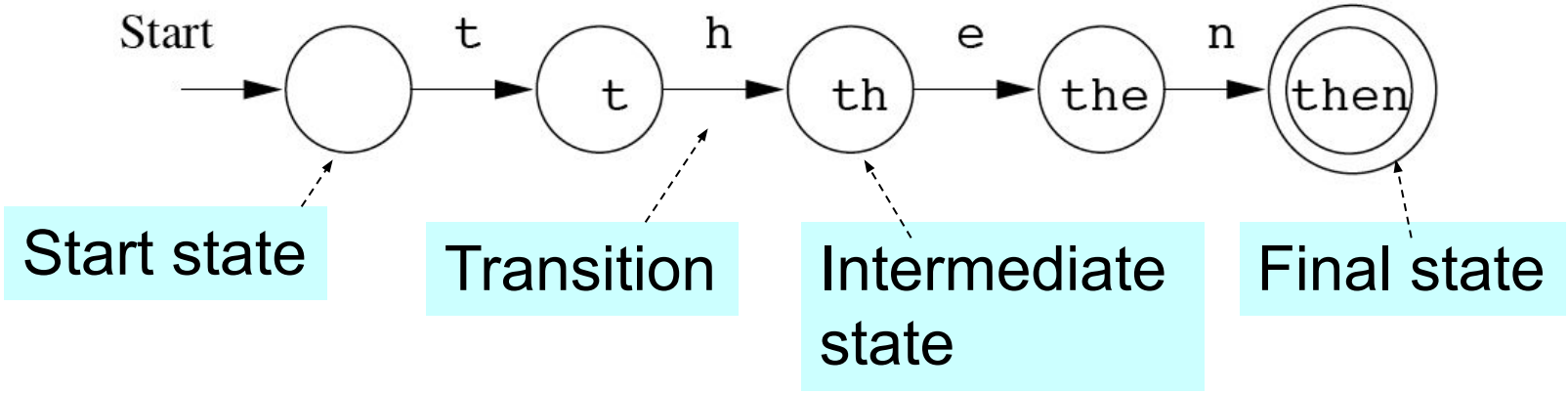
Q) Is w $\in$ the language of strings with equal number of 0s and 1s?

Finite Automata : Examples

- On/Off switch



- Modeling recognition of the word "**then**"



Finite Automata

- Some Applications
  - Software for designing and checking the behavior of digital circuits.
  - Lexical analyzer of a typical compiler.
  - Software for scanning large bodies of text (e.g., web pages) for pattern finding.
  - Software for verifying systems of all types that have a finite number of states (e.g., stock market transaction, communication/network protocol).

Formal Proofs

Definitions, Theorems, Lemma and Corollary

- **Definition**: A precise and unambiguous description of the meaning of a mathematical term. It characterizes the meaning of a word by giving all the properties and only those properties that must be true.
- **Theorem**: A mathematical statement that is proved using rigorous mathematical reasoning. In a mathematical paper, the term theorem is often reserved for the most important results.
- **Lemma**: A minor result whose sole purpose is to help in proving a theorem. It is a stepping stone on the path to proving a theorem. It is an intermediate result that we show to prove a larger result.

Definitions, Theorems, Lemma and Corollary

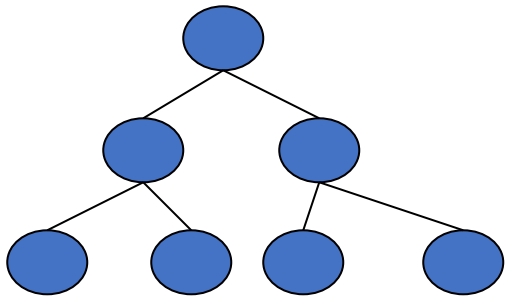
- **Corollary**: A result in which the proof relies heavily on a given theorem. It is a result that follows from an already proven result. (“this is a corollary of Theorem A”).

An example:

Theorem: *The height of an n-node binary tree is at least floor(log n)*

Lemma: *Level i of a complete binary tree has 2^i nodes.*

Corollary: *A complete binary tree of height h has 2^{h+1}-1 nodes.*



Proof by contradiction

- We assume that the **theorem is false** and then show that this assumption leads to an **obviously false consequence**, called a contradiction.

Proof by induction

- For each positive integer **n**, let **P(n)** be a mathematical statement that depends on **n**.
- Assume we wish to prove that **P(n)** is true for all positive integers **n**.

STEPS:

Basis: Prove that **P(1)** is true.

Induction step: Prove that for all **n ≥ 1**, the following holds: If **P(n)** is true, then **P(n + 1)** is also true.

- In the induction step, **we choose an arbitrary integer n ≥ 1 and assume that P(n) is true;** this is called the **induction hypothesis**. Then we prove that P(n + 1) is also true.

Proof by induction

- **Theorem**: For all positive integers n, we have

$$1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$$

- **Proof**:
- We start with the **basis** of the induction. If n = 1, then the left-hand side is equal to 1, and so is the right-hand side. So the theorem is true for n = 1.
- For the **induction** step, let **n ≥ 1** and assume that the theorem is true for **n.**
- We have to prove that the theorem is true for n + 1.

$$1 + 2 + 3 + \ldots + (n+1) = \frac{(n+1)(n+2)}{2}$$

Proof by induction

$$
\begin{aligned}
1+2+3+\ldots+(n+1) &= \underbrace{1+2+3+\ldots+n}_{=\frac{n(n+1)}{2}}+(n+1) \\
&= \frac{n(n+1)}{2}+(n+1) \\
&= \frac{(n+1)(n+2)}{2}.
\end{aligned}
$$

- An alternative proof of the theorem: Let S = 1 + 2 + 3 + . . . + n. Then,

$$
\begin{array}{rcccccccccccc}
S & = & 1 & + & 2 & + & 3 & + & \ldots & + & (n-2) & + & (n-1) & + & n \\
S & = & n & + & (n-1) & + & (n-2) & + & \ldots & + & 3 & + & 2 & + & 1 \\
\hline
2S & = & (n+1) & + & (n+1) & + & (n+1) & + & \ldots & + & (n+1) & + & (n+1) & + & (n+1)
\end{array}
$$

- Since there are n terms on the right-hand side, we have 2S = n(n+1). This implies that S = n(n + 1)/2.

Finite Automata

(1)

Finite State Machine (Prerequisites)

Symbol - a, b, c ... / 0, 1, 2, 3, ...

Alphabet - Σ - Collection of Symbols.

Eg. {a, b}, {d, e, f, g}

{0, 1, 2} ....

String - Sequence of Symbols. Eg. a, b, 0, 1, aa, bb, ab, 01 ...

Language - Set of Strings. Eg. $\Sigma = \{0, 1\}$

Suppose:

L_1 = set of all Strings of length 3

$= \{00, 01, 10, 11\}$ → Finite Set

L_2 = set of all strings that begin with 0

$= \{0, 00, 01, 000, 001, 010, 011, 0000, \ldots\}$ → Infinite set

(2) Finite State Machine (Prerequisite)

Powers of Σ:

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

Σ^0 = set of all strings of length 0: $\Sigma^0 = \{\epsilon\}$

Σ^1 = set of all strings of length 1: $\Sigma^1 = \{0, 1\}$

Σ^2 = " " " " length 2: $\Sigma^2 = \{00, 01, 10,$

$\vdots$

Σ^n = set of all strings of length n

Cardinality – Number of elements in a set

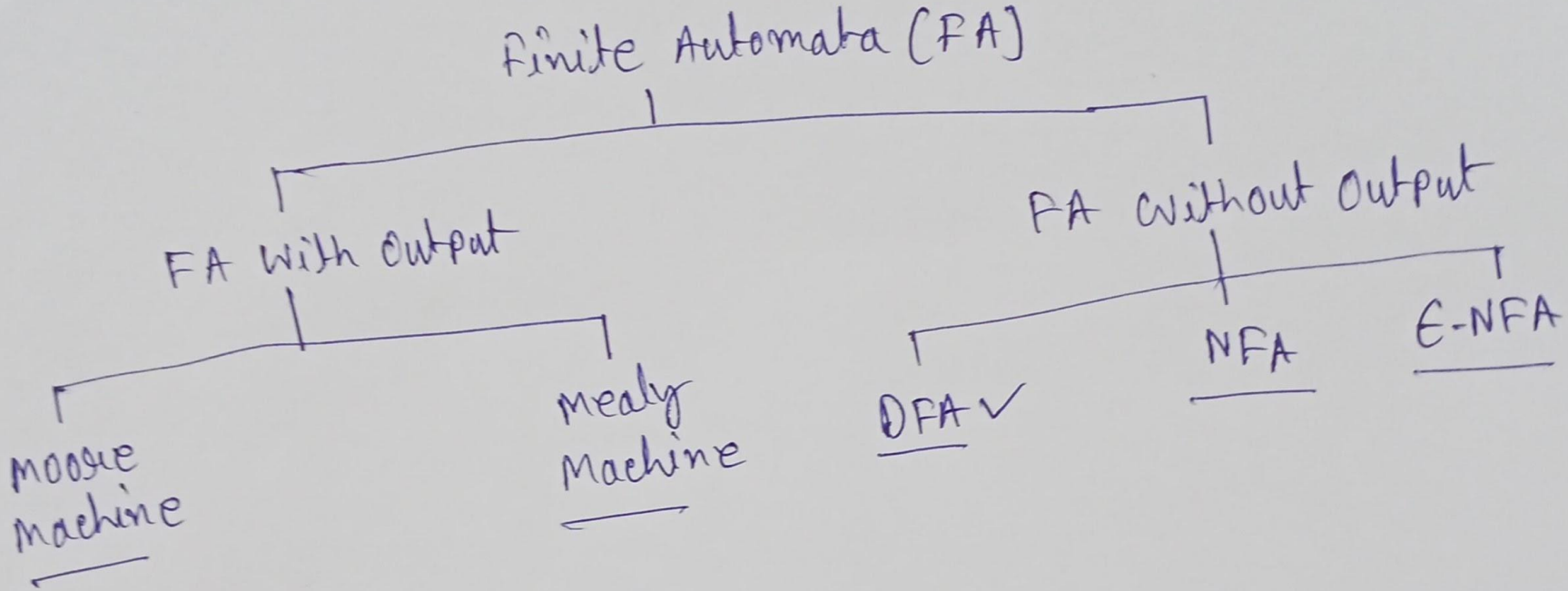
$$\boxed{|\Sigma^n| = 2^n}$$

$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3$

$= \{\epsilon\} \cup \{0, 1\} \cup \{00, 01, 10, 11\} \cup \ldots$

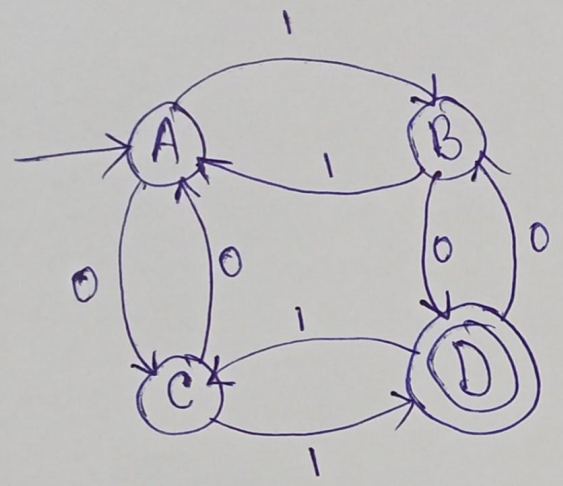
= Set of all possible strings of all lengths over $\{0, 1\}$ → Infinite set.

③ Finite State machine



DFA - Deterministic Finite Automata
For finite state machine (properties)
- It is a simplest model of Computation
- It has a very limited memory

DFA



Structure of a DFA

⇒ DFA Can be defined using 5 tuples :-

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

Q = set of all states

Σ = inputs

q_0 = Start state/initial State

F = Set of final States

δ = transition function from $Q \times \Sigma \to Q$

⇓

$Q = \{A, B, C, D\}$

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

$q_0 = A$

$F = \{D\}$

	0	1
A	C	B
B	D	A
C	A	D
D	B	C

⇓

⇒ $\delta(A, 0) = C$

$\delta(A, 1) = B$

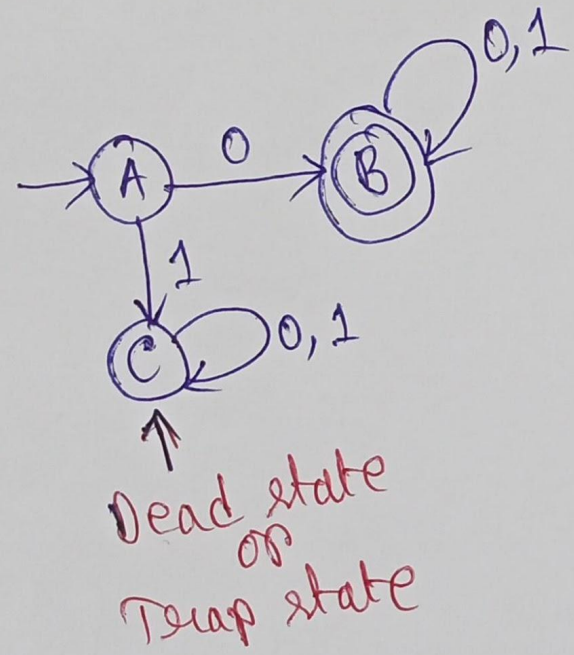
$\delta(B, 0) = D$

$\delta(D, 1) = C.$

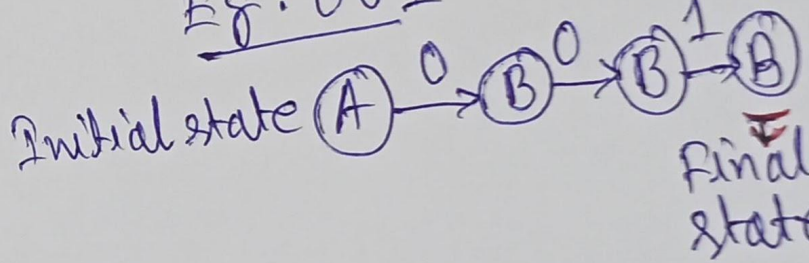
Deterministic Finite Automata (Example-1)

L1 = Set of all strings that starts with '0'

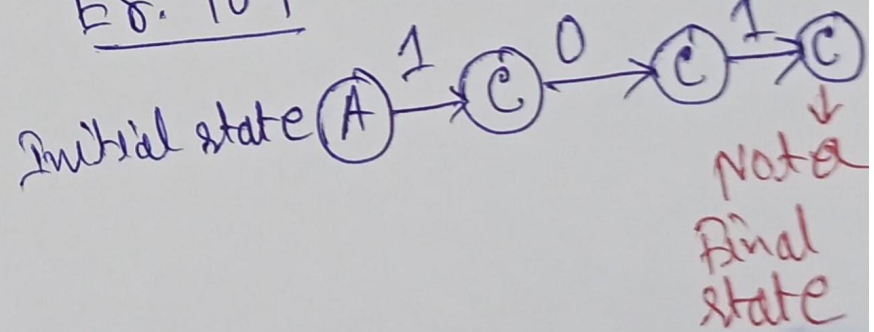
= {0, 00, 01, 000, 010, 011, 0000, ... }



Eg. 001



Eg. 101

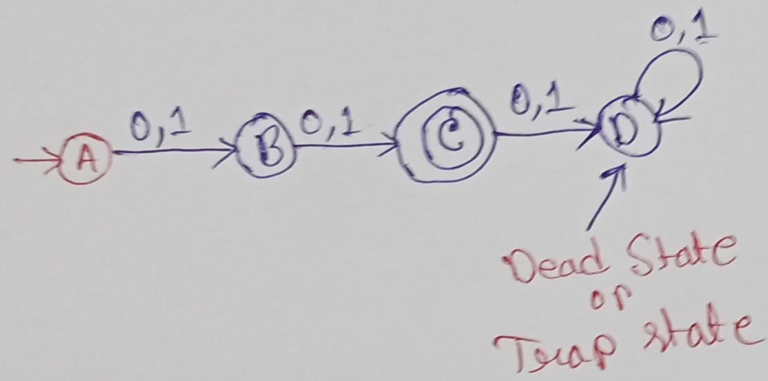


⑥ DFA (Example-2)

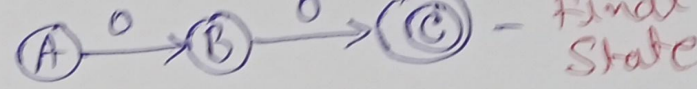
⇒ Construct a DFA that accepts sets of all strings over {0, 1} of length 2.

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

$L = \{00, 01, 10, 11\}$

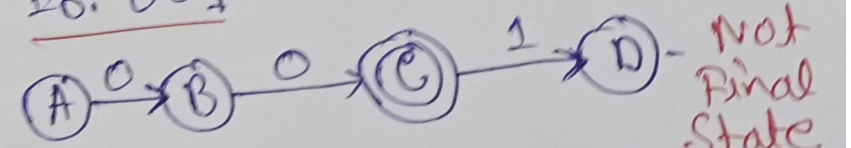


Eg. 00

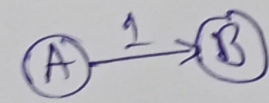


Eg. 10

Eg. 001



Eg. 1



⑦ DFA (Example-3)

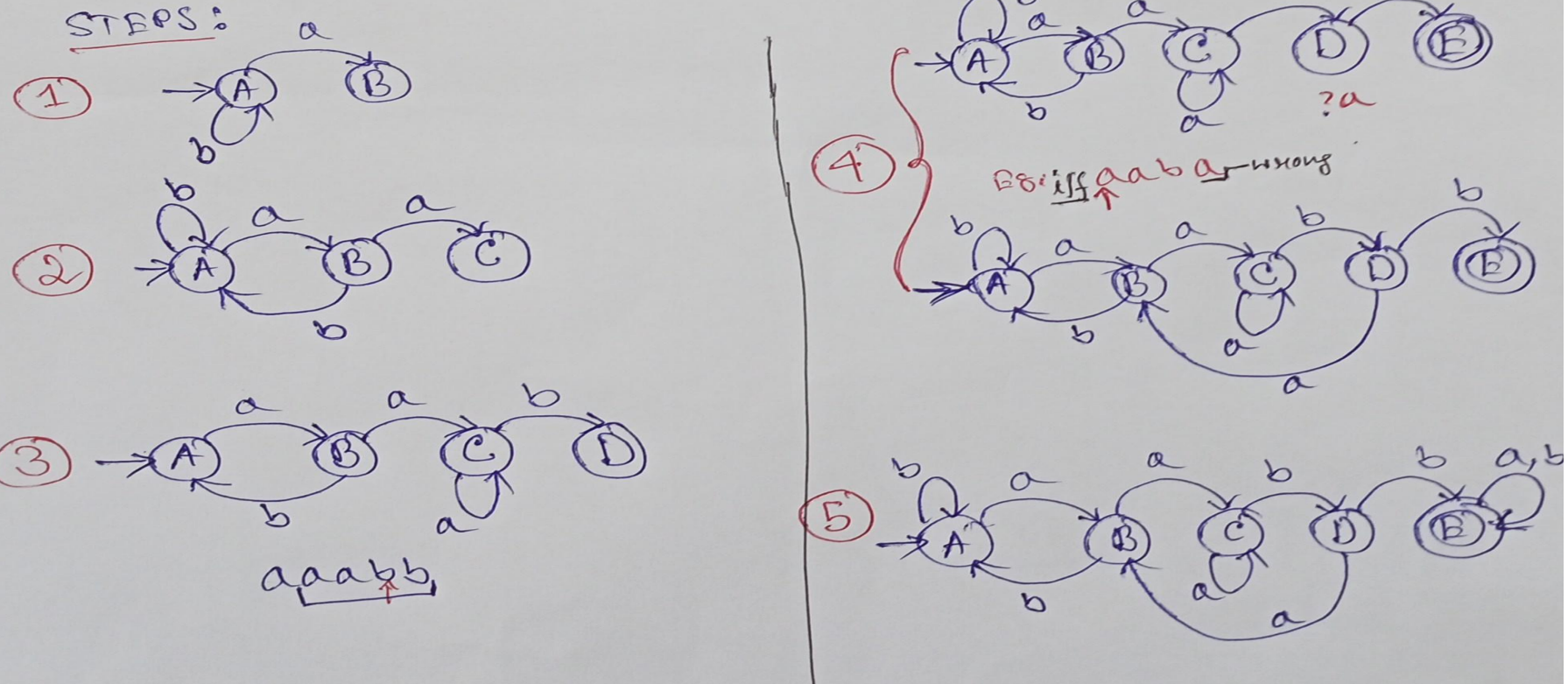
⇒ Construct a DFA that accepts any strings over {a, b} that does not contain the string aabb in it.

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

Try to design a simpler problem

⇒ Let us construct a DFA that accepts all strings over {a, b} that contains the string aabb in it.

STEPS:



(8) DFA (Example 3)

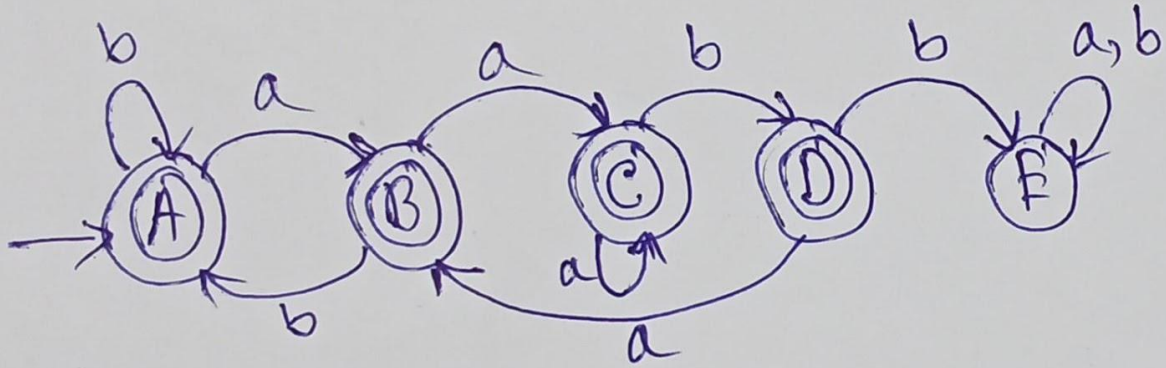
⇒ Construct a DFA that accepts any strings over {a, b} that does not contain the string aabb in it.

⇒ Flip the States

– make the final state into non-final state.

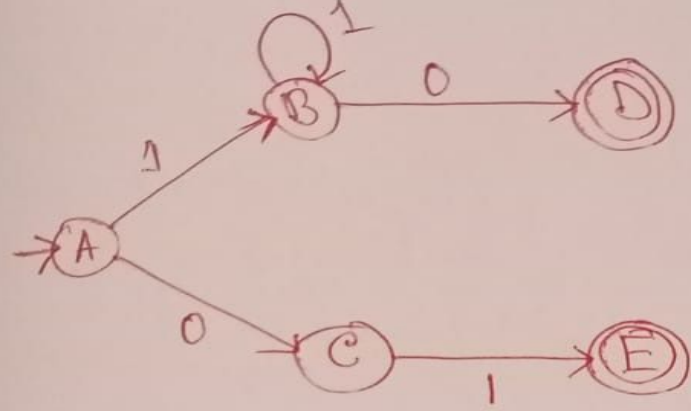
and

– make the non-final states into final state.



DFA (Example-4)

⇒ How to figure out what a DFA recognizes?



10 ✓

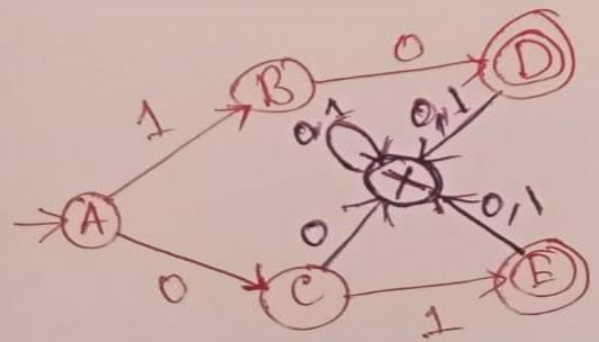
1 1111 0 ✓ } Atleast one binary digit '1'

A B D

01 ✓

L = { Accepts the string 01 or a string of atleast one '1' followed by a '0' }

Eg. 001, 010, 011, 1101, 1100



→ Ok, Complete.

Thank You

Regular Languages

Regular Languages

⇒ A language is said to be a Regular Language if and only if some finite state machine recognizes it.

So what languages are not Regular?

The languages
» which are not recognized by any FSM.
» which requires memory.

⇒ when a language not recognized by any FSM?
Ans:- when a language requires memory.

Because of,
- memory of FSM is very limited.
→ It cannot store or count strings

Eg. abababbabb

store required

$a^N b^N$
aaabbb [count required]
aaaabbbb

NOT REGULAR

Operations on Regular Languages

UNION — $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$

CONCATENATION — $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$

STAR — $A^* = \{x_1 x_2 x_3 \ldots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$

Eg. $A = \{pq, r\}$, $B = \{t, uv\}$

$A \cup B = \{pq, r, t, uv\}$

$A \circ B = \{pqt, pquv, rt, ruv\}$

$A^* = \{\epsilon, pq, r, pqr, rpq, pqpq, rr, pqpqpq, rrr, \ldots\}$

Theorem 1: The class of Regular languages is closed under UNION.

Theorem 2: The class of Regular languages is closed under CONCATENATION.

DFA (Excercise)

⇒ For each of the following language, construct a DFA that accepts the language, In all cases alphabet is $\{0, 1\}$.

① $\{w : w$ is a binary string containing an odd number of 1's$\}$

② $\{w : w$ is a binary string containing 101 as substring$\}$

③ $\{w :$ the length of w is divisible by three$\}$

④ $\{w :$ 110 is not a substring of $w\}$

⑤ $\{w : w$ contains the substring 1011$\}$

⑥ $\{w : w$ contains an even number of 0's or exactly two 1's$\}$

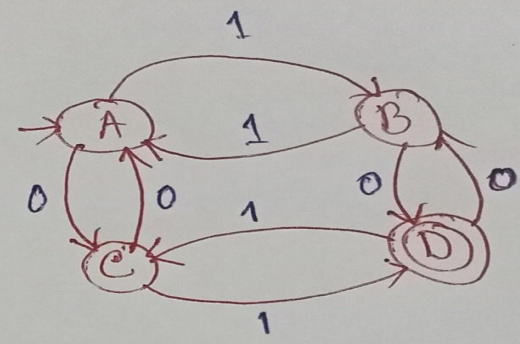
Non-Deterministic FA

Deterministic Finite Automata

↓

DETERMINISM

» In DFA, given the current state we know what the next state will be.



» It has only one unique next state.

» It has no choice or randomness.

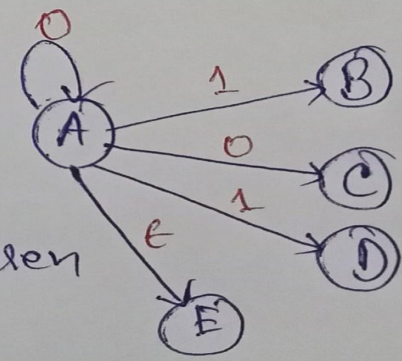
» It is simple and easy to design.

Non-deterministic Finite Automata

↓

NON-DETERMINISM

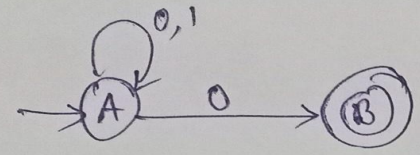
» In NFA, given the current state there could be multiple next states.



» The next state may be chosen at random

» All the next states may be chosen in parallel.

NFA - Formal Definition



L = { Set of all strings that end with 0 }

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

Q = Set of all states — { A, B }
Σ = inputs — { 0, 1 }
q_0 = Start state/initial state — A
F = Set of final states — B
δ = — ϕ

$\delta = Q \times \Sigma \rightarrow 2^Q$

↓

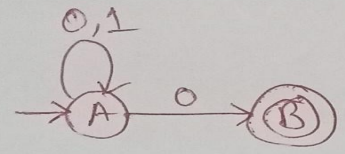
$A \times 0 \rightarrow A$ ✓
$A \times 0 \rightarrow B$ ✓
$A \times 1 \rightarrow A$
$B \times 0 \rightarrow \phi$
$B \times 1 \rightarrow \phi$

more than one state

Ex.
⇒ $A \xrightarrow{1} A, B, AB, \phi$
$2^2 = 4$
⇒ 3 states - A, B, C

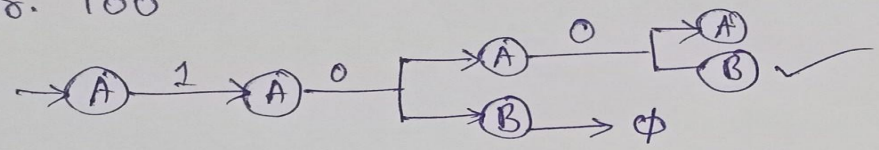
$A \xrightarrow{1} A, B, C, AB, AC, BC, ABC, \phi$
$2^3 = 8$

NFA - Example-1

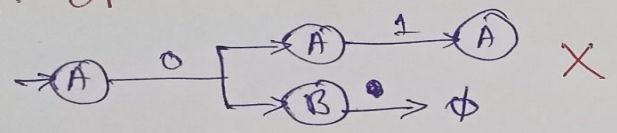


L = { Set of all strings that end with 0 }

Eg. 100



Eg. 01



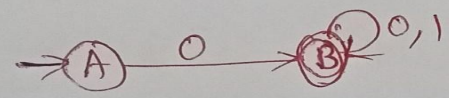
X

⇒ If there is any way to run the machine that ends in any set of states out of which atleast one state is a final state, then the NFA accepts

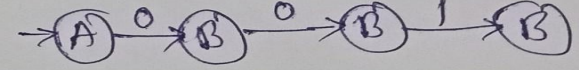
Example-2

L = { Set of all strings that start with 0 }

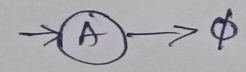
= { 0, 00, 01, 000, - - - }



Eg, 001 ✓



Eg, 101 X



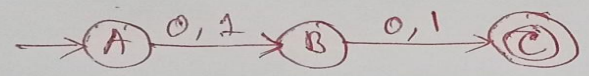
Dead Configuration

NFA – Example – 3

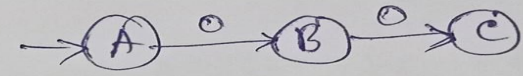
⇒ Construct a NFA that accepts sets of all strings over {0, 1} of length 2

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

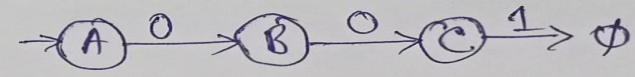
$L = \{00, 01, 10, 11\}$



Ex, 00 ✓

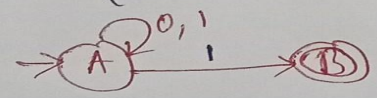


Ex, 001 ✗



Example – 4

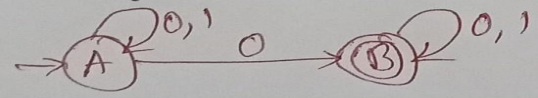
Ex a) $L1 = \{$ Set of all strings that ends with '1' $\}$



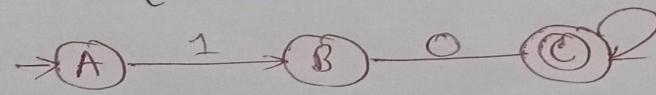
→ 01, 001, 0001 → 0^*1 / 1

→ 101, 1101, ~

Ex b) $L2 = \{$ Set of all strings that contain '0' $\}$

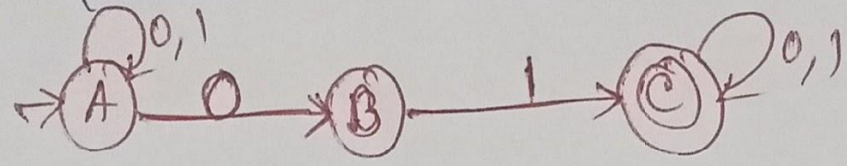


Ex c) $L3 = \{$ Set of all strings that starts with '10' $\}$

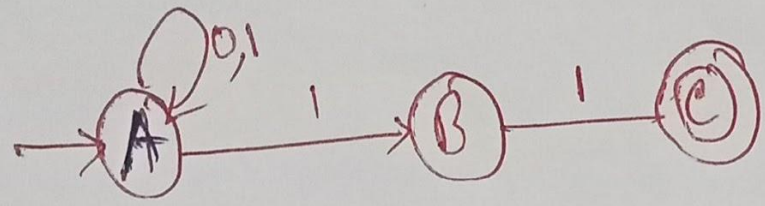


NFA - Example 4

Ex 4) $L4 = \{$ Set of all strings that contain '01' $\}$



Ex 5) $L5 = \{$ Set of all strings that ends with '11' $\}$



Assignment: If you were to construct the equivalent DFAs for the above NFAs, then tell that how many minimum number of states would you use for the construction of each of the DFA's

Conversion of NFA to DFA

Every DFA is an NFA, but not vice versa

But there is an equivalent DFA for every NFA

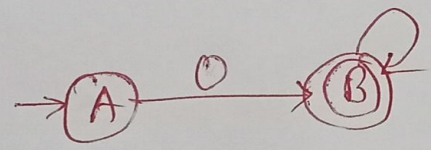
DFA $\delta = Q \times \Sigma \rightarrow Q$

NFA $\delta = Q \times \Sigma \rightarrow 2^Q$

NFA $\cong$ DFA

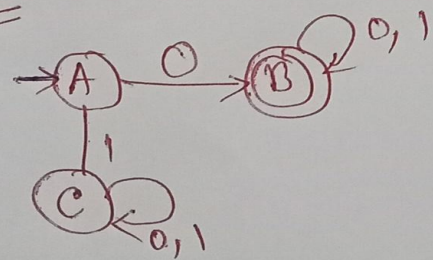
L = { Set of all strings over (0, 1) that starts with '0' }

NFA



	0	1
A	B	ϕ
B	B	B

DFA



	0	1
A	B	C
B	B	B
C	C	C

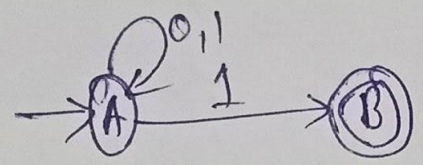
C - Dead state/ Trap state

Conversion of NFA to DFA - Examples

L = {Set of all strings over (0, 1) that ends with '1'}

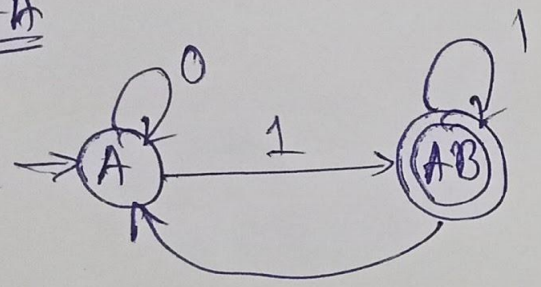
$\Sigma = \{0, 1\}$

NFA



	0	1
A	{A}	{A, B}
B	ϕ	ϕ

DFA



	0	1
A	{A}	{AB}
AB	{A}	{AB}

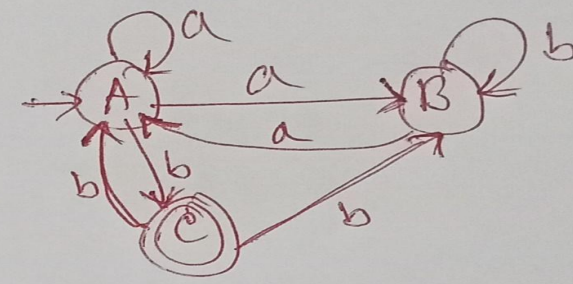
AB - Single State

Subset Construction method

Conversion of NFA to DFA - Example1

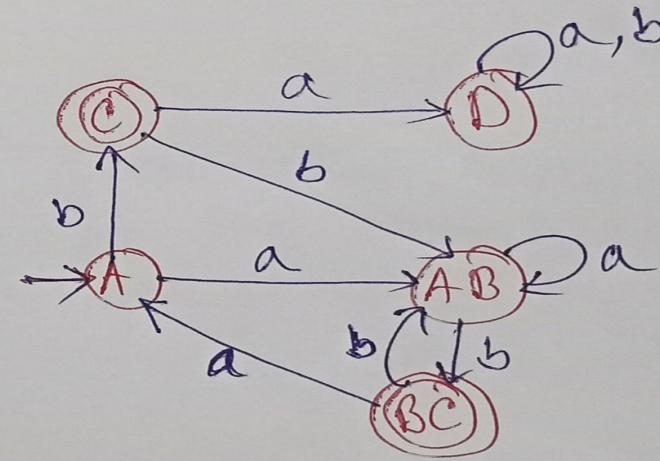
⇒ Find the equivalent DFA for the NFA given by $M = [\{A, B, C\}, (a, b), \delta, A, \{C\}]$ where δ is given by:

	a	b
→A	A, B	C
B	A	B
(C)	—	A, B



DFA

	a	b
→A	AB	C
AB	AB	BC
(BC)	A	AB
(C)	D	AB
D	D	D

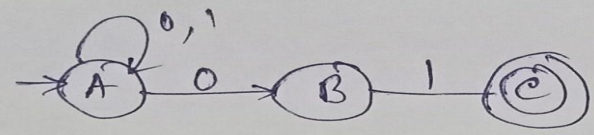


Assignment: Try to find out what does this NFA and its DFA accept.

Conversion of NFA to DFA - Examples

⇒ Given below is the NFA for a language
L = { Set of all strings over (0, 1) that ends with '01' }
Construct its equivalent DFA

NFA



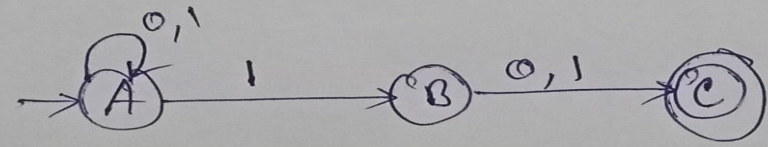
	0	1
→ A	A, B	A
B	ϕ	C
(C)	ϕ	ϕ

DFA ?

⇒ Design an NFA for a language that accepts all strings over {0, 1} in which the second last symbol is always '1'. Then convert it to its equivalent DFA.

Eg. 0010
0110
1100111
0001111

NFA



	0	1
→ A		
B		
(C)		

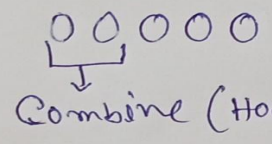
DFA ?

Minimization of DFA

Minimization of DFA

⇒ Minimization of DFA is required to obtain the minimum version of any DFA which consists of the minimum number of states possible.

DFA with 5 States / with 4 States



⇒ Two states can be combine when they are Equivalent.

⇒ Two states 'A' and 'B' are said to be equivalent iff

$\delta(A, x) \rightarrow F$ and $\delta(B, x) \rightarrow F$

OR

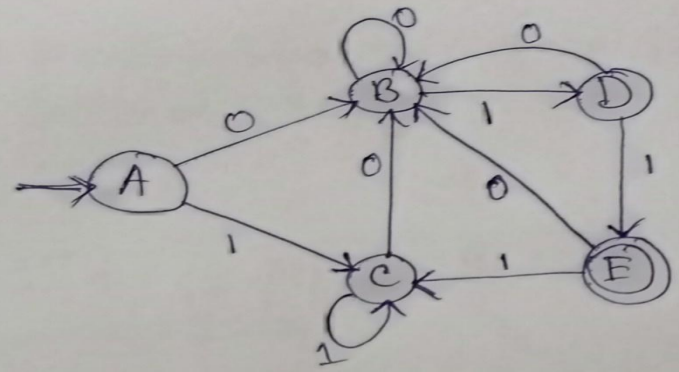
$\delta(A, x) \not\rightarrow F$ and $\delta(B, x) \not\rightarrow F$

where 'x' is any input string

⇒ Types of equivalent

If $|x| = 0$, then A and B are said to be 0 equivalent
If $|x| = 1$, then A and B are said to be 1 equivalent
If $|x| = 2$, then A and B are said to be 2 equivalent
⋮
If $|x| = n$, then A and B are said to be n equivalent

Minimization of DFA - Example1 (Part-1)



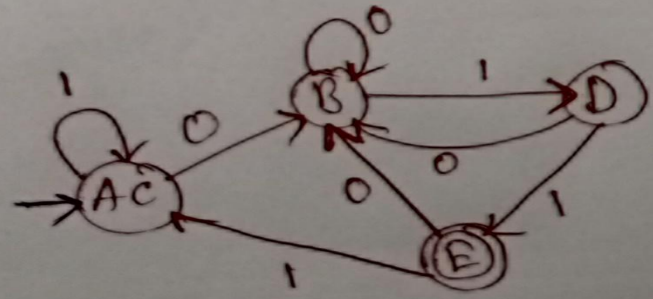
	0	1
→ A	B	C
B	B	D
C	B	C
D	B	E
(E)	B	C

0-Equivalence: {A B C D} {E}

1-Equivalence: {A, B, C} {D} {E} ← [A B ✓ A C ✓ C D ✗]

2-Equivalence: {A C} {B} {D} {E}

3-Equivalence: {A, C} {B} {D} {E}



→ minimal version of above DFA

Partitioning method

Minimization of DFA – Examples (Part-2)

⇒ Construct a minimum DFA equivalent to the DFA described by

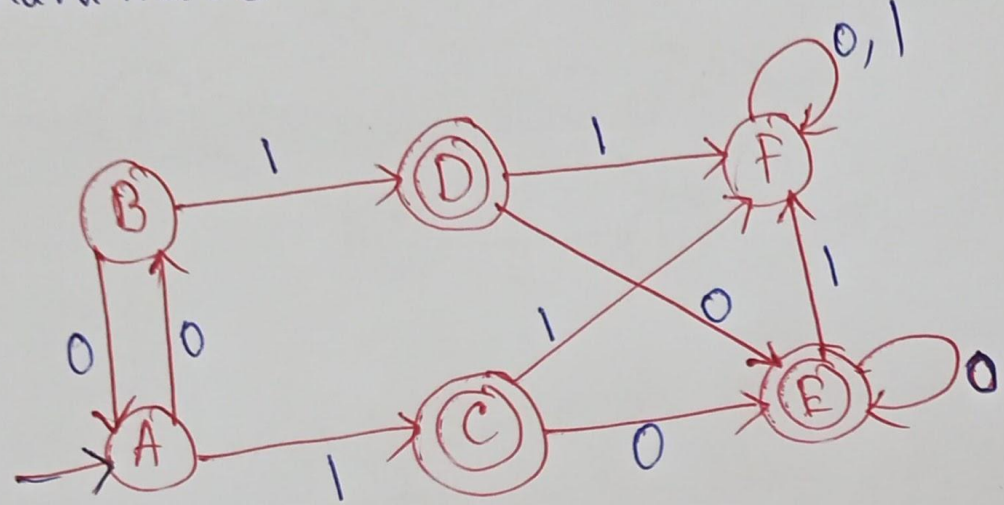
	0	1
→ q_0	q_1	q_5
q_1	q_6	q_2
(q_2)	q_0	q_2
q_3	q_2	q_6
q_4	q_7	q_5
q_5	q_2	q_6
q_6	q_6	q_4
q_7	q_6	q_2

Assignment

Minimization of DFA - Examples (Part-3)

⇒ When there are more than one final states involved

Minimize the following DFA:

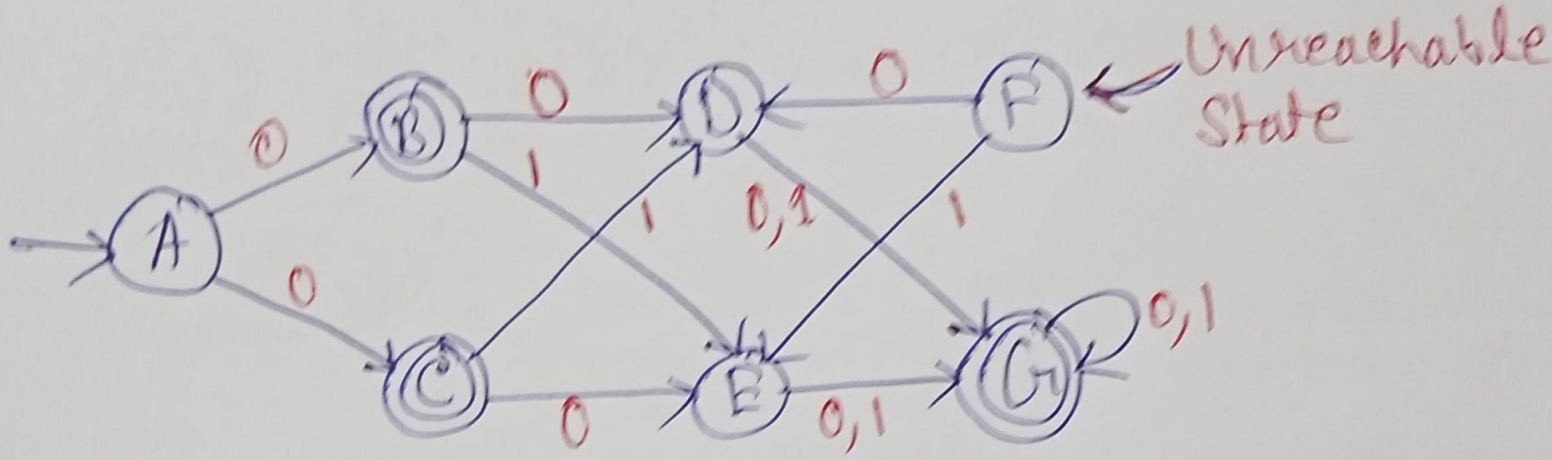


	0	1
→A	B	C
B	A	D
C	E	F
D	E	F
E	E	F
F	F	F

Assignment

Minimization of DFA - Examples

⇒ When there are Unreachable states involved



⇒ A State is said to be Unreachable if there is no way it can be reached from the Initial State.

⇒ Remove the Unreachable state from DFA.

Assignment

ε-NFA

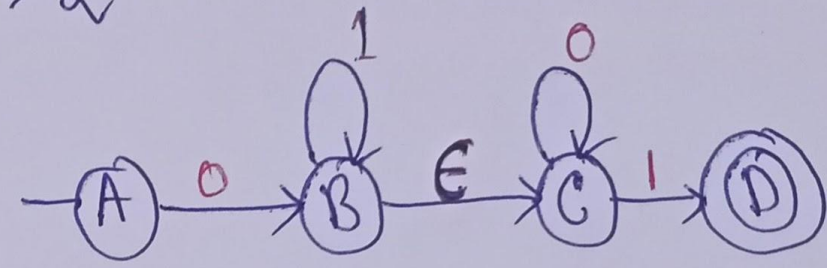
Epsilon (ϵ) - NFA

ϵ - NFA

↳ empty Symbols / Null Symbol

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

$\delta : Q \times \Sigma \cup \epsilon \rightarrow 2^Q$



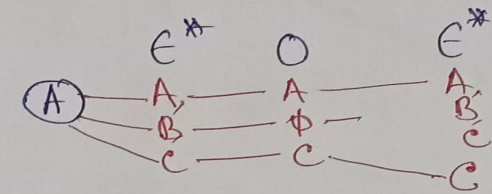
Note: Every state on ϵ goes to itself.

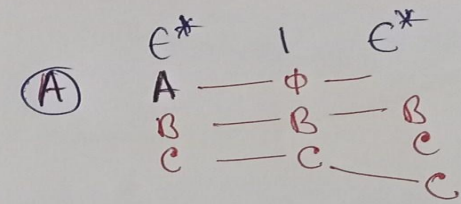
Conversion of ϵ-NFA to NFA

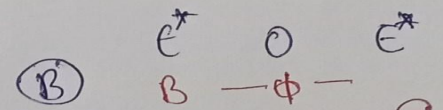
Convert the following ϵ-NFA to its equivalent NFA

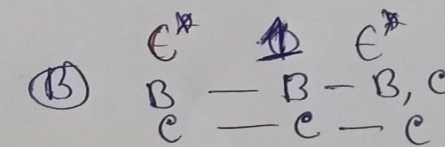
State	ϵ^*	input	ϵ^*

ϵ-closure (ϵ^*) - All the states that can be reached from a particular state only be seeing ϵ Symbol.









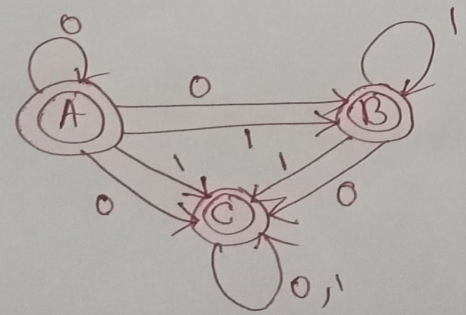
NFA

	0	1
→ A	{A, B, C}	{B, C}
B	{C}	{B, C}
C	{C}	{C}

⇒ Final state?

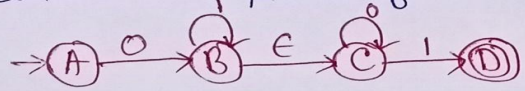
↳ Any state that can reach the final state only by seeing ϵ.

NFA



Conversion of ϵ-NFA to NFA - Example

⇒ Convert the following ϵ-NFA to its equivalent NFA

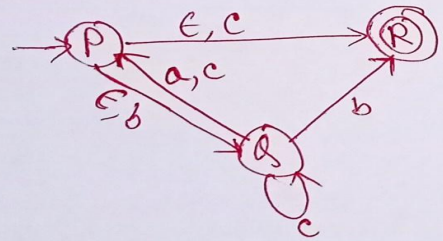


$\epsilon^* 0 \epsilon^*$ $\epsilon^* 1 \epsilon^*$

NFA

	0	1
→ A		
B		
C		
(D)		

⇒ Convert the following ϵ-NFA to its equivalent NFA



NFA

	a	b	c
→ P			
Q			
(R)			

$\epsilon^* a \epsilon^*$ $\epsilon^* b \epsilon^*$ $\epsilon^* c \epsilon^*$

Assignment

Regular Expression

Regular Expression

⇒ Regular Expressions are used for representing certain sets of strings in an algebraic fashion.

① Any terminal symbol i.e. symbols $\in \Sigma$ including $\wedge$ and ϕ are regular expressions. $a, b, c, \ldots \wedge, \phi$

② The union of two regular expressions is also a regular expression. $R_1, R_2 \rightarrow (R_1 + R_2)$

③ The Concatenation of two regular expressions is also a regular expression. $R_1, R_2 \rightarrow (R_1 . R_2)$

④ The iteration (or closure) of a regular expression is also a regular expression. $R \rightarrow R^*$ $a^* = \wedge, a, aa \ldots$

⑤ The regular expression over Σ are precisely those obtained recursively by the application of the above rules once or several times.

Regular Expression - Examples

⇒ Describe the following sets as Regular Expressions

1) $\{0, 1, 2\}$ 0 or 1 or 2

$R = 0 + 1 + 2$

2) $\{\wedge, ab\}$

$R = \wedge ab$

3) $\{abb, a, b, bba\}$ abb or a or b or bba

$R = abb + a + b + bba$

4) $\{\wedge, 0, 00, 000, ---\}$ Closure of 0

$R = 0^*$

⑤ $\{1, 11, 111, 1111, ----\}$

$R = 1^+$

Identities of Regular Expression

1) $\phi + R = R$

2) $\phi R + R\phi = \phi$

3) $\epsilon R = R\epsilon = R$

4) $\epsilon^* = \epsilon$ and $\phi^* = \epsilon$

5) $R + R = R$

6) $R^* R^* = R^*$

7) $RR^* = R^*R$

8) $(R^*)^* = R^*$

9) $\epsilon + RR^* = \epsilon + R^*R = R^*$ → $R^+ \cup \epsilon = R^*$

10) $(PQ)^*P = P(QP)^*$

11) $(P+Q)^* = (P^*Q^*)^* = (P^* + Q^*)^*$

12) $(P+Q)R = PR + QR$ and
$R(P+Q) = RP + RQ$

ARDEN'S THEOREM

⇒ If P and Q are Two Regular Expressions over Σ, and if P does not Contain ϵ, then the following equation in R given by $\boxed{R = Q + RP}$ has a unique solution i.e. $\boxed{R = QP^*}$

$$R = Q + RP \longrightarrow (1)$$
$$= Q + QP^*P$$
$$= Q(\epsilon + P^*P)$$
$$= QP^*$$

$R = QP^*$

$[\epsilon + R^*R = R^*]$

Proved that $R = QP^*$ is a ~~soutu~~ solution.

⇒ Prove that $R = QP^*$ is a unique solution.

$$R = Q + RP$$
$$= Q + [Q + RP]P$$
$$= Q + QP + RP^2$$
$$= Q + QP + [Q + RP]P^2$$
$$= Q + QP + QP^2 + RP^3$$
$$\vdots$$
$$= Q + QP + QP^2 + \cdots + QP^n + RP^{n+1}$$

$[R = QP^*]$

$$= Q + QP + QP^2 + \cdots + QP^n + QP^*P^{n+1}$$
$$= Q[\epsilon + P + P^2 + \cdots + P^n + P^*P^{n+1}]$$
$$R = QP^*$$

An Example Proof using Identities of RE

⇒ Prove that $(1+00^*1)+(1+00^*1)(0+10^*1)^*(0+10^*1)$ is equal to $0^*1(0+10^*1)^*$

$$LHS = (1+00^*1)+(1+00^*1)(0+10^*1)^*(0+10^*1)$$

$$= (1+00^*1)\left[\epsilon+(0+10^*1)^*(0+10^*1)\right]$$

$\epsilon + R^*R = R^*$

$$= (1+00^*1)(0+10^*1)^*$$

$$= (\epsilon.1+00^*1)(0+10^*1)^*$$

$\epsilon . R = R$

$$= (\epsilon+00^*)1(0+10^*1)^*$$

$\epsilon + R^*R = R^*$

$$= 0^*1(0+10^*1)^* = RHS$$

Designing RE - Examples

⇒ Design RE for the following languages over {a, b}

1) Language accepting strings of length exactly 2
2) Language accepting strings of length atleast 2
3) Language accepting strings of length atmost 2

Soln

1) $L_1 = \{aa, ab, ba, bb\}$

$R = aa + ab + ba + bb$

$= a(a+b) + b(a+b)$

$= (a+b)(a+b)$

2) $L_2 = \{aa, ab, ba, bb, aaa, \ldots\}$

$R = (a+b)(a+b)(a+b)^*$ $\quad *$ = 0, 1, 2, 3 ...

3) $L_3 = \{\epsilon, a, b, aa, ab, ba, bb\}$

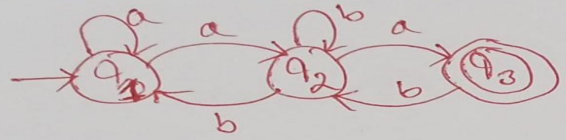
$R = \epsilon + a + b + aa + ab + ba + bb$

$= (\epsilon + a + b)(\epsilon + a + b)$

Inter-conversion (RE & FA)

Designing Regular Expression - Examples

⇒ Find the RE for the following NFA



$q_3 = q_2 a \longrightarrow (1)$

$q_2 = q_1 a + q_2 b + q_3 b \longrightarrow (2)$

$q_1 = \epsilon + q_1 a + q_2 b \longrightarrow (3)$

(1) → $q_3 = q_2 a$

$= (q_1 a + q_2 b + q_3 b)\, a$

$= q_1 aa + q_2 ba + q_3 ba \longrightarrow (4)$

(2) → $q_2 = q_1 a + q_2 b + q_3 b$ Putting value of q_3 from (1)

$= q_1 a + q_2 b + (q_2 a) b$

$= q_1 a + q_2 b + q_2 ab$

$q_2 = q_1 a + q_2 (b + ab)$

$R = Q + RP$

$R = QP^*$ [Arden's Theorem]

$q_2 = q_1 a (b+ab)^* \longrightarrow (5)$

(3) → $q_1 = \epsilon + q_1 a + q_2 b$

Putting value of q_2 from (5)

$q_1 = \epsilon + q_1 a + ((q_1 a)(b+ab)^*)\, b$

$q_1 = \epsilon + q_1 (a + a(b+ab)^* b)$

$$q_1 = \epsilon(a+(a(b+ab)^*)b)^* \qquad \epsilon \cdot R = R$$

$$q_1 = a+a(b+ab)^*b)^* \longrightarrow (6)$$

Final state $\rightarrow (q_3)$

$$q_3 = q_2 a$$

Putting the value of q_2 from (5)

$$q_3 = q_1 a(b+ab)^* a$$

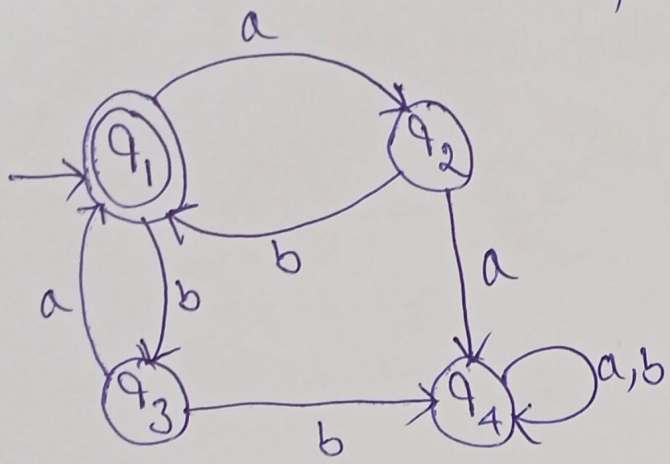
putting the value of q_1 from (6)

$$q_3 = (a+a(b+ab)^*b)^* a(b+ab)^* a$$

= Required RE for the given NFA

Designing Regular Expression – Examples

Find the RE for the following DFA



$q_1 = \epsilon + q_2 b + q_3 a \rightarrow (I)$

$q_2 = q_1 a \rightarrow (II)$

$q_3 = q_1 b \rightarrow (III)$

$q_4 = q_2 a + q_3 b + q_4 a + q_4 b \rightarrow (IV)$

(I) $\rightarrow$ $q_1 = \epsilon + q_2 b + q_3 a$

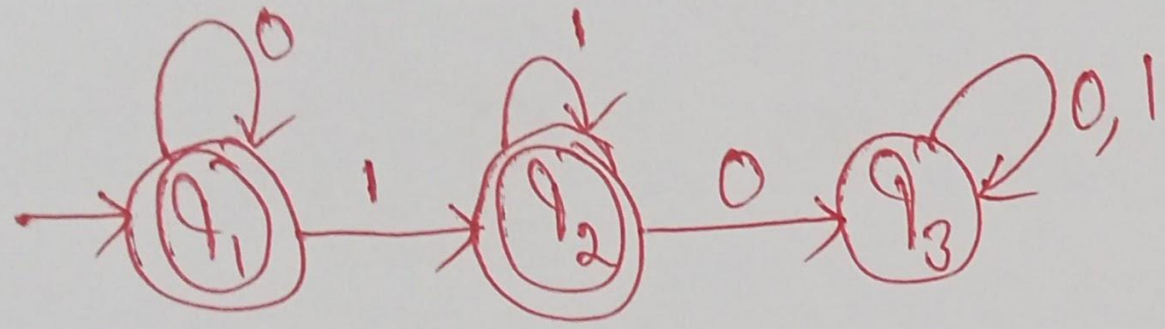
Putting values of q_2 and q_3 from (II) and (III)

Assignment

Designing RE - Examples

(when there are multiple Final States)

Find the RE for the following DFA

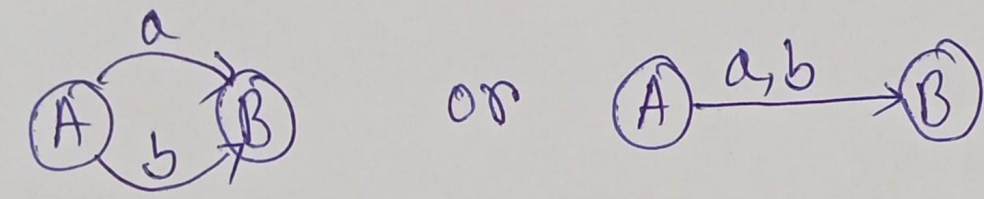


Assignment

Conversion of RE to Finite Automata

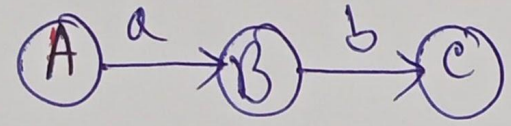
Important Rules →

⇒ (a+b)

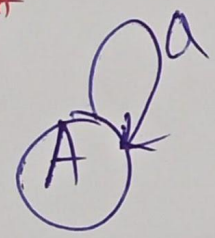


or

⇒ (a.b)



⇒ a^*

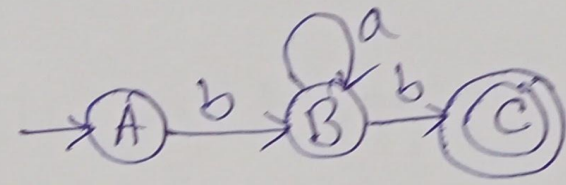


Conversion of RE to FA - Examples

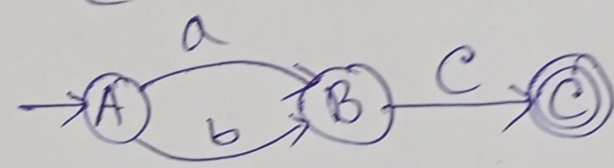
Convert the following RE to their equivalent FA:

1. ba^*b
2. $(a+b)c$
3. $a(bc)^*$

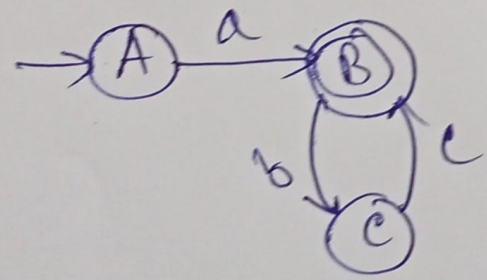
① ba^*b → bb, bab, baab.



② $(a+b)c$ → ac, bc



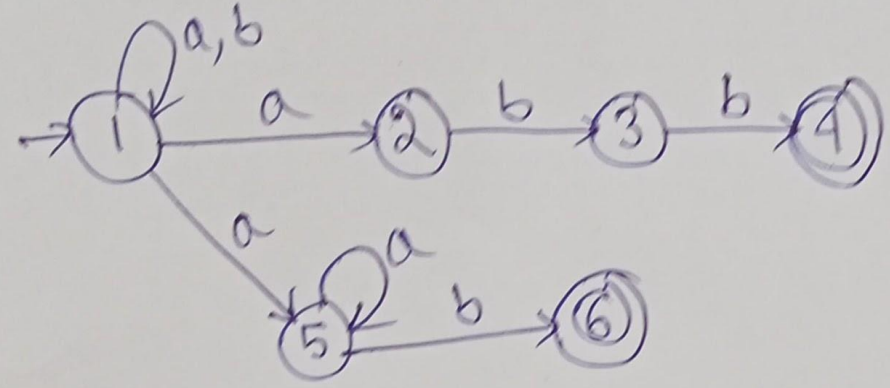
③ $a(bc)^*$ → a, abc, abcbc, abcbcbc, ...



Conversion of RE to FA - Examples

⇒ Convert the following RE to its equivalent FA:

$(a|b)^*(abb|a^+b)$ $\boxed{+/|}$



$\therefore a^+ = \{a, aa, aaa, \ldots\}$

$a^* = \{\epsilon, a, aa, aaa, \ldots\}$

⇒ Convert the following RE to its equivalent FA:

$10+(0+11)0^*1$

Assignment.

Thank You

Pumping Lemma for RE

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 $S = xyz$ such that the following Conditions must be true.

1) $xy^iz \in A$ for every $i \geq 0$

2) $|y| > 0$

3) $|xy| \leq p$

Pumping Lemma (For Regular Expression)

⇒ To prove that a language is not Regular using Pumping Lemma, follow the below steps:

(we prove using Contradiction)

→ Assume that A is Regular

→ It has to have a pumping Length (say p)

→ All strings longer than p can be pumped $|S| \geq p$

→ Now find a string 'S' in A such that $|S| \geq p$

→ Divide S into xyz

→ Show that $xy^iz \notin A$ for some i

→ Then Consider all ways that S can be divided into xyz.

→ Show that none of these can satisfy all the 3 pumping conditions at the same time

→ S can not be pumped = Contradiction

Pumping Lemma (For Regular Languages)-Example

⇒ Using pumping Lemma prove that the language $A = \{a^n b^n \mid n \geq 0\}$ is not Regular

Proof

→ Assume that A is Regular

→ It need to have a pumping length, say P

→ Choose string, $S = a^P b^P$

x y z

→ For example, P = 7 , ⇒ S = aaaaaaa bbbbbbb

→ All the possible ways to divide S are as follows.

Case 1: The y is in the 'a' part

a a a a a a a b b b b b b b

x y z

→ $xy^iz \Rightarrow xy^2z$ X

aa aaaa aaaaab bbbbbbb

11 ≠ 7

Case 2: The y is in the 'b' part

a a a a a a a b b b b b b b

x y z

→ $xy^iz \Rightarrow xy^2z$ X

aaaaaaabb bbbbbbbbb

7 ≠ 11

Case 3: The y is in the 'a' and 'b' part

a a a a a a a b b b b b b b

x y z

→ $xy^iz \Rightarrow xy^2z$ X

aaaaa aabb aabb bbbbb

not following $a^n b^n$

Check $|xy| \leq P$, P = 7

Pumping Lemma (FOR RL) - Example

⇒ Using pumping Lemma prove that the language $A = \{YY \mid Y \in \{0,1\}^*\}$ is Not Regular

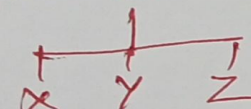
01,01

Proof

→ Assume that A is Regular

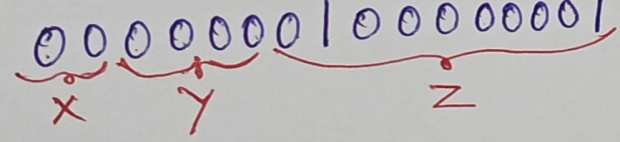
→ Pumping length say p

→ $S = 0^p 1\ 0^p 1$



p=7

0000000100000001



$xy^iz \Rightarrow xy^2z$

00000000000100000001

$\notin A$

$|y| > 0$ ✓

$|xy| \leq p = 7$ ✓

Therefore: A is not Regular

Grammars and Context Free Language

Mathematical model of Grammar

⇒ Noam Chomsky gave a mathematical model of Grammar which is effective for writing Computer languages.

⇒ The four types of Grammar according to Noam Chomsky are:

Grammar Type	Grammar Accepted	Language Accepted	Automaton
Type 0	Unrestricted Grammar	Recursively Enumerable Language	Turing Machine
Type 1	Context Sensitive Grammar	Context Sensitive Language	Linear Bounded Automaton
Type 2	Context Free Grammar	Context Free Language	Pushdown Automata
Type 3	Regular Grammar	Regular Language	Finite State Automaton

Grammar

⇒ A Grammar 'G' can be formally described using 4 tuples as $G_1 = (V, T, S, P)$ where,

V = Set of variables or Non-Terminal Symbols

T = Set of Terminal Symbols

S = Start Symbol

P = Production rules for Terminal and Non-Terminals

A production rule has the form $\alpha \to \beta$ where α and β are strings on $V \cup T$ and atleast one symbol of α belongs to V.

Example: $G_1 = (\{S, A, B\}, \{a, b\}, S, \{S \to AB, A \to a, B \to b\})$

$V = \{S, A, B\}$

$T = \{a, b\}$

$S = S$

$P = S \to AB, A \to a, B \to b$

Eg. $S \to AB$
$\to aB$
$\to ab$

Derivations from a Grammar

The set of all strings that can be derived from a Grammar is said to be the the Language generated from that Grammar.

Example 1: Consider the Grammar

$G_1 = (\{S, A\}, \{a, b\}, S, \{S \to aAb, aA \to aaAb, A \to \epsilon\})$

$S \to aAb$
$\to aaAbb$
$\to aaaAbbb$
$\to aaabbb$

$L(G_1) = \{a^n b^n \mid n > 0\}$

Example 2: $G_2 = (\{S, A, B\}, \{a, b\}, S, \{S \to AB, A \to aA \mid a, B \to bB \mid b\})$

(1) $S \to AB$
$\to aB$
$\to ab$

(2) $S \to AB$
$\to aAB$
$\to aAbB$
$\to aabb$

(3) $S \to AB$
$\to aAB$
$\to aAb$
$\to aab$

(4) $S \to AB$
$\to abB$
$\to abb$

$L(G_2) = \{ab, a^2b^2, a^3b, ab^2, \ldots\}$
$= \{a^m b^n \mid m \geq 0 \text{ and } n \geq 0\}$

Context Free Language

⇒ In formal language theory, a Context Free Language is a language generated by some Context Free Grammar.

⇒ The set of all CFL is identical to the set of languages accepted by Pushdown Automata.

Context Free Grammar is defined by 4 tuples as $G = \{V, \Sigma, S, P\}$ where

V = Set of variables or Non-Terminal symbols

Σ = Set of Terminal symbols

S = Start Symbol

P = Production Rule

Context Free Grammar has Production Rule of the form

$$A \rightarrow \alpha$$

where, $\alpha = \{V \cup \Sigma\}^*$ and $A \in V$

CFL – Example

⇒ For generating a language that generates equal number of a's and b's in the form $a^n b^n$, The CFG will be defined as

$$G_1 = \{(S, A), (a, b), (S \to aAb,\ A \to aAb \mid \epsilon)\}$$

$S \to aAb$

$\to aaAbb$ (by $A \to aAb$)

$\to aaaAbbb$ (by $A \to aAb$)

$\to aaabbb$ (by $A \to \epsilon$)

$\Rightarrow a^3 b^3 \Rightarrow a^n b^n$

Method to find whether a string belongs to a Grammar or not.

① Start with the Start Symbol and choose the closest Production that matches to the given string.

② Replace the variables with its most appropriate Production. Repeat the process until the string is generated or until no other productions are left.

Example: Verify whether the Grammar
$S \to 0B|1A$, $A \to 0|0S|1AA|\wedge$, $B \to 1|1S|0BB$
generates the string 00110101

$S \to 0\underline{B}$ (S → 0B)
$\to 00BB$ (B → 0BB)
$\to 001B$ (B → 1)
$\to 0011S$ (B → 1S)
$\to 00110B$ (S → 0B)
$\to 001101\underline{S}$ (B → 1S)
$\to 00110101$ (B → 1)

Example: verify whether the Grammar
$S \to aAb$, $A \to aAb|\wedge$ generates the string aabbb.

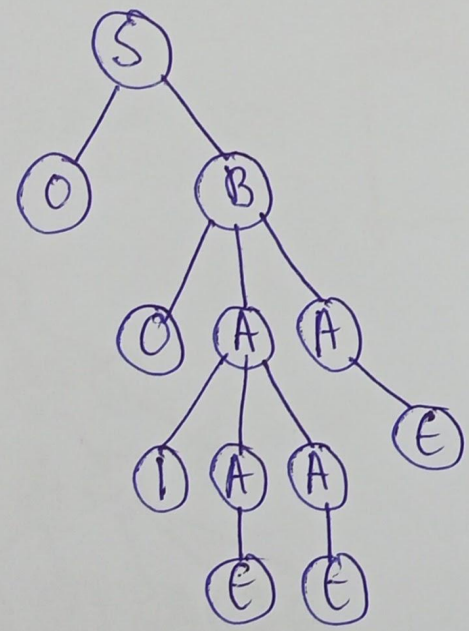
Derivation Tree, Ambiguous Grammar and PDA

Derivation Tree

⇒ A Derivation Tree or Parse Tree is an ordered rooted tree that graphically represents the semantic information of strings derived from a CFG.

Example: For the Grammar $G = \{V, T, P, S\}$ where

$S \to 0B,\ A \to 1AA \mid \epsilon,\ B \to 0AA$



Rootvertex: Must be labelled by the Start Symbol

vertex: Labelled by Non-Terminal Symbols

Leaves: Labelled by Terminal Symbols or ϵ

Derivation Tree

Left Derivation Tree

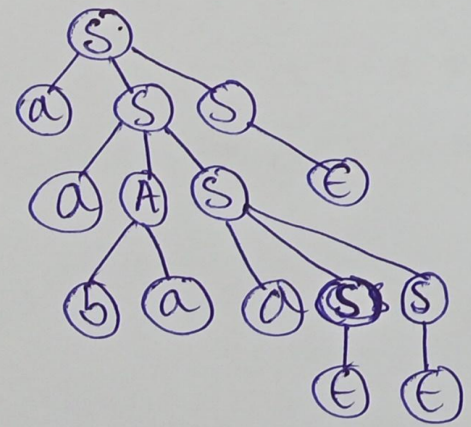
A Left Derivation Tree is obtained by applying production to the leftmost variable in each step.

Right Derivation Tree

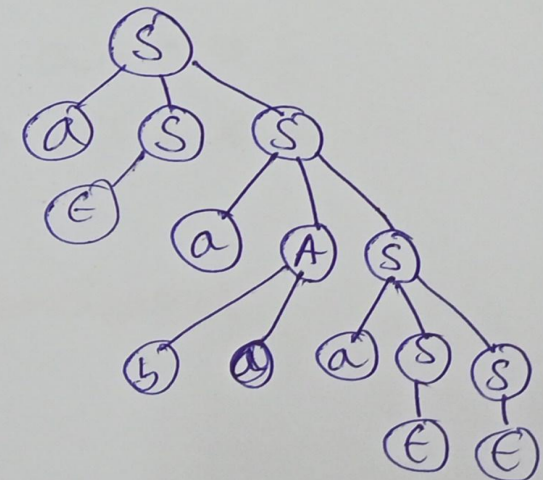
A Right Derivation Tree is obtained by applying Production to the rightmost variable in each step.

Eg. For generating the string aabaa from the Grammar

$S \rightarrow aAS \mid aSS \mid \epsilon$, $A \rightarrow SbA \mid ba$



aabaa



aabaa

Ambiguous Grammar

⇒ A Grammar is said to be Ambiguous if there exists two more derivation tree for a string w (that means two or more left derivation trees)

Example: $G = (\{S\}, \{a+b, +, *\}, P, S\}$ where P consists of $S \to S+S \mid S*S \mid a \mid b$. The String $a+a*b$ can be generated as:

$S \to \underline{S}+S$
$\to a+\underline{S}$
$\to a+\underline{S}*S$
$\to a+a*\underline{S}$
$\to a+a*b$

$S \to \underline{S}*S$
$\to \underline{S}+S*S$
$\to a+\underline{S}*S$
$\to a+a*\underline{S}$
$\to a+a*b$

Thus, this Grammar is Ambiguous

Pushdown Automata (Introduction)

⇒ A pushdown Automata (PDA) is a way to implement a CFG in a similar way we design FA for Regular Grammar

→ It is more powerful than FSM

→ FSM has a very limited memory but PDA has more memory

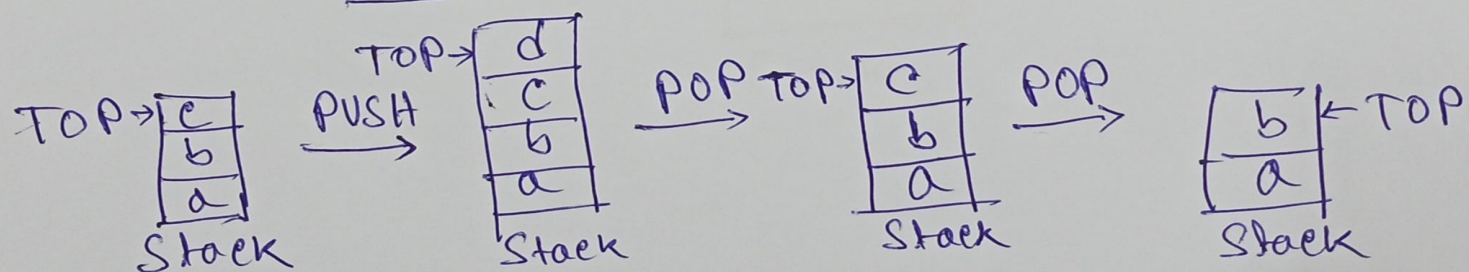
→ PDA = Finite State machine + A Stack

⇒ A Stack is a way we arrange elements one on top of another.

⇒ A stack does two basic operations:

PUSH: A new element is added at the top of the stack.

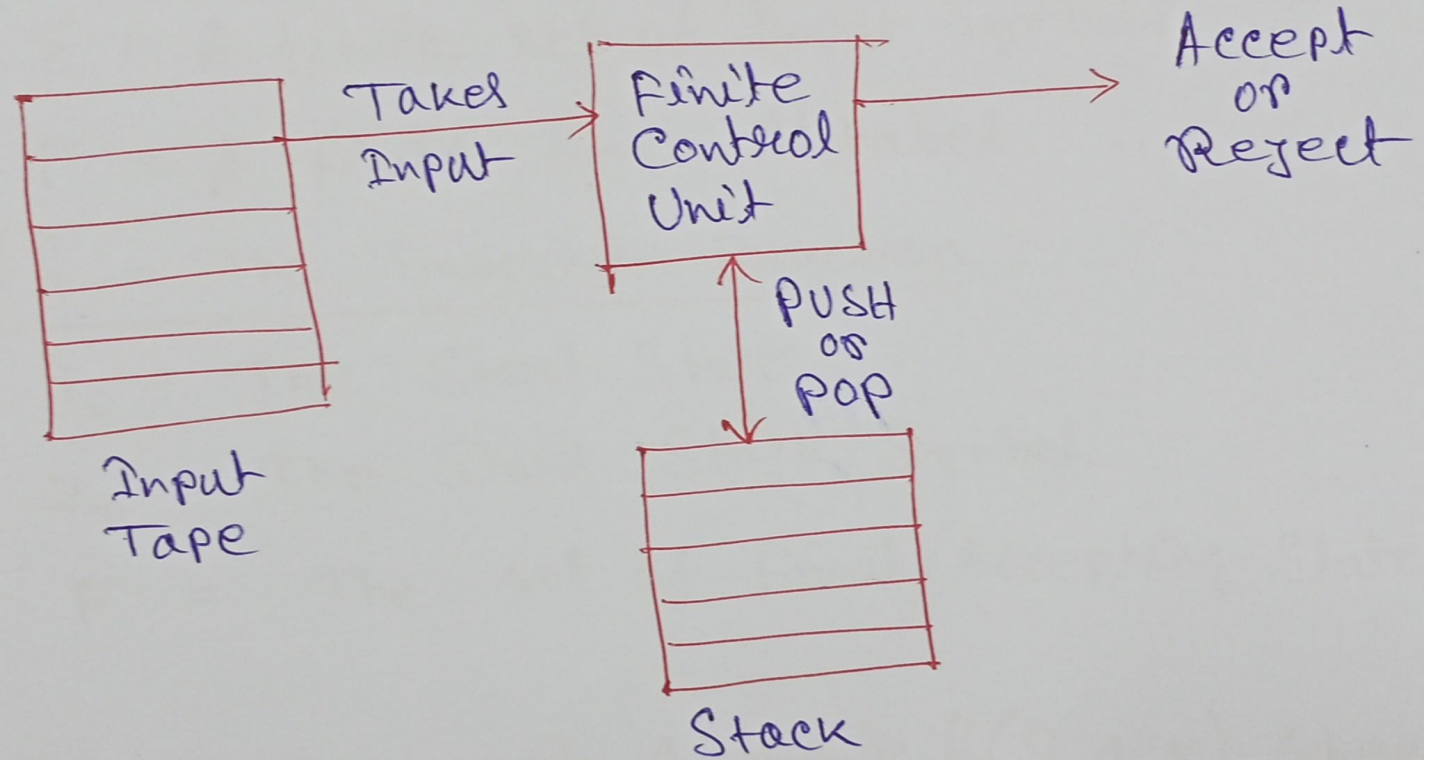
POP: The top element of the stack is read or removed



Pushdown Automata (Introduction)

⇒ A Pushdown Automata has 3 Components

1) An input tape
2) A Finite Control Unit
3) A Stack with infinite size



Pushdown Automata (Formal Definition)

⇒ A Pushdown Automata is formally defined by 7 Tuples as shown below:

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

Where

Q = A finite set of States

Σ = A finite set of Input Symbols

Γ = A finite Stack Alphabet

δ = The Transition Function

q_0 = The Start State

Z_0 = The Start Stack Symbol

F = The set of Final/Accepting States

⇒ δ takes as argument a triple $\delta(q, a, X)$ where:

i) q is a State in Q

ii) a is either an Input Symbol in Σ or $a = \epsilon$

iii) X is a Stack Symbol, that is a member of Γ

⇒ The output of δ is finite set of pairs (P, Y) where

→ P is a new State

→ Y is a string of stack Symbols that replaces X at the top of the Stack.

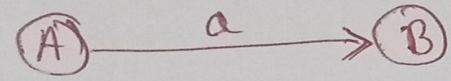
Eg. If $Y = \epsilon$ then the stack is popped.

If $Y = X$ then the stack is Unchanged.

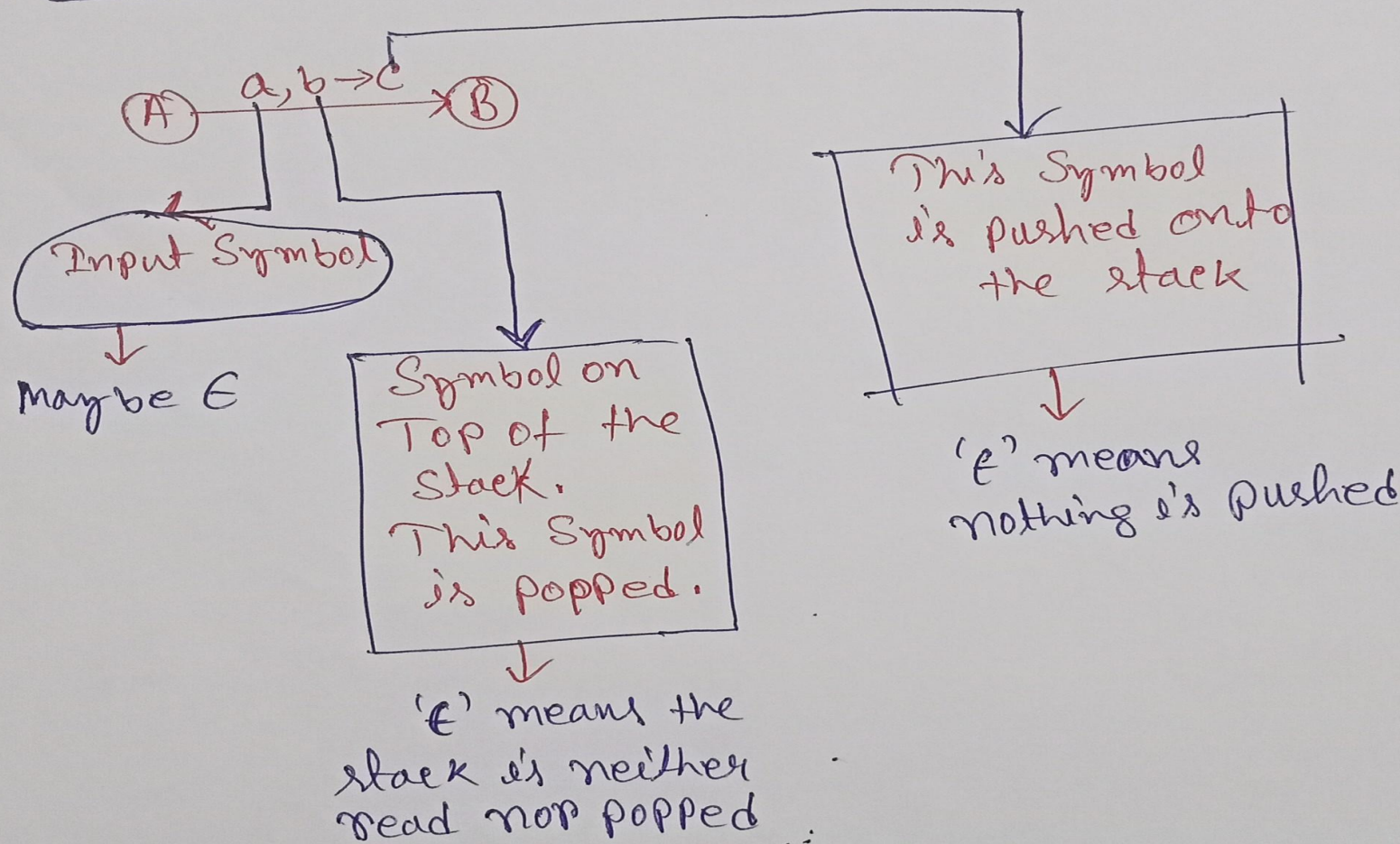
If $Y = YZ$ then X is replaced by Z and Y is pushed onto the stack.

Pushdown Automata (Graphical Notation)

Finite State Machine

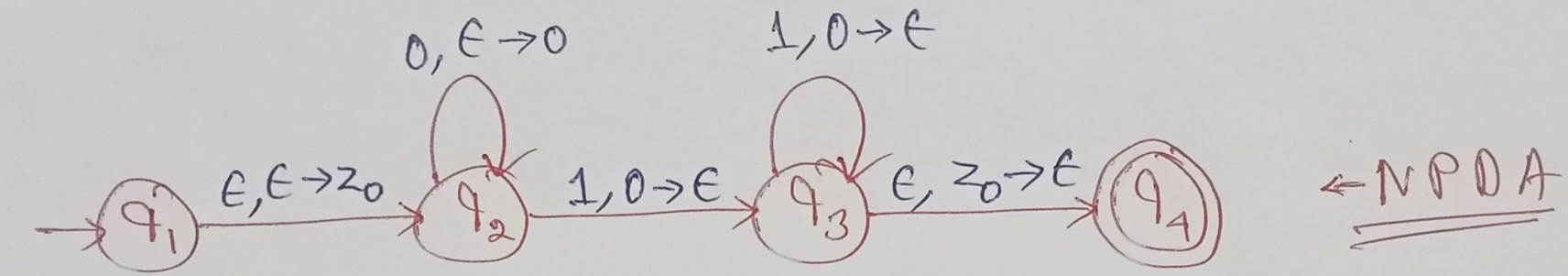


Pushdown Automata



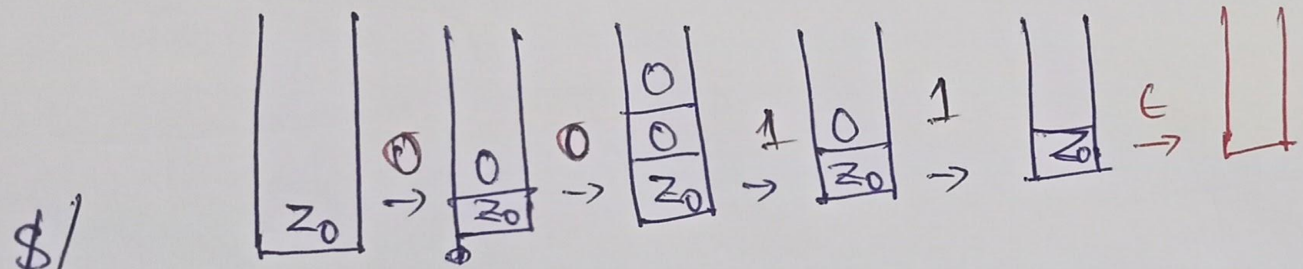
Pushdown Automata – Example

Example: Construct a PDA that accepts $L = \{0^n 1^n \mid n \geq 0\}$



← NPDA

0011 - ✓



Accepted cases → ① Reach the final state
→ ② Stack is empty.

PUSHdown Automata - Example (Even palindrome)

Construct a PDA that accepts Even Palindromes of the form

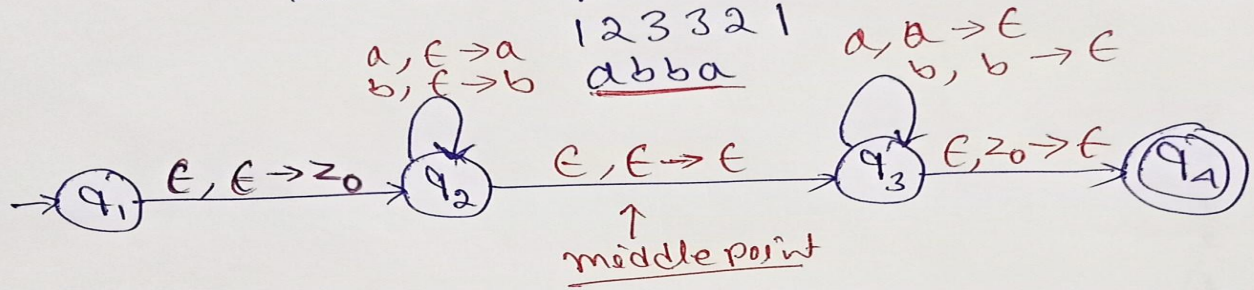
$$L = \{ ww^R \mid w = (a+b)^+ \}$$

PALINDROMES: A word or sequence that reads the same backwards as forwards

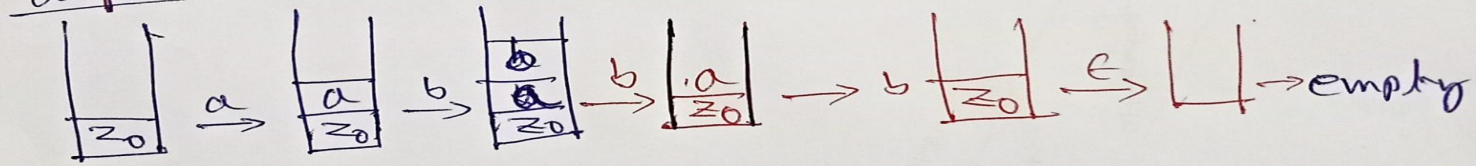
Example : NOON

12332 1

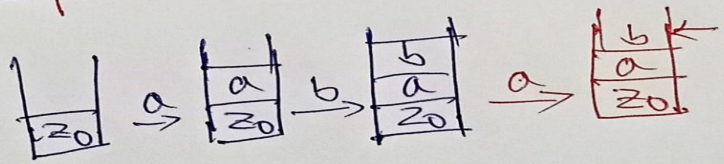
abba



abba ✓



abaab ✗

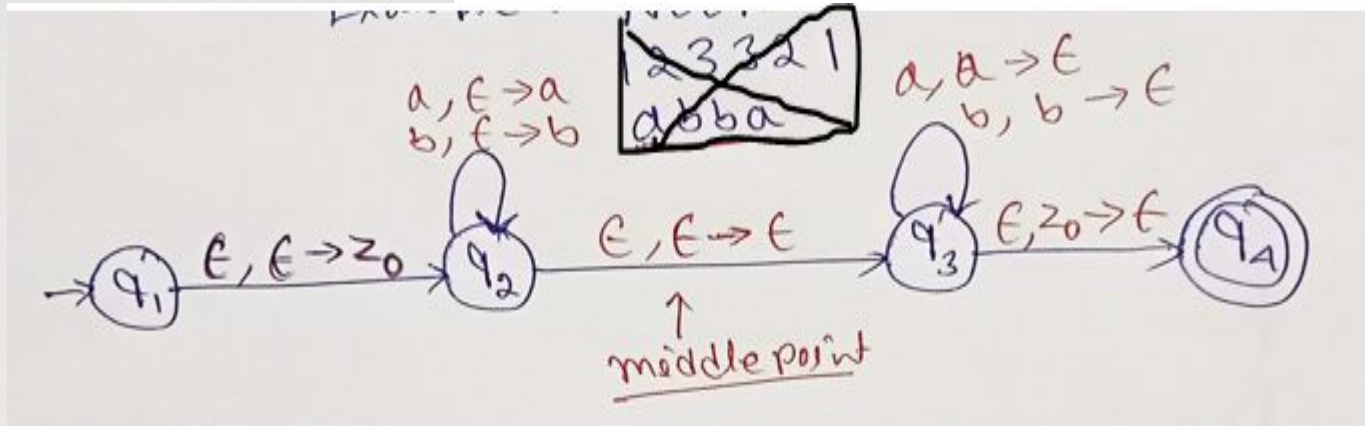


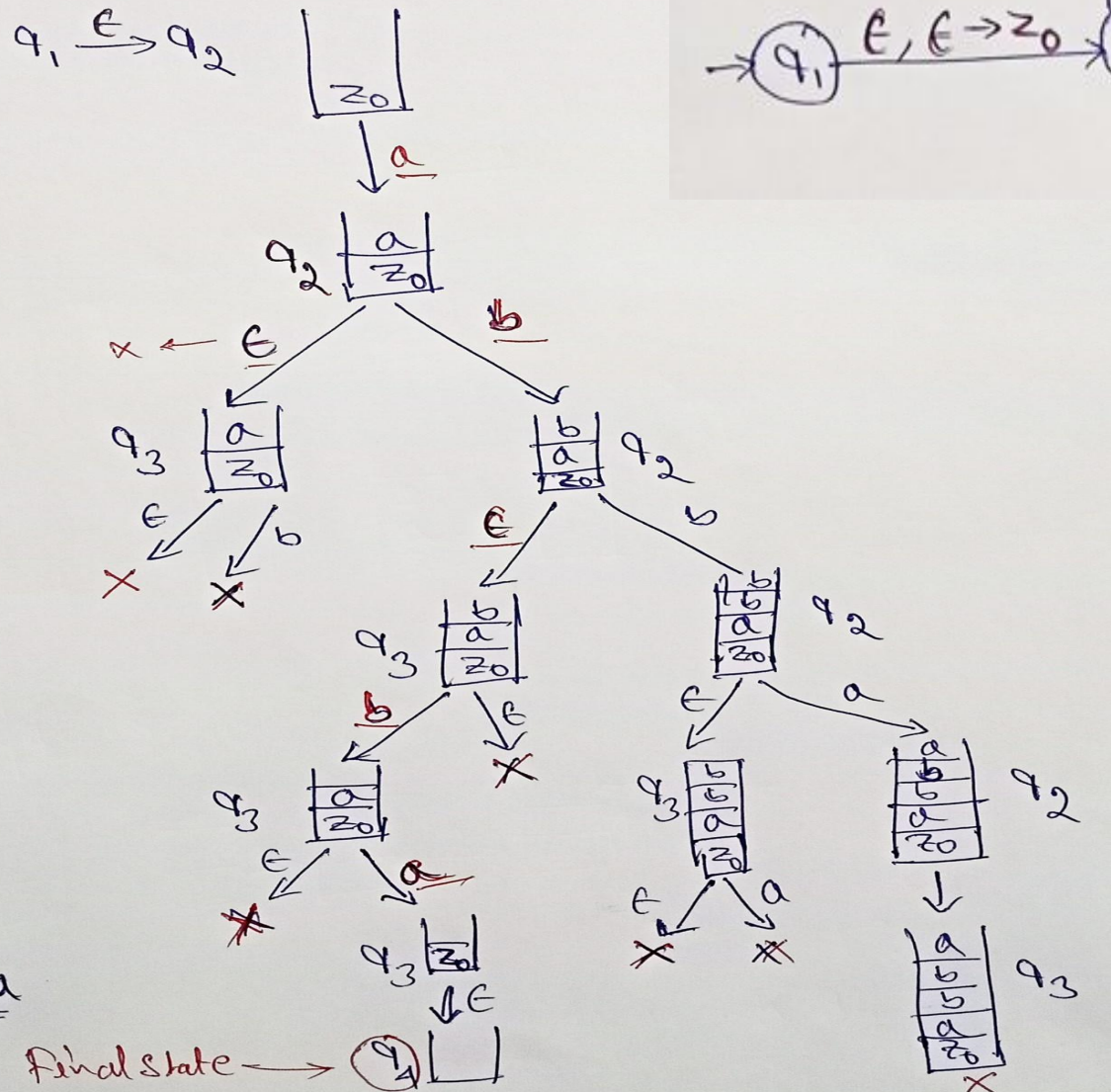
doubt?

⇒ How do we know the the mid point of the string?.

PDA - Example (Even Palindrome)

Example: $\epsilon\ a\ \epsilon\ b\ \epsilon\ b\ \epsilon\ a\ \epsilon$





Pushdown Automata - Example

Example: $\epsilon a \epsilon b \epsilon a \epsilon b \epsilon$

Check How the PDA is working?

Simplification of Context-free Grammar

Simplification of Context Free Grammar

Reduction of CFG

In CFG, sometimes all the production rules and symbols are not needed for the derivation of strings. Besides this, there may also be some NULL productions and Unit productions. Elimination of these productions and symbols is called Simplification of CFG.

Simplification consists of the following steps:

1. Reduction of CFG.
2. Removal of Unit productions.
3. Removal of NULL production.

Reduction of CFG

⇒CFGs are reduced in two phases:

Phase 1: Derivation of an equivalent grammar G', from the CFG G, such that each variable derives some terminal string.

Derivation Procedure:

Step 1: Include all symbols W_1, that derives some terminal and initialize $i=1$.

Step 2: Include symbols W_{i+1}, that derives W_i.

Step 3: Increment i and repeat Step 2, until $W_{i+1} = W_i$.

Step 4: Include all production rules that have W_i in it.

Phase 2: Derivation of an equivalent grammar G'', from the CFG, G', such that each symbol appears in a sentential form.

Derivation Procedure:

Step 1: Include the Start Symbol in Y_1 and initialize $i=1$.

Step 2: Include ~~the~~ all symbols Y_{i+1}, that can be derived from Y_i and include all production rules that have been applied.

Step 3: Increment i and repeat Step 2, until $Y_{i+1} = Y_i$.

– Example –

=> Find a reduced grammar equivalent ~~the~~ to the grammar G, having production rules

P: $S \to AC \mid B$, $A \to a$, $C \to c/BC$, $E \to aA/e$

Phase 1

$T = \{a, c, e\}$

$W_1 = \{A, C, E\}$

$W_2 = \{A, C, E, S\}$

$W_3 = \{A, C, E, S\}$

$G' = \{(A, C, E, S), \{a, c, e\}, P, (S)\}$

P: $S \to AC$, $A \to a$, $C \to c$, $E \to aA/e$

Phase 2:

$Y_1 = \{S\}$

$Y_2 = \{S, A, c\}$

$Y_3 = \{S, A, C, a, c\}$

$Y_4 = \{S, A, C, a, c\}$

$G'' = \{(A, C, S), \{a, c\}, P, \{S\}\}$

P: $S \to AC$, $A \to c$, $C \to e$.

Simplification of CFG

(2) Removal of unit productions:

Any production Rule of the form $A \rightarrow B$ where $A, B \in$ Non-Terminals is called Unit Production.

Procedure for removal

Step1: To remove $A \rightarrow B$, add production $A \rightarrow x$ to the grammar rule whenever $B \rightarrow x$ occurs in the grammar. [$x \in$ Terminal, x Can be NULL]

Step2: Delete $A \rightarrow B$ from the grammar.

Step3: Repeat from step 1 until all unit productions are removed.

– Example –

Remove Unit productions from the Grammar whose production rule is given by

P : $S \to XY,\ X \to a,\ Y \to Z \mid b,\ Z \to M,\ M \to N,\ N \to a$

Ans:

$Y \to Z,\ Z \to M,\ M \to N$

i) Because $N \to a$, we add $M \to a$

P : $S \to XY,\ X \to a,\ Y \to Z \mid b,\ Z \to M,\ M \to a,\ N \to a$

ii) Because $M \to a$, we add $Z \to a$

P : $S \to XY,\ X \to a,\ Y \to Z \mid b,\ Z \to a,\ M \to a,\ N \to a$

iii) Because $Z \to a$, we add $Y \to a$

P : $S \to XY,\ X \to a,\ Y \to a \mid b,\ Z \to a,\ M \to a,\ N \to a$

$\Rightarrow$ Unreachable Symbols : Z, M, N.

↓

From Start symbol there is no way to reach those Symbols. [Remove it]

P : $S \to XY,\ X \to a,\ Y \to a \mid b$.

Simplification of CFGs

(3) Removal of NULL productions.

In a CFG, a Non-Terminal Symbol 'A' is a nullable Variable if there is a production $A \to \epsilon$ or there is a derivation that starts at 'A' and leads to ϵ.

Procedure for Removal:

Step1: To remove $A \to \epsilon$, look for all productions whose right side contains A.

Step2: Replace each occurenceee of 'A' in each of these productions with ϵ.

Step3: Add the resultant production to the Grammar.

- Example -

Remove NULL productions from the following Grammar

P: $S \to ABAC,\ A \to aA/\epsilon,\ B \to bB/\epsilon,\ C \to c.$

NULL production: $A \to \epsilon$, $B \to \epsilon$

1) To eliminate $A \to \epsilon$

→ $S \to ABAC$

$S \to ABC \mid BAC \mid BC$

→ $A \to aA$

$A \to a$

P: $S \to ABAC \mid ABC \mid BAC \mid BC$

$A \to aA/a$, $B \to bB/\epsilon$, $C \to c$

2) To elimina $B \to \epsilon$

→ $B \to bB$

$B \to b$

→ $S \to AAC \mid AC \mid C$

✓ P: $S \to ABAC \mid ABC \mid BAC \mid BC \mid AAC \mid AC \mid C$

$A \to aA \mid a$

$B \to bB \mid b$

$C \to c$

Chomsky Normal Form, Conversion (CFG-CNF)

Chomsky Normal Form (CNF)

⇒ In Chomsky Normal Form (CNF), we have a restriction on the length of Right Hand Side (RHS) where elements in RHS should either be two variables or a terminal.

⇒ A CFG is in CNF if the productions are in the following forms:

$A \to a$
$A \to BC$

where A, B and C $\in V$, $a \in T$

CNF

⇒ Steps to Convert a given CFG to CNF Form:

Step1: If the Start Symbol S occurs on some right Side of production rules then Create a new Start Symbol S' and add a new production S' → S, otherwise add the Start Symbol S in production rules.

Step2: Remove NULL production. ✓

Step3: Remove UNIT production. ✓

Step4: Replace each production $A \to B_1 \cdots B_n$ where $n > 2$, with $A \to B_1 C$ where $C \to B_2 \cdots B_n$.
Repeat this step for all productions having two or more Symbols on the right side.

Step5: If any production is in the form of $A \to aB$, where $a \in T$ and $A, B \in V$ then the production is replaced by $A \to XB$ and $X \to a$.

Repeat this step for every production which is in the form of $A \to aB$.

Conversion of CFG to CNF

⇒ Convert the following CFG to CNF:

P: $S \to ASA \mid aB$, $A \to B \mid S$, $B \to b \mid \epsilon$

Step 1: P: $S' \to S$, $S \to ASA \mid aB$, $A \to B \mid S$, $B \to b \mid \epsilon$

Step 2: Remove the null productions: $B \to \epsilon$ and $A \to \epsilon$

After Removing $B \to \epsilon$:

P: $S' \to S$, $S \to ASA \mid aB \mid a$, $A \to B \mid S \mid \epsilon$, $B \to b$

After Removing $A \to \epsilon$:

P: $S' \to S$, $S \to ASA \mid aB \mid a \mid AS \mid SA \mid S$,
$A \to B \mid S$, $B \to b$

Step 3: Remove the Unit Productions: $S \to S$, $S' \to S$, $A \to B$ and $A \to S$

After Removing $S \to S$:

P: $S' \to S$, $S \to ASA \mid aB \mid a \mid AS \mid SA$, $A \to B \mid S$,
$B \to b$

After Removing $S' \to S$: P: $S' \to ASA \mid aB \mid a \mid AS \mid SA$
$S \to ASA \mid aB \mid a \mid AS \mid SA$
$A \to B \mid S$
$B \to b$

After Removing $A \to B$: P: $S' \to ASA|aB|a|AS|SA$,
$S \to ASA|aB|a|AS|SA$
$A \to b|S$, $B \to b$.

After Removing $A \to S$: P: $S' \to ASA|aB|a|AS|SA$,
$S \to ASA|aB|a|AS|SA$,
$A \to b|ASA|aB|a|AS|SA$,
$B \to b$

Step 4: Now find out the productions that has more than two variables in RHS.

$S' \to ASA$, $S \to ASA$, $A \to ASA$

After Removing: P: $S' \to AX|aB|a|AS|SA$,
$S \to AX|aB|a|AS|SA$,
$A \to b|AX|aB|a|AS|SA$.
$B \to b$
$X \to SA$

: $S' \to aB,\ S \to aB,\ A \to aB$

P: $S' \to AX \mid YB \mid a \mid AS \mid SA,$

$S \to AX \mid YB \mid a \mid AS \mid SA,$

$A \to b \mid AX \mid YB \mid a \mid AS \mid SA;$

$B \to b,$

$X \to SA,$

$Y \to a$

CNF