

## Finite Automata Formal Languages: A Padma Reddy

**Definition:** A DFA is 5-tuple or quintuple  $M = (Q, \Sigma, \delta, q_0, A)$  where

$Q$  is non-empty, finite set of states.

$\Sigma$  is non-empty, finite set of input alphabets.

$\delta$  is transition function, which is a mapping from  $Q \times \Sigma$  to  $Q$ .

$q_0 \in Q$  is the start state.

$A \subseteq Q$  is set of accepting or final states.

Note: For each input symbol  $a$ , from a given state there is exactly one transition (there can be no transitions from a state also) and we are sure (or can determine) to which state the machine enters. So, the machine is called **Deterministic** machine. Since it has finite number of states the machine is called Deterministic finite machine or Deterministic Finite Automaton or Finite State Machine (FSM).

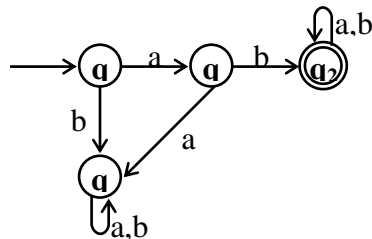
The language accepted by DFA is

$$L(M) = \{ w \mid w \in \Sigma^* \text{ and } \delta^*(q_0, w) \in A \}$$

The non-acceptance of the string  $w$  by an FA or DFA can be defined in formal notation as:

$$L(M) = \{ w \mid w \in \Sigma^* \text{ and } \delta^*(q_0, w) \notin A \}$$

**Obtain a DFA to accept strings of a's and b's starting with the string ab**



**Fig.2.7 Transition diagram to accept string  $ab(a+b)^*$**

So, the DFA which accepts strings of a's and b's starting with the string  $ab$  is given by  $M = (Q, \Sigma, \delta, q_0, A)$  where

$$Q = \{q_0, q_1, q_2, q_3\}$$

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

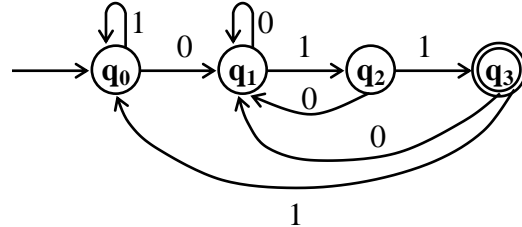
$q_0$  is the start state

$$A = \{q_2\}.$$

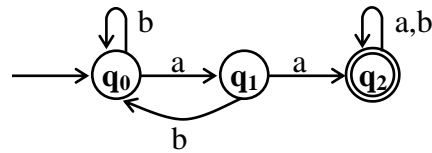
$\delta$  is shown the transition table 2.4.

	$\delta$	$\leftarrow \Sigma \rightarrow$	
		a	b
$\uparrow$ States $\downarrow$	$\rightarrow q_0$	$q_1$	$q_3$
	$q_1$	$q_3$	$q_2$
	$\odot q_2$	$q_2$	$q_2$
	$q_3$	$q_3$	$q_3$

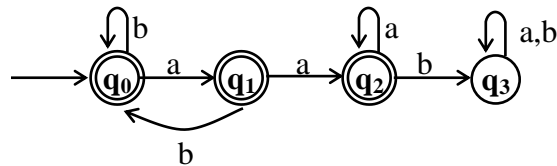
Draw a DFA to accept string of 0's and 1's ending with the string 011.



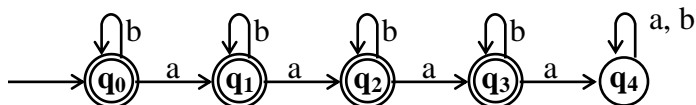
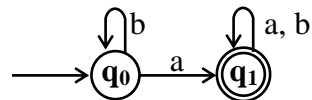
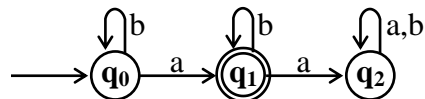
Obtain a DFA to accept strings of a's and b's having a sub string aa



Obtain a DFA to accept strings of a's and b's except those containing the substring aab.



Obtain DFAs to accept strings of a's and b's having exactly one a,



Obtain a DFA to accept strings of a's and b's having even number of a's and b's

The machine to accept even number of a's and b's is shown in fig.2.22.

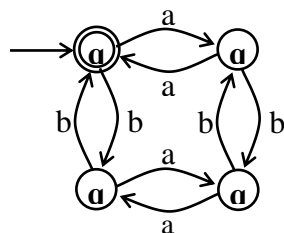
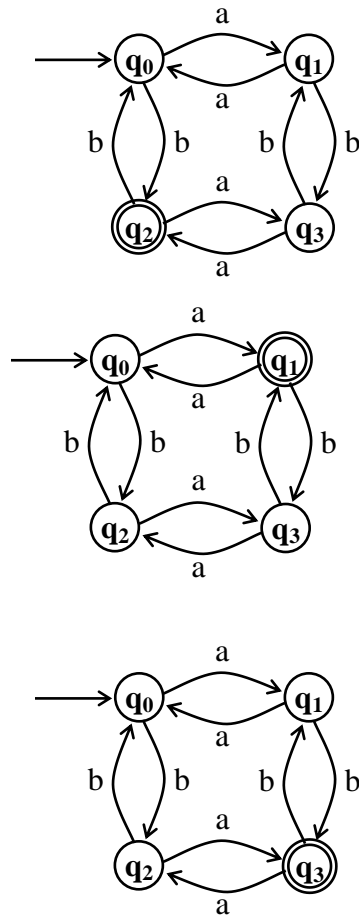


Fig.2.22 DFA to accept even no. of a's and b's



## Regular language

**Definition:** Let  $M = (Q, \Sigma, \delta, q_0, A)$  be a DFA. The language  $L$  is regular if there exists a machine  $M$  such that  $L = L(M)$ .

## \* Applications of Finite Automata \*

### String matching/processing

### Compiler Construction

The various compilers such as C/C++, Pascal, Fortran or any other compiler is designed using the finite automata. The DFAs are extensively used in the building the various phases of compiler such as

- Lexical analysis (To identify the tokens, identifiers, to strip of the comments etc.)
- Syntax analysis (To check the syntax of each statement or control statement used in the program)
- Code optimization (To remove the un wanted code)
- Code generation (To generate the machine code)

## Other applications

The concept of finite automata is used in wide applications. It is not possible to list all the applications as there are infinite number of applications. This section lists some applications:

1. Large natural vocabularies can be described using finite automaton which includes the applications such as spelling checkers and advisers, multi-language dictionaries, to indent the documents, in calculators to evaluate complex expressions based on the priority of an operator etc. to name a few. Any editor that we use uses finite automaton for implementation.
2. Finite automaton is very useful in recognizing difficult problems i.e., sometimes it is very essential to solve an un-decidable problem. Even though there is no general solution exists for the specified problem, using theory of computation, we can find the approximate solutions.
3. Finite automaton is very useful in hardware design such as circuit verification, in design of the hardware board (mother board or any other hardware unit), automatic traffic signals, radio controlled toys, elevators, automatic sensors, remote sensing or controller etc.
4. In game theory and games wherein we use some control characters to fight against a monster, economics, computer graphics, linguistics etc., finite automaton plays a very important role.

## Non deterministic finite automata(NFA)

**Definition:** An NFA is a 5-tuple or quintuple  $M = (Q, \Sigma, \delta, q_0, A)$  where

$Q$  is non empty, finite set of states.

$\Sigma$  is non empty, finite set of input alphabets.

$\delta$  is transition function which is a mapping from

$Q \times \{\Sigma \cup \epsilon\}$  to subsets of  $2^Q$ . This function shows the change of state from one state to a set of states based on the input symbol.

$q_0 \in Q$  is the start state.

$A \subseteq Q$  is set of final states.

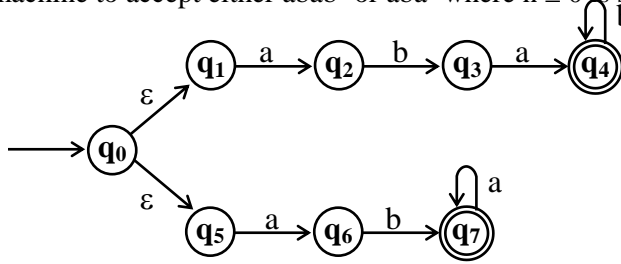
## Acceptance of language

**Definition:** Let  $M = (Q, \Sigma, \delta, q_0, A)$  be a DFA where  $Q$  is set of finite states,  $\Sigma$  is set of input alphabets (from which a string can be formed),  $\delta$  is transition function from  $Q \times \{\Sigma \cup \epsilon\}$  to  $2^Q$ ,  $q_0$  is the start state and  $A$  is the final or accepting state. The string (also called language)  $w$  accepted by an NFA can be defined in formal notation as:

$$L(M) = \{ w \mid w \in \Sigma^* \text{ and } \delta^*(q_0, w) = Q \text{ with atleast one Component of } Q \text{ in } A \}$$

**Obtain an NFA to accept the following language  $L = \{w \mid w \in abab^n \text{ or } aba^n \text{ where } n \geq 0\}$**

The machine to accept either  $abab^n$  or  $aba^n$  where  $n \geq 0$  is shown below:



## 2.1 Conversion from NFA to DFA

Let  $M_N = (Q_N, \Sigma_N, \delta_N, q_0, A_N)$  be an NFA and accepts the language  $L(M_N)$ . There should be an equivalent DFA  $M_D = (Q_D, \Sigma_D, \delta_D, q_0, A_D)$  such that  $L(M_D) = L(M_N)$ . The procedure to convert an NFA to its equivalent DFA is shown below:

Step1:

The start state of NFA  $M_N$  is the start state of DFA  $M_D$ . So, add  $q_0$  (which is the start state of NFA) to  $Q_D$  and find the transitions from this state. The way to obtain different transitions is shown in step2.

Step2:

For each state  $[q_i, q_j, \dots, q_k]$  in  $Q_D$ , the transitions for each input symbol in  $\Sigma$  can be obtained as shown below:

1.  $\delta_D([q_i, q_j, \dots, q_k], a) = \delta_N(q_i, a) \cup \delta_N(q_j, a) \cup \dots \cup \delta_N(q_k, a)$   
 $= [q_l, q_m, \dots, q_n]$  say.
2. Add the state  $[q_l, q_m, \dots, q_n]$  to  $Q_D$ , if it is not already in  $Q_D$ .
3. Add the transition from  $[q_i, q_j, \dots, q_k]$  to  $[q_l, q_m, \dots, q_n]$  on the input symbol  $a$  iff the state  $[q_l, q_m, \dots, q_n]$  is added to  $Q_D$  in the previous step.

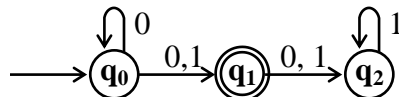
Step3:

The state  $[q_a, q_b, \dots, q_c] \in Q_D$  is the final state, if at least one of the state in  $q_a, q_b, \dots, q_c \in A_N$  i.e., at least one of the component in  $[q_a, q_b, \dots, q_c]$  should be the final state of NFA.

Step4:

If epsilon ( $\epsilon$ ) is accepted by NFA, then start state  $q_0$  of DFA is made the final state.

**Convert the following NFA into an equivalent DFA.**



Step1:  $q_0$  is the start of DFA (see step1 in the conversion procedure).

$$\text{So, } Q_D = \{[q_0]\} \quad (2.7)$$

Step2: Find the new states from each state in  $Q_D$  and obtain the corresponding transitions.

**Consider the state  $[q_0]$ :**

$$\begin{aligned} \text{When } a = 0 \\ \delta_D([q_0], 0) &= \delta_N([q_0], 0) \\ &= [q_0, q_1] \\ (2.8) \end{aligned}$$

$$\begin{aligned} \text{When } a = 1 \\ \delta_D([q_0], 1) &= \delta_N([q_0], 1) \\ &= [q_1] \\ (2.9) \end{aligned}$$

Since the states obtained in (2.8) and (2.9) are not in  $Q_D(2.7)$ , add these two states to  $Q_D$  so that

$$Q_D = \{[q_0], [q_0, q_1], [q_1]\} \quad (2.10)$$

The corresponding transitions on  $a = 0$  and  $a = 1$  are shown below.

$\longleftrightarrow \Sigma \longrightarrow$			
	$\delta$	0	1
$\uparrow$	$[q_0]$	$[q_0, q_1]$	$[q_1]$
$\downarrow$	$[q_0, q_1]$		
	$[q_1]$		

**Consider the state  $[q_0, q_1]$ :**

$$\begin{aligned} \text{When } a = 0 \\ \delta_D([q_0, q_1], 0) &= \delta_N([q_0, q_1], 0) \\ &= \delta_N(q_0, 0) \cup \delta_N(q_1, 0) \\ &= \{q_0, q_1\} \cup \{q_2\} \\ &= [q_0, q_1, q_2] \\ (2.11) \end{aligned}$$

$$\begin{aligned} \text{When } a = 1 \\ \delta_D([q_0, q_1], 1) &= \delta_N([q_0, q_1], 1) \\ &= \delta_N(q_0, 1) \cup \delta_N(q_1, 1) \\ &= \{q_1\} \cup \{q_2\} \\ &= [q_1, q_2] \end{aligned}$$

(2.12)

Since the states obtained in (2.11) and (2.12) are not defined in  $Q_D$  (see 2.10), add these two states to  $Q_D$  so that

$$Q_D = \{ [q_0], [q_0, q_1], [q_1], [q_0, q_1, q_2], [q_1, q_2] \} \quad (2.13)$$

and add the transitions on  $a = 0$  and  $a = 1$  as shown below:

		$\longleftrightarrow \Sigma \longrightarrow$	
$\begin{array}{c} \uparrow \\ Q \\ \downarrow \end{array}$	$\delta$	0	1
	$[q_0]$	$[q_0, q_1]$	$[q_1]$
	$[q_0, q_1]$	$[q_0, q_1, q_2]$	$[q_1, q_2]$
	$[q_1]$		
	$[q_0, q_1, q_2]$		
	$[q_1, q_2]$		

**Consider the state  $[q_1]$ :**

When  $a = 0$

$$\begin{aligned} \delta_D([q_1], 0) &= \delta_N([q_1], 0) \\ &= [q_2] \end{aligned} \quad (2.14)$$

When  $a = 1$

$$\begin{aligned} \delta_D([q_1], 1) &= \delta_N([q_1], 1) \\ &= [q_2] \end{aligned} \quad (2.15)$$

Since the states obtained in (2.14) and (2.15) are same and the state  $q_2$  is not in  $Q_D$  (see 2.13), add the state  $q_2$  to  $Q_D$  so that

$$Q_D = \{ [q_0], [q_0, q_1], [q_1], [q_0, q_1, q_2], [q_1, q_2], [q_2] \} \quad (2.16)$$

and add the transitions on  $a = 0$  and  $a = 1$  as shown below:

		$\longleftrightarrow \Sigma \longrightarrow$	
	$\delta$	0	1
	$[q_0]$	$[q_0, q_1]$	$[q_1]$
	$[q_0, q_1]$	$[q_0, q_1, q_2]$	$[q_1, q_2]$
	$[q_1]$	$[q_2]$	$[q_2]$
	$[q_0, q_1, q_2]$		
	$[q_1, q_2]$		

$[q_1, q_2]$		
$[q_2]$		

Consider the state  $[q_0, q_1, q_2]$ :

When  $a = 0$

$$\begin{aligned}
 \delta_D([q_0, q_1, q_2], 0) &= \delta_N([q_0, q_1, q_2], 0) \\
 &= \delta_N(q_0, 0) \cup \delta_N(q_1, 0) \cup \delta_N(q_2, 0) \\
 &= \{q_0, q_1\} \cup \{q_2\} \cup \{\phi\} \\
 &= [q_0, q_1, q_2] \\
 &\quad (2.17)
 \end{aligned}$$

When  $a = 1$

$$\begin{aligned}
 \delta_D([q_0, q_1, q_2], 1) &= \delta_N([q_0, q_1, q_2], 1) \\
 &= \delta_N(q_0, 1) \cup \delta_N(q_1, 1) \cup \delta_N(q_2, 1) \\
 &= \{q_1\} \cup \{q_2\} \cup \{q_2\} \\
 &= [q_1, q_2] \\
 &\quad (2.18)
 \end{aligned}$$

Since the states obtained in (2.17) and (2.18) are not new states (are already in  $Q_D$ , see 2.16), do not add these two states to  $Q_D$ . But, the transitions on  $a = 0$  and  $a = 1$  should be added to the transitional table as shown below:

		$\longleftrightarrow \Sigma \longrightarrow$	
$\uparrow$ Q $\downarrow$	$\delta$	0	1
	$[q_0]$	$[q_0, q_1]$	$[q_1]$
	$[q_0, q_1]$	$[q_0, q_1, q_2]$	$[q_1, q_2]$
	$[q_1]$	$[q_2]$	$[q_2]$
	$[q_0, q_1, q_2]$	$[q_1, q_2]$	
	$[q_1, q_2]$		
	$[q_2]$		

Consider the state  $[q_1, q_2]$ :

When  $a = 0$

$$\begin{aligned}
 \delta_D([q_1, q_2], 0) &= \delta_N([q_1, q_2], 0) \\
 &= \delta_N(q_1, 0) \cup \delta_N(q_2, 0) \\
 &= \{q_2\} \cup \{\phi\} \\
 &= [q_2]
 \end{aligned}$$



(2.19)

When  $a = 1$

$$\begin{aligned}
 \delta_D([q_1, q_2], 1) &= \delta_N([q_1, q_2], 1) \\
 &= \delta_N(q_1, 1) \cup \delta_N(q_2, 1) \\
 &= \{q_2\} \cup \{q_2\} \\
 &= [q_2] \\
 &\quad (2.20)
 \end{aligned}$$

Since the states obtained in (2.19) and (2.20) are not new states (are already in  $Q_D$  see 2.16), do not add these two states to  $Q_D$ . But, the transitions on  $a = 0$  and  $a = 1$  should be added to the transitional table as shown below:

$\longleftrightarrow \Sigma \longrightarrow$

	$\delta$	0	1
$\uparrow$ Q $\downarrow$	$[q_0]$	$[q_0, q_1]$	$[q_1]$
	$[q_0, q_1]$	$[q_0, q_1, q_2]$	$[q_1, q_2]$
	$[q_1]$	$[q_2]$	$[q_2]$
	$[q_0, q_1, q_2]$	$[q_0, q_1, q_2]$	$[q_1, q_2]$
	$[q_1, q_2]$	$[q_2]$	$[q_2]$
	$[q_2]$		

**Consider the state  $[q_2]$ :**

When  $a = 0$

$$\begin{aligned}
 \delta_D([q_2], 0) &= \delta_N([q_2], 0) \\
 &= \{\phi\} \\
 &\quad (2.21)
 \end{aligned}$$

When  $a = 1$

$$\begin{aligned}
 \delta_D([q_2], 1) &= \delta_N([q_2], 1) \\
 &= [q_2] \\
 &\quad (2.22)
 \end{aligned}$$

Since the states obtained in (2.21) and (2.22) are not new states (are already in  $Q_D$ , see 2.16), do not add these two states to  $Q_D$ . But, the transitions on  $a = 0$  and  $a = 1$  should be added to the transitional table. The final transitional table is shown in table 2.14. and final DFA is shown in figure 2.35.

$\delta$	0	1
$[q_0]$	$[q_0, q_1]$	$[q_1]$
	$[q_0, q_1, q_2]$	$[q_1, q_2]$
$[q_0, q_1]$	$[q_2]$	$[q_2]$

$[q_1]$

	$[q_0, q_1, q_2]$	$[q_1, q_2]$
	$[q_2]$	$[q_2]$
$[q_2]$	$\phi$	$[q_2]$

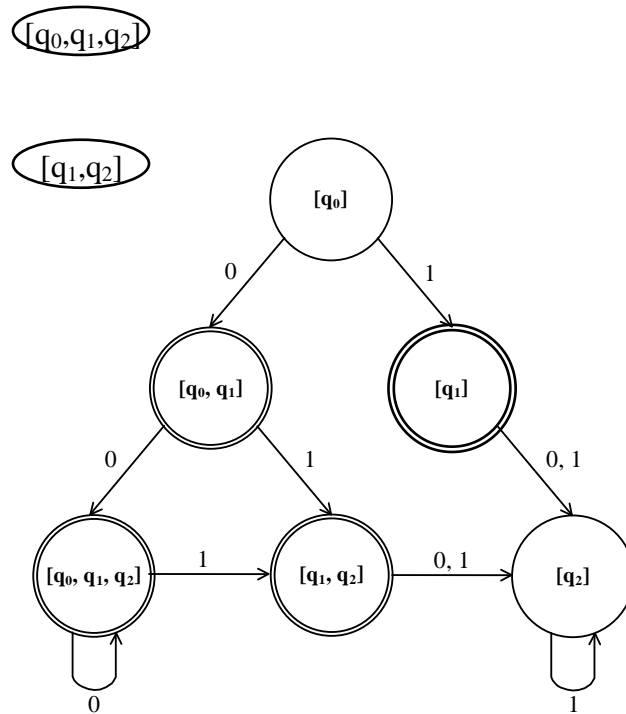
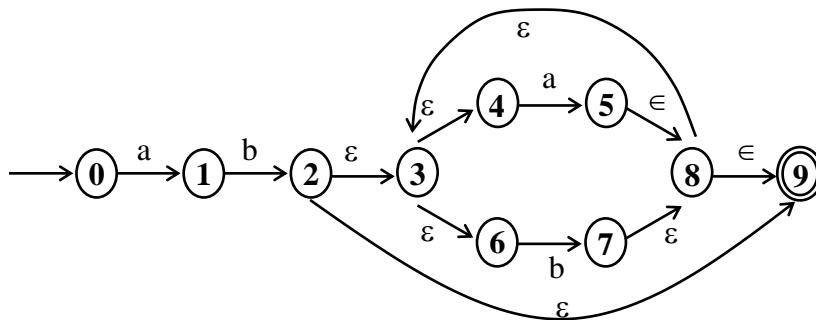


Fig.2.35 The DFA

Convert the following NFA to its equivalent DFA.



Let  $Q_D = \{0\}$  (A)

Consider the state [A]:

When input is  $a$ :

$$\begin{aligned}
\delta(A, a) &= \delta_N(0, a) \\
&= \{1\} \\
&\text{(B)}
\end{aligned}$$

When input is  $b$ :

$$\begin{aligned}
\delta(A, b) &= \delta_N(0, b) \\
&= \{\phi\}
\end{aligned}$$

**Consider the state [B]:**

When input is  $a$ :

$$\begin{aligned}
\delta(B, a) &= \delta_N(1, a) \\
&= \{\phi\}
\end{aligned}$$

When input is  $b$ :

$$\begin{aligned}
\delta(B, b) &= \delta_N(1, b) \\
&= \{2\} \\
&= \{2, 3, 4, 6, 9\} \quad \text{(C)}
\end{aligned}$$

This is because, in state 2, due to  $\epsilon$ -transitions (or without giving any input) there can be transition to states 3, 4, 6, 9 also. So, all these states are reachable from state 2. Therefore,

$$\delta(B, b) = \{2, 3, 4, 6, 9\} = C$$

**Consider the state [C]:**

When input is  $a$ :

$$\begin{aligned}
\delta(C, a) &= \delta_N(\{2, 3, 4, 6, 9\}, a) \\
&= \{5\} \\
&= \{5, 8, 9, 3, 4, 6\} \\
&= \{3, 4, 5, 6, 8, 9\} \quad \text{(ascending order)} \quad \text{(D)}
\end{aligned}$$

This is because, in state 5 due to  $\epsilon$ -transitions, the states reachable are  $\{8, 9, 3, 4, 6\}$ . Therefore,

$$\delta(C, a) = \{3, 4, 5, 6, 8, 9\} = D$$

When input is  $b$ :

$$\begin{aligned}
\delta(C, b) &= \delta_N(\{2, 3, 4, 6, 9\}, b) \\
&= \{7\} \\
&= \{7, 8, 9, 3, 4, 6\} \\
&= \{3, 4, 6, 7, 8, 9\} \text{(ascending order)} \\
&\text{(E)}
\end{aligned}$$

This is because, from state 7 the states that are reachable without any input (i.e.,  $\varepsilon$ -transition) are {8, 9, 3, 4, 6}. Therefore,

$$\delta(C, b) = \{3, 4, 6, 7, 8, 9\} = E$$

**Consider the state [D]:**

When input is  $a$ :

$$\begin{aligned}\delta(D, a) &= \delta_N(\{3,4,5,6,8,9\}, a) \\ &= \{5\} \\ &= \{5, 8, 9, 3, 4, 6\} \\ &= \{3, 4, 5, 6, 8, 9\} \quad (\text{ascending order}) \quad (D)\end{aligned}$$

When input is  $b$ :

$$\begin{aligned}\delta(D, b) &= \delta_N(\{3,4,5,6,8,9\}, b) \\ &= \{7\} \\ &= \{7, 8, 9, 3, 4, 6\} \\ &= \{3, 4, 6, 7, 8, 9\} \quad (\text{ascending order}) \quad (E)\end{aligned}$$

**Consider the state [E]:**

When input is  $a$ :

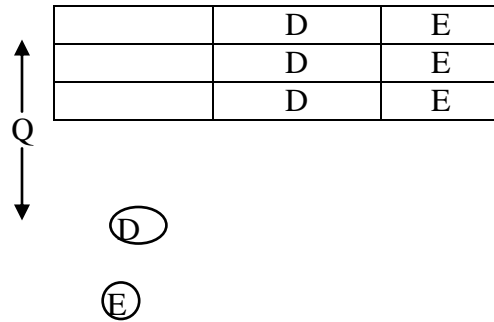
$$\begin{aligned}\delta(E, a) &= \delta_N(\{3,4,6,7,8,9\}, a) \\ &= \{5\} \\ &= \{5, 8, 9, 3, 4, 6\} \\ &= \{3, 4, 5, 6, 8, 9\}(\text{ascending order}) \\ &\quad (D)\end{aligned}$$

When input is  $b$ :

$$\begin{aligned}\delta(E, b) &= \delta_N(\{3,4,6,7,8,9\}, b) \\ &= \{7\} \\ &= \{7, 8, 9, 3, 4, 6\} \\ &= \{3, 4, 6, 7, 8, 9\}(\text{ascending order}) \\ &\quad (E)\end{aligned}$$

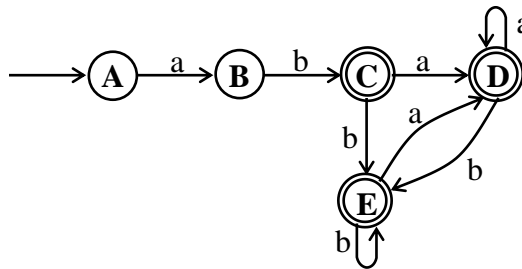
Since there are no new states, we can stop at this point and the transition table for the DFA is shown in table 2.15.

$\longleftrightarrow \Sigma \longrightarrow$		
$\delta$	a	b
A	B	-
B	-	C



**Table 2.15 Transitional table**

The states C,D and E are final states, since 9 (final state of NFA) is present in C, D and E. The final transition diagram of DFA is shown in figure 2.36



**Fig. 2.36 The DFA**

## Regular Languages

### Regular expression

**Definition:** A regular expression is recursively defined as follows.

1.  $\phi$  is a regular expression denoting an empty language.
2.  $\epsilon$ -(epsilon) is a regular expression indicates the language containing an empty string.
3.  $a$  is a regular expression which indicates the language containing only  $\{a\}$
4. If  $R$  is a regular expression denoting the language  $L_R$  and  $S$  is a regular expression denoting the language  $L_S$ , then
  - a.  $R+S$  is a regular expression corresponding to the language  $L_R \cup L_S$ .
  - b.  $R.S$  is a regular expression corresponding to the language  $L_R \cdot L_S$ .
  - c.  $R^*$  is a regular expression corresponding to the language  $L_R^*$ .
5. The expressions obtained by applying any of the rules from 1-4 are regular expressions.

The table 3.1 shows some examples of regular expressions and the language corresponding to these regular expressions.

Regular	Meaning
---------	---------

expressions	
$(a+b)^*$	Set of strings of a's and b's of any length including the NULL string.
$(a+b)^*abb$	Set of strings of a's and b's ending with the string <i>abb</i>
$ab(a+b)^*$	Set of strings of a's and b's starting with the string <i>ab</i> .
$(a+b)^*aa(a+b)^*$	Set of strings of a's and b's having a sub string <i>aa</i> .
$a^*b^*c^*$	Set of string consisting of any number of a's(may be empty string also) followed by any number of b's(may include empty string) followed by any number of c's(may include empty string).
$a^+b^+c^+$	Set of string consisting of at least one 'a' followed by string consisting of at least one 'b' followed by string consisting of at least one 'c'.
$aa^*bb^*cc^*$	Set of string consisting of at least one 'a' followed by string consisting of at least one 'b' followed by string consisting of at least one 'c'.
$(a+b)^*(a+bb)$	Set of strings of a's and b's ending with either <i>a</i> or <i>bb</i>
$(aa)^*(bb)^*b$	Set of strings consisting of even number of a's followed by odd number of b's
$(0+1)^*000$	Set of strings of 0's and 1's ending with three consecutive zeros(or ending with 000)
$(11)^*$	Set consisting of even number of 1's

Table 3.1 Meaning of regular expressions

Obtain a regular expression to accept a language consisting of strings of a's and b's of even length.

String of a's and b's of even length can be obtained by the combination of the strings *aa*, *ab*, *ba* and *bb*. The language may even consist of an empty string denoted by  $\epsilon$ . So, the regular expression can be of the form

$$(aa + ab + ba + bb)^*$$

The  $*$  closure includes the empty string.

Note: This regular expression can also be represented using set notation as

$$L(R) = \{(aa + ab + ba + bb)^n \mid n \geq 0\}$$

Obtain a regular expression to accept a language consisting of strings of a's and b's of odd length.

String of a's and b's of odd length can be obtained by the combination of the strings *aa*, *ab*, *ba* and *bb* followed by either *a* or *b*. So, the regular expression can be of the form

$$(aa + ab + ba + bb)^* (a+b)$$

String of a's and b's of odd length can also be obtained by the combination of the strings aa, ab, ba and bb preceded by either *a* or *b*. So, the regular expression can also be represented as

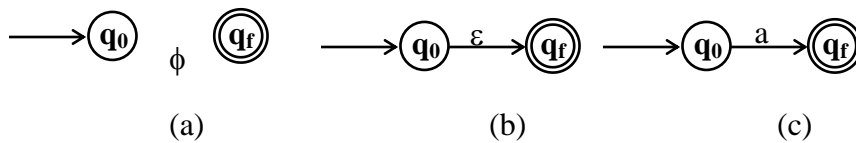
$$(a+b) (aa + ab + ba + bb)^*$$

Note: Even though these two expression are seems to be different, the language corresponding to those two expression is same. So, a variety of regular expressions can be obtained for a language and all are equivalent.

### Obtain NFA from the regular expression

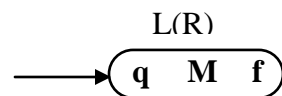
**Theorem:** Let *R* be a regular expression. Then there exists a finite automaton  $M = (Q, \Sigma, \delta, q_0, A)$  which accepts  $L(R)$ .

Proof: By definition,  $\phi$ ,  $\epsilon$  and *a* are regular expressions. So, the corresponding machines to recognize these expressions are shown in figure 3.1.a, 3.1.b and 3.1.c respectively.



**Fig 3.1 NFAs to accept  $\phi$ ,  $\epsilon$  and *a***

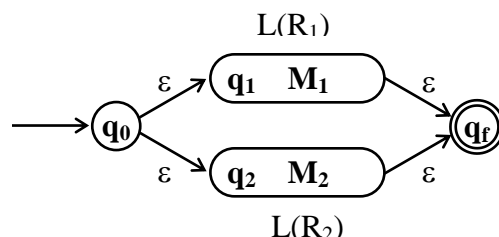
The schematic representation of a regular expression *R* to accept the language  $L(R)$  is shown in figure 3.2. where *q* is the start state and *f* is the final state of machine *M*.



**Fig 3.2 Schematic representation of FA accepting  $L(R)$**

In the definition of a regular expression it is clear that if *R* and *S* are regular expression, then *R*+*S* and *R*.*S* and *R*\* are regular expressions which clearly uses three operators '+', '.' and '\*'. Let us take each case separately and construct equivalent machine. Let  $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, f_1)$  be a machine which accepts the language  $L(R_1)$  corresponding to the regular expression *R*<sub>1</sub>. Let  $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, f_2)$  be a machine which accepts the language  $L(R_2)$  corresponding to the regular expression *R*<sub>2</sub>.

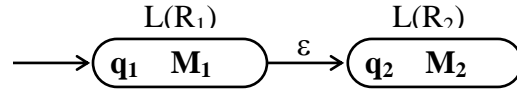
**Case 1:**  $R = R_1 + R_2$ . We can construct an NFA which accepts either  $L(R_1)$  or  $L(R_2)$  which can be represented as  $L(R_1 + R_2)$  as shown in figure 3.3.



**Fig. 3.3 To accept the language  $L(R_1 + R_2)$**

It is clear from figure 3.3 that the machine can either accept  $L(R_1)$  or  $L(R_2)$ . Here,  $q_0$  is the start state of the combined machine and  $q_f$  is the final state of combined machine M.

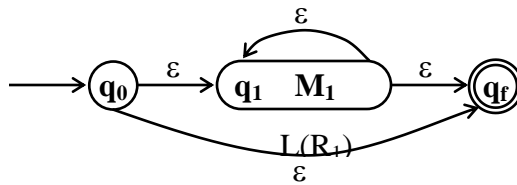
**Case 2:**  $R = R_1 \cdot R_2$ . We can construct an NFA which accepts  $L(R_1)$  followed by  $L(R_2)$  which can be represented as  $L(R_1 \cdot R_2)$  as shown in figure 3.4.



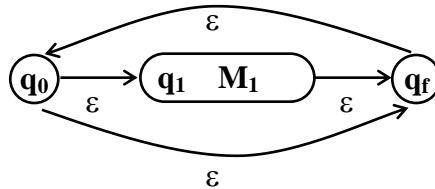
**Fig. 3.4 To accept the language  $L(R_1 \cdot R_2)$**

It is clear from figure 3.4 that the machine after accepting  $L(R_1)$  moves from state  $q_1$  to state  $q_2$ . Since there is a  $\epsilon$ -transition, without any input there will be a transition from state  $f_1$  to state  $q_2$ . In state  $q_2$ , upon accepting  $L(R_2)$ , the machine moves to  $f_2$  which is the final state. Thus,  $q_1$  which is the start state of machine  $M_1$  becomes the start state of the combined machine M and  $f_2$  which is the final state of machine  $M_2$ , becomes the final state of machine M and accepts the language  $L(R_1 \cdot R_2)$ .

**Case 3:**  $R = (R_1)^*$ . We can construct an NFA which accepts either  $L(R_1)^*$  as shown in figure 3.5.a. It can also be represented as shown in figure 3.5.b.



(a)



(b)



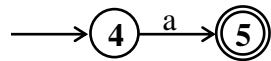
**Fig. 3.5 To accept the language  $L(R_1)^*$**

It is clear from figure 3.5 that the machine can either accept  $\epsilon$  or any number of  $L(R_1)$ s thus accepting the language  $L(R_1)^*$ . Here,  $q_0$  is the start state  $q_f$  is the final state.

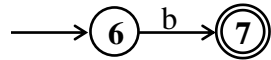
Obtain an NFA which accepts strings of a's and b's starting with the string ab.

The regular expression corresponding to this language is  $ab(a+b)^*$ .

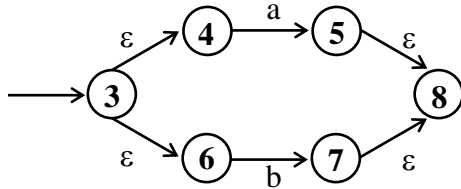
Step 1: The machine to accept 'a' is shown below.



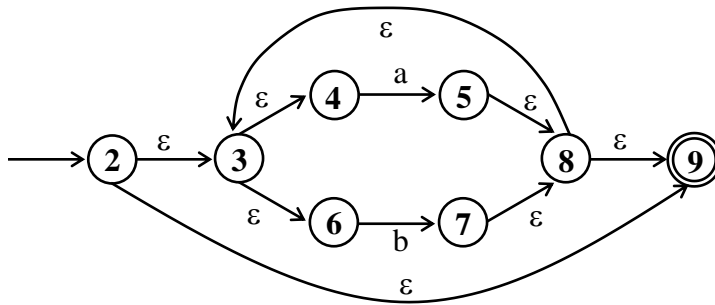
Step 2: The machine to accept 'b' is shown below.



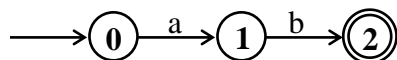
Step 3: The machine to accept (a + b) is shown below.



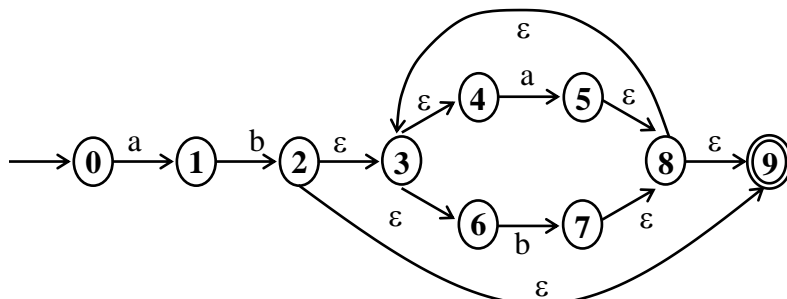
Step 4: The machine to accept  $(a+b)^*$  is shown below.



Step 5: The machine to accept ab is shown below.



Step 6: The machine to accept  $ab(a+b)^*$  is shown below.

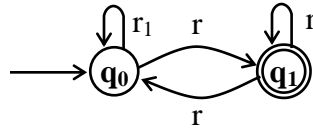


**Fig. 3.6 To accept the language  $L(ab(a+b)^*)$**

### Obtain the regular expression from FA

**Theorem:** Let  $M = (Q, \Sigma, \delta, q_0, A)$  be an FA recognizing the language  $L$ . Then there exists an equivalent regular expression  $R$  for the regular language  $L$  such that  $L = L(R)$ .

The general procedure to obtain a regular expression from FA is shown below. Consider the generalized graph



**Fig. 3.9 Generalized transition graph**

where  $r_1, r_2, r_3$  and  $r_4$  are the regular expressions and correspond to the labels for the edges. The regular expression for this can take the form:

$$r = r_1^* r_2 (r_4 + r_3 r_1^* r_2)^* \quad (3.1)$$

Note:

1. Any graph can be reduced to the graph shown in figure 3.9. Then substitute the regular expressions appropriately in the equation 3.1 and obtain the final regular expression.

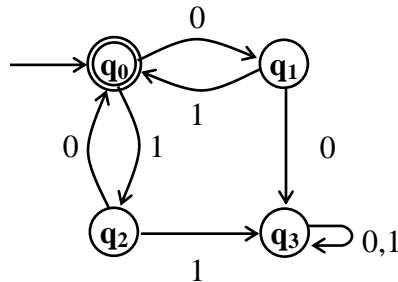
2. If  $r_3$  is not there in figure 3.9, the regular expression can be of the form

$$r = r_1^* r_2 r_4^* \quad (3.2)$$

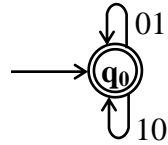
3. If  $q_0$  and  $q_1$  are the final states then the regular expression can be of the form

$$r = r_1^* + r_1^* r_2 r_4^* \quad (3.3)$$

Obtain a regular expression for the FA shown below:



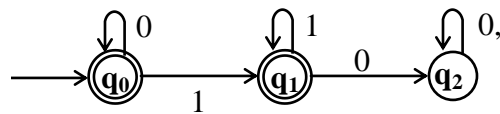
The figure can be reduced as shown below:



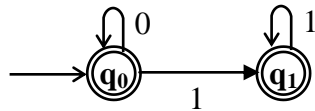
It is clear from this figure that the machine accepts strings of 01's and 10's of any length and the regular expression can be of the form

$$(01 + 10)^*$$

What is the language accepted by the following FA



Since, state  $q_2$  is the dead state, it can be removed and the following FA is obtained.



The state  $q_0$  is the final state and at this point it can accept any number of 0's which can be represented using notation as

$$0^*$$

$q_1$  is also the final state. So, to reach  $q_1$  one can input any number of 0's followed by 1 and followed by any number of 1's and can be represented as

$$0^* 1 1^*$$

So, the final regular expression is obtained by adding  $0^*$  and  $0^* 1 1^*$ . So, the regular expression is

$$\begin{aligned} \text{R.E} &= 0^* + 0^* 1 1^* \\ &= 0^* ( \epsilon + 1 1^* ) \\ &= 0^* ( \epsilon + 1^+ ) \\ &= 0^* ( 1^* ) = 0^* 1^* \end{aligned}$$

It is clear from the regular expression that language consists of any number of 0's (possibly  $\epsilon$ ) followed by any number of 1's (possibly  $\epsilon$ ).

## Applications of Regular Expressions

**Pattern Matching** refers to a set of objects with some common properties. We can match an identifier or a decimal number or we can search for a string in the text.

An application of regular expression in UNIX editor *ed*.

In UNIX operating system, we can use the editor *ed* to search for a specific pattern in the text. For example, if the command specified is

*/acb\*c/*

then the editor searches for a string which starts with *ac* followed by zero or more *b*'s and followed by the symbol *c*. Note that the editor *ed* accepts the regular expression and searches for that particular pattern in the text. As the input can vary dynamically, it is challenging to write programs for string patterns of these kinds.

**Definition:** A DFA is 5-tuple or quintuple  $M = (Q, \Sigma, \delta, q_0, A)$  where

$Q$  is non-empty, finite set of states.

$\Sigma$  is non-empty, finite set of input alphabets.

$\delta$  is transition function, which is a mapping from  $Q \times \Sigma$  to  $Q$ .

$q_0 \in Q$  is the start state.

$A \subseteq Q$  is set of accepting or final states.

Note: For each input symbol  $a$ , from a given state there is exactly one transition (there can be no transitions from a state also) and we are sure (or can determine) to which state the machine enters. So, the machine is called **Deterministic** machine. Since it has finite number of states the machine is called Deterministic finite machine or Deterministic Finite Automaton or Finite State Machine (FSM).

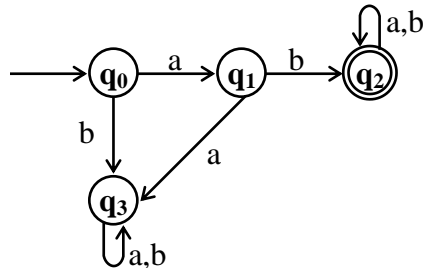
The language accepted by DFA is

$$L(M) = \{ w \mid w \in \Sigma^* \text{ and } \delta^*(q_0, w) \in A \}$$

The non-acceptance of the string  $w$  by an FA or DFA can be defined in formal notation as:

$$\overline{L(M)} = \{ w \mid w \in \Sigma^* \text{ and } \delta^*(q_0, w) \notin A \}$$

**Obtain a DFA to accept strings of a's and b's starting with the string ab**



**Fig.2.7 Transition diagram to accept string  $ab(a+b)^*$**

So, the DFA which accepts strings of a's and b's starting with the string  $ab$  is given by  $M = (Q, \Sigma, \delta, q_0, A)$  where

$$Q = \{q_0, q_1, q_2, q_3\}$$

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

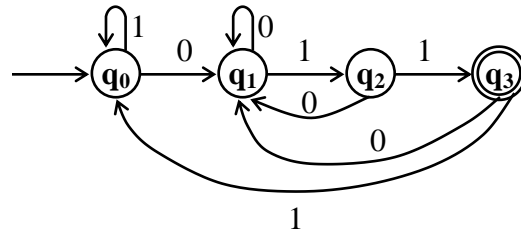
$q_0$  is the start state

$$A = \{q_2\}.$$

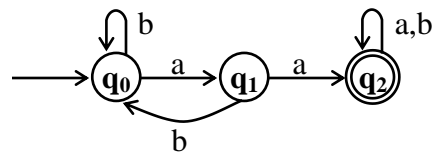
$\delta$  is shown the transition table 2.4.

		$\leftarrow \Sigma \rightarrow$	
		a	b
$\downarrow$ States $\uparrow$	$\rightarrow q_0$	$q_1$	$q_3$
	$q_1$	$q_3$	$q_2$
	$\odot q_2$	$q_2$	$q_2$
	$q_3$	$q_3$	$q_3$

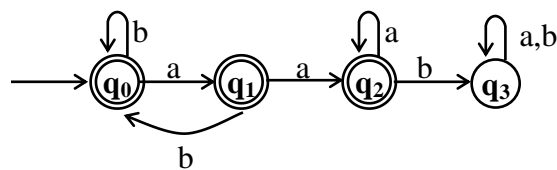
Draw a DFA to accept string of 0's and 1's ending with the string 011.



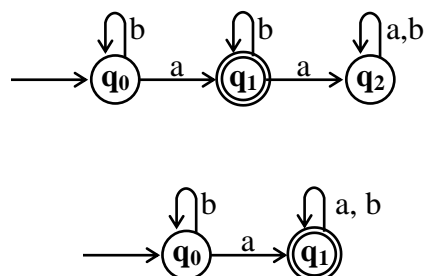
Obtain a DFA to accept strings of a's and b's having a sub string aa

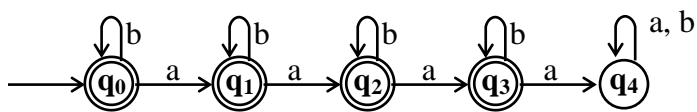


Obtain a DFA to accept strings of a's and b's except those containing the substring aab.



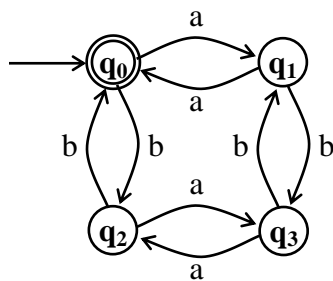
Obtain DFAs to accept strings of a's and b's having exactly one a,



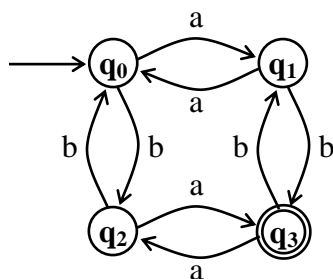
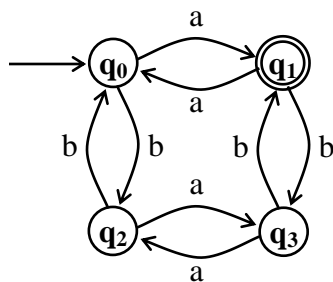
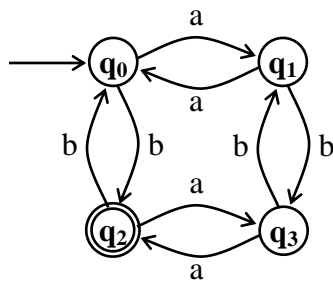


**Obtain a DFA to accept strings of a's and b's having even number of a's and b's**

The machine to accept even number of a's and b's is shown in fig.2.22.



**Fig.2.22 DFA to accept even no. of a's and b's**



## Regular language

**Definition:** Let  $M = (Q, \Sigma, \delta, q_0, A)$  be a DFA. The language  $L$  is regular if there exists a machine  $M$  such that  $L = L(M)$ .

## \* Applications of Finite Automata \*

### String matching/processing

### Compiler Construction

The various compilers such as C/C++, Pascal, Fortran or any other compiler is designed using the finite automata. The DFAs are extensively used in the building the various phases of compiler such as

- Lexical analysis (To identify the tokens, identifiers, to strip of the comments etc.)
- Syntax analysis (To check the syntax of each statement or control statement used in the program)
- Code optimization (To remove the un wanted code)
- Code generation (To generate the machine code)

### Other applications

The concept of finite automata is used in wide applications. It is not possible to list all the applications as there are infinite number of applications. This section lists some applications:

5. Large natural vocabularies can be described using finite automaton which includes the applications such as spelling checkers and advisers, multi-language dictionaries, to indent the documents, in calculators to evaluate complex expressions based on the priority of an operator etc. to name a few. Any editor that we use uses finite automaton for implementation.
6. Finite automaton is very useful in recognizing difficult problems i.e., sometimes it is very essential to solve an un-decidable problem. Even though there is no general solution exists for the specified problem, using theory of computation, we can find the approximate solutions.
7. Finite automaton is very useful in hardware design such as circuit verification, in design of the hardware board (mother board or any other hardware unit), automatic traffic signals, radio controlled toys, elevators, automatic sensors, remote sensing or controller etc.
8. In game theory and games wherein we use some control characters to fight against a monster, economics, computer graphics, linguistics etc., finite automaton plays a very important role.

## Non deterministic finite automata(NFA)

**Definition:** An NFA is a 5-tuple or quintuple  $M = (Q, \Sigma, \delta, q_0, A)$  where  $Q$  is non empty, finite set of states.



$\Sigma$  is non empty, finite set of input alphabets.

$\delta$  is transition function which is a mapping from  $Q \times \{\Sigma \cup \varepsilon\}$  to subsets of  $2^Q$ . This function shows the change of state from one state to a set of states based on the input symbol.

$q_0 \in Q$  is the start state.

$A \subseteq Q$  is set of final states.

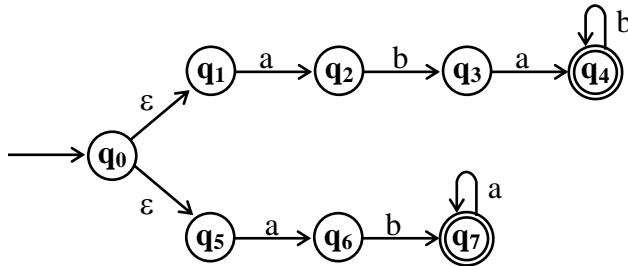
### Acceptance of language

**Definition:** Let  $M = (Q, \Sigma, \delta, q_0, A)$  be a DFA where  $Q$  is set of finite states,  $\Sigma$  is set of input alphabets (from which a string can be formed),  $\delta$  is transition function from  $Q \times \{\Sigma \cup \varepsilon\}$  to  $2^Q$ ,  $q_0$  is the start state and  $A$  is the final or accepting state. The string (also called language)  $w$  accepted by an NFA can be defined in formal notation as:

$$L(M) = \{ w \mid w \in \Sigma^* \text{ and } \delta^*(q_0, w) = Q \text{ with atleast one component of } Q \text{ in } A \}$$

**Obtain an NFA to accept the following language  $L = \{w \mid w \in abab^n \text{ or } aba^n \text{ where } n \geq 0\}$**

The machine to accept either  $abab^n$  or  $aba^n$  where  $n \geq 0$  is shown below:



### 2.2 Conversion from NFA to DFA

Let  $M_N = (Q_N, \Sigma_N, \delta_N, q_0, A_N)$  be an NFA and accepts the language  $L(M_N)$ . There should be an equivalent DFA  $M_D = (Q_D, \Sigma_D, \delta_D, q_0, A_D)$  such that  $L(M_D) = L(M_N)$ . The procedure to convert an NFA to its equivalent DFA is shown below:

Step1:

The start state of NFA  $M_N$  is the start state of DFA  $M_D$ . So, add  $q_0$  (which is the start state of NFA) to  $Q_D$  and find the transitions from this state. The way to obtain different transitions is shown in step2.

Step2:

For each state  $[q_i, q_j, \dots, q_k]$  in  $Q_D$ , the transitions for each input symbol in  $\Sigma$  can be obtained as shown below:

$$4. \delta_D([q_i, q_j, \dots, q_k], a) = \delta_N(q_i, a) \cup \delta_N(q_j, a) \cup \dots \delta_N(q_k, a) \\ = [q_l, q_m, \dots, q_n] \text{ say.}$$

5. Add the state  $[q_l, q_m, \dots, q_n]$  to  $Q_D$ , if it is not already in  $Q_D$ .

6. Add the transition from  $[q_i, q_j, \dots, q_k]$  to  $[q_l, q_m, \dots, q_n]$  on the input symbol  $a$  iff the state  $[q_l, q_m, \dots, q_n]$  is added to  $Q_D$  in the previous step.

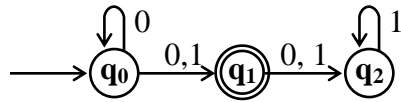
Step3:

The state  $[q_a, q_b, \dots, q_c] \in Q_D$  is the final state, if at least one of the state in  $q_a, q_b, \dots, q_c \in A_N$  i.e., at least one of the component in  $[q_a, q_b, \dots, q_c]$  should be the final state of NFA.

Step4:

If epsilon ( $\epsilon$ ) is accepted by NFA, then start state  $q_0$  of DFA is made the final state.

**Convert the following NFA into an equivalent DFA.**



Step1:  $q_0$  is the start of DFA (see step1 in the conversion procedure).

$$\text{So, } Q_D = \{[q_0]\} \quad (2.7)$$

Step2: Find the new states from each state in  $Q_D$  and obtain the corresponding transitions.

**Consider the state  $[q_0]$ :**

When  $a = 0$

$$\delta_D([q_0], 0) = \delta_N([q_0], 0) \\ = [q_0, q_1] \quad (2.8)$$

When  $a = 1$

$$\delta_D([q_0], 1) = \delta_N([q_0], 1) \\ = [q_1] \quad (2.9)$$

Since the states obtained in (2.8) and (2.9) are not in  $Q_D(2.7)$ , add these two states to  $Q_D$  so that

$$Q_D = \{[q_0], [q_0, q_1], [q_1]\} \quad (2.10)$$

The corresponding transitions on  $a = 0$  and  $a = 1$  are shown below.

$\Sigma$

		$\longleftrightarrow$	$\longrightarrow$
	$\delta$	0	1
$\uparrow$	$[q_0]$	$[q_0, q_1]$	$[q_1]$
$Q$	$[q_0, q_1]$		
$\downarrow$	$[q_1]$		

**Consider the state  $[q_0, q_1]$ :**

When  $a = 0$

$$\begin{aligned}
 \delta_D([q_0, q_1], 0) &= \delta_N([q_0, q_1], 0) \\
 &= \delta_N(q_0, 0) \cup \delta_N(q_1, 0) \\
 &= \{q_0, q_1\} \cup \{q_2\} \\
 &= [q_0, q_1, q_2] \\
 &\quad (2.11)
 \end{aligned}$$

When  $a = 1$

$$\begin{aligned}
 \delta_D([q_0, q_1], 1) &= \delta_N([q_0, q_1], 1) \\
 &= \delta_N(q_0, 1) \cup \delta_N(q_1, 1) \\
 &= \{q_1\} \cup \{q_2\} \\
 &= [q_1, q_2] \\
 &\quad (2.12)
 \end{aligned}$$

Since the states obtained in (2.11) and (2.12) are not defined in  $Q_D$  (see 2.10), add these two states to  $Q_D$  so that

$$Q_D = \{[q_0], [q_0, q_1], [q_1], [q_0, q_1, q_2], [q_1, q_2]\} \quad (2.13)$$

and add the transitions on  $a = 0$  and  $a = 1$  as shown below:

		$\longleftrightarrow$	$\xrightarrow{\Sigma}$
	$\delta$	0	1
$\uparrow$	$[q_0]$	$[q_0, q_1]$	$[q_1]$
$Q$	$[q_0, q_1]$	$[q_0, q_1, q_2]$	$[q_1, q_2]$
$\downarrow$	$[q_1]$		
	$[q_0, q_1, q_2]$		
	$[q_1, q_2]$		

**Consider the state  $[q_1]$ :**

When  $a = 0$

$$\begin{aligned}
 \delta_D([q_1], 0) &= \delta_N([q_1], 0) \\
 &= [q_2] \\
 &\quad (2.14)
 \end{aligned}$$

When  $a = 1$

$$\begin{aligned}
\delta_D([q_1], 1) &= \delta_N([q_1], 1) \\
&= [q_2] \\
(2.15)
\end{aligned}$$

Since the states obtained in (2.14) and (2.15) are same and the state  $q_2$  is not in  $Q_D$  (see 2.13), add the state  $q_2$  to  $Q_D$  so that

$$Q_D = \{[q_0], [q_0, q_1], [q_1], [q_0, q_1, q_2], [q_1, q_2], [q_2]\} \quad (2.16)$$

and add the transitions on  $a = 0$  and  $a = 1$  as shown below:

		$\longleftrightarrow \Sigma \longrightarrow$	
$\uparrow$ $Q$ $\downarrow$	$\delta$	0	1
	$[q_0]$	$[q_0, q_1]$	$[q_1]$
	$[q_0, q_1]$	$[q_0, q_1, q_2]$	$[q_1, q_2]$
	$[q_1]$	$[q_2]$	$[q_2]$
	$[q_0, q_1, q_2]$		
	$[q_1, q_2]$		
	$[q_2]$		

**Consider the state  $[q_0, q_1, q_2]$ :**

When  $a = 0$

$$\begin{aligned}
\delta_D([q_0, q_1, q_2], 0) &= \delta_N([q_0, q_1, q_2], 0) \\
&= \delta_N(q_0, 0) \cup \delta_N(q_1, 0) \cup \delta_N(q_2, 0) \\
&= \{q_0, q_1\} \cup \{q_2\} \cup \{\phi\} \\
&= [q_0, q_1, q_2] \\
(2.17)
\end{aligned}$$

When  $a = 1$

$$\begin{aligned}
\delta_D([q_0, q_1, q_2], 1) &= \delta_N([q_0, q_1, q_2], 1) \\
&= \delta_N(q_0, 1) \cup \delta_N(q_1, 1) \cup \delta_N(q_2, 1) \\
&= \{q_1\} \cup \{q_2\} \cup \{q_2\} \\
&= [q_1, q_2] \\
(2.18)
\end{aligned}$$

Since the states obtained in (2.17) and (2.18) are not new states (are already in  $Q_D$ , see 2.16), do not add these two states to  $Q_D$ . But, the transitions on  $a = 0$  and  $a = 1$  should be added to the transitional table as shown below:

		$\longleftrightarrow \Sigma \longrightarrow$		
		$\delta$	0	1
Q	$\uparrow$	$[q_0]$	$[q_0, q_1]$	$[q_1]$
		$[q_0, q_1]$	$[q_0, q_1, q_2]$	$[q_1, q_2]$
		$[q_1]$	$[q_2]$	$[q_2]$
		$[q_0, q_1, q_2]$	$[q_0, q_1, q_2]$	$[q_1, q_2]$
	$\downarrow$	$[q_1, q_2]$		
		$[q_2]$		

**Consider the state  $[q_1, q_2]$ :**

When  $a = 0$

$$\begin{aligned}
 \delta_D([q_1, q_2], 0) &= \delta_N([q_1, q_2], 0) \\
 &= \delta_N(q_1, 0) \cup \delta_N(q_2, 0) \\
 &= \{q_2\} \cup \{\phi\} \\
 &= [q_2] \\
 &\text{(2.19)}
 \end{aligned}$$

When  $a = 1$

$$\begin{aligned}
 \delta_D([q_1, q_2], 1) &= \delta_N([q_1, q_2], 1) \\
 &= \delta_N(q_1, 1) \cup \delta_N(q_2, 1) \\
 &= \{q_2\} \cup \{q_2\} \\
 &= [q_2] \\
 &\text{(2.20)}
 \end{aligned}$$

Since the states obtained in (2.19) and (2.20) are not new states (are already in  $Q_D$  see 2.16), do not add these two states to  $Q_D$ . But, the transitions on  $a = 0$  and  $a = 1$  should be added to the transitional table as shown below:

		$\longleftrightarrow \Sigma \longrightarrow$		
		$\delta$	0	1
Q	$\uparrow$	$[q_0]$	$[q_0, q_1]$	$[q_1]$
		$[q_0, q_1]$	$[q_0, q_1, q_2]$	$[q_1, q_2]$
		$[q_1]$	$[q_2]$	$[q_2]$
		$[q_0, q_1, q_2]$	$[q_0, q_1, q_2]$	$[q_1, q_2]$
	$\downarrow$	$[q_1, q_2]$	$[q_2]$	$[q_2]$
		$[q_2]$		

**Consider the state  $[q_2]$ :**

When  $a = 0$

$$\begin{aligned}\delta_D([q_2], 0) &= \delta_N([q_2], 0) \\ &= \{\phi\} \\ (2.21)\end{aligned}$$

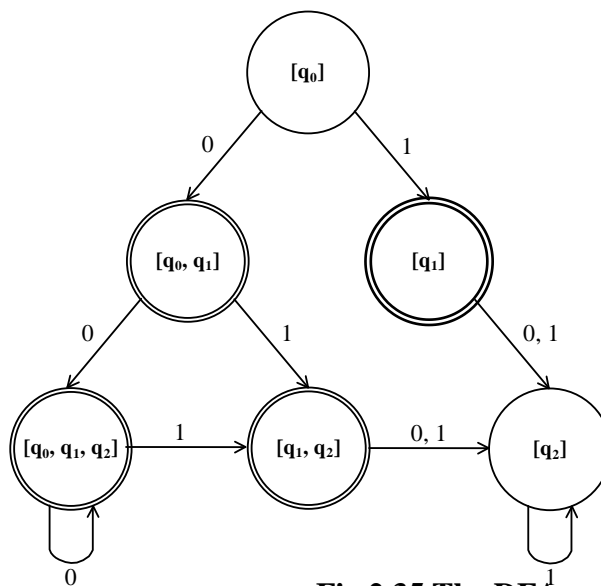
When  $a = 1$

$$\begin{aligned}\delta_D([q_2], 1) &= \delta_N([q_2], 1) \\ &= [q_2] \\ (2.22)\end{aligned}$$

Since the states obtained in (2.21) and (2.22) are not new states (are already in  $Q_D$ , see 2.16), do not add these two states to  $Q_D$ . But, the transitions on  $a = 0$  and  $a = 1$  should be added to the transitional table. The final transitional table is shown in table 2.14. and final DFA is shown in figure 2.35.

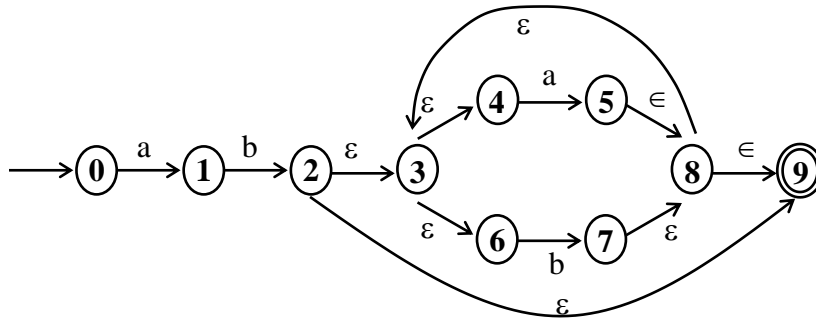
		$\longleftrightarrow \Sigma \longrightarrow$	
$\updownarrow Q$	$\delta$	0	1
	$[q_0]$	$[q_0, q_1]$	$[q_1]$
	$[q_0, q_1]$	$[q_0, q_1, q_2]$	$[q_1, q_2]$
	$[q_1]$	$[q_2]$	$[q_2]$
	$[q_0, q_1, q_2]$	$[q_0, q_1, q_2]$	$[q_1, q_2]$
	$[q_1, q_2]$	$[q_2]$	$[q_2]$
	$[q_2]$	$\phi$	$[q_2]$

**Table 2.14 Transitional table**



**Fig.2.35 The DFA**

Convert the following NFA to its equivalent DFA.




---

Let  $Q_D = \{0\}$  (A)

**Consider the state [A]:**

When input is  $a$ :

$$\begin{aligned} \delta(A, a) &= \delta_N(0, a) \\ &= \{1\} \\ &\text{(B)} \end{aligned}$$

When input is  $b$ :

$$\begin{aligned} \delta(A, b) &= \delta_N(0, b) \\ &= \{\phi\} \end{aligned}$$

**Consider the state [B]:**

When input is  $a$ :

$$\begin{aligned} \delta(B, a) &= \delta_N(1, a) \\ &= \{\phi\} \end{aligned}$$

When input is  $b$ :

$$\begin{aligned} \delta(B, b) &= \delta_N(1, b) \\ &= \{2\} \\ &= \{2, 3, 4, 6, 9\} \end{aligned} \quad \text{(C)}$$

This is because, in state 2, due to  $\epsilon$ -transitions (or without giving any input) there can be transition to states 3, 4, 6, 9 also. So, all these states are reachable from state 2. Therefore,

$$\delta(B, b) = \{2, 3, 4, 6, 9\} = C$$

**Consider the state [C]:**

When input is  $a$ :

$$\begin{aligned} \delta(C, a) &= \delta_N(\{2, 3, 4, 6, 9\}, a) \\ &= \{5\} \end{aligned}$$

$$\begin{aligned}
&= \{5, 8, 9, 3, 4, 6\} \\
&= \{3, 4, 5, 6, 8, 9\} \quad (\text{ascending order}) \quad (D)
\end{aligned}$$

This is because, in state 5 due to  $\varepsilon$ -transitions, the states reachable are  $\{8, 9, 3, 4, 6\}$ . Therefore,

$$\delta(C, a) = \{3, 4, 5, 6, 8, 9\} = D$$

When input is  $b$ :

$$\begin{aligned}
\delta(C, b) &= \delta_N(\{2, 3, 4, 6, 9\}, b) \\
&= \{7\} \\
&= \{7, 8, 9, 3, 4, 6\} \\
&= \{3, 4, 6, 7, 8, 9\} (\text{ascending order}) \\
&\quad (E)
\end{aligned}$$

This is because, from state 7 the states that are reachable without any input (i.e.,  $\varepsilon$ -transition) are  $\{8, 9, 3, 4, 6\}$ . Therefore,

$$\delta(C, b) = \{3, 4, 6, 7, 8, 9\} = E$$

**Consider the state [D]:**

When input is  $a$ :

$$\begin{aligned}
\delta(D, a) &= \delta_N(\{3, 4, 5, 6, 8, 9\}, a) \\
&= \{5\} \\
&= \{5, 8, 9, 3, 4, 6\} \\
&= \{3, 4, 5, 6, 8, 9\} \quad (\text{ascending order}) \quad (D)
\end{aligned}$$

When input is  $b$ :

$$\begin{aligned}
\delta(D, b) &= \delta_N(\{3, 4, 5, 6, 8, 9\}, b) \\
&= \{7\} \\
&= \{7, 8, 9, 3, 4, 6\} \\
&= \{3, 4, 6, 7, 8, 9\} \quad (\text{ascending order}) \quad (E)
\end{aligned}$$

**Consider the state [E]:**

When input is  $a$ :

$$\begin{aligned}
\delta(E, a) &= \delta_N(\{3, 4, 6, 7, 8, 9\}, a) \\
&= \{5\} \\
&= \{5, 8, 9, 3, 4, 6\} \\
&= \{3, 4, 5, 6, 8, 9\} (\text{ascending order}) \\
&\quad (D)
\end{aligned}$$



When input is  $b$ :

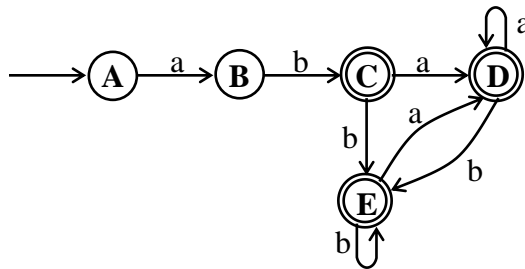
$$\begin{aligned}
 \delta(E, b) &= \delta_N(\{3,4,6,7,8,9\}, b) \\
 &= \{7\} \\
 &= \{7, 8, 9, 3, 4, 6\} \\
 &= \{3, 4, 6, 7, 8, 9\}(\text{ascending order}) \\
 &\quad (E)
 \end{aligned}$$

Since there are no new states, we can stop at this point and the transition table for the DFA is shown in table 2.15.

		$\longleftrightarrow \Sigma \longrightarrow$	
	$\delta$	a	b
$\updownarrow Q$	A	B	-
	B	-	C
	<u>C</u>	D	E
	<u>D</u>	D	E
	<u>E</u>	D	E

**Table 2.15 Transitional table**

The states C,D and E are final states, since 9 (final state of NFA) is present in C, D and E. The final transition diagram of DFA is shown in figure 2.36



**Fig. 2.36 The DFA**

# Regular Languages

## Regular expression

**Definition:** A regular expression is recursively defined as follows.

6.  $\phi$  is a regular expression denoting an empty language.
7.  $\epsilon$ -(epsilon) is a regular expression indicates the language containing an empty string.
8.  $a$  is a regular expression which indicates the language containing only  $\{a\}$
9. If  $R$  is a regular expression denoting the language  $L_R$  and  $S$  is a regular expression denoting the language  $L_S$ , then
  - a.  $R+S$  is a regular expression corresponding to the language  $L_R \cup L_S$ .
  - b.  $R.S$  is a regular expression corresponding to the language  $L_R \cdot L_S$ .
  - c.  $R^*$  is a regular expression corresponding to the language  $L_R^*$ .
10. The expressions obtained by applying any of the rules from 1-4 are regular expressions.

The table 3.1 shows some examples of regular expressions and the language corresponding to these regular expressions.

Regular expressions	Meaning
$(a+b)^*$	Set of strings of a's and b's of any length including the NULL string.
$(a+b)^*abb$	Set of strings of a's and b's ending with the string $abb$
$ab(a+b)^*$	Set of strings of a's and b's starting with the string $ab$ .
$(a+b)^*aa(a+b)^*$	Set of strings of a's and b's having a sub string $aa$ .
$a^*b^*c^*$	Set of string consisting of any number of a's(may be empty string also) followed by any number of b's(may include empty string) followed by any number of c's(may include empty string).
$a^+b^+c^+$	Set of string consisting of at least one 'a' followed by string consisting of at least one 'b' followed by string consisting of at least one 'c'.
$aa^*bb^*cc^*$	Set of string consisting of at least one 'a' followed by string consisting of at least one 'b' followed by string consisting of at least one 'c'.
$(a+b)^* (a + bb)$	Set of strings of a's and b's ending with either $a$ or $bb$
$(aa)^*(bb)^*b$	Set of strings consisting of even number of a's followed by odd number of b's
$(0+1)^*000$	Set of strings of 0's and 1's ending with three

	consecutive zeros(or ending with 000)
(11)*	Set consisting of even number of 1's

**Table 3.1 Meaning of regular expressions**

Obtain a regular expression to accept a language consisting of strings of a's and b's of even length.

String of a's and b's of even length can be obtained by the combination of the strings aa, ab, ba and bb. The language may even consist of an empty string denoted by  $\epsilon$ . So, the regular expression can be of the form

$$(aa + ab + ba + bb)^*$$

The \* closure includes the empty string.

Note: This regular expression can also be represented using set notation as

$$L(R) = \{(aa + ab + ba + bb)^n \mid n \geq 0\}$$

Obtain a regular expression to accept a language consisting of strings of a's and b's of odd length.

String of a's and b's of odd length can be obtained by the combination of the strings aa, ab, ba and bb followed by either  $a$  or  $b$ . So, the regular expression can be of the form

$$(aa + ab + ba + bb)^* (a+b)$$

String of a's and b's of odd length can also be obtained by the combination of the strings aa, ab, ba and bb preceded by either  $a$  or  $b$ . So, the regular expression can also be represented as

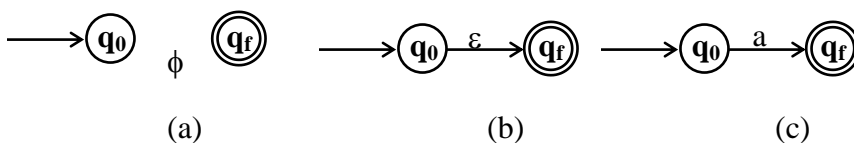
$$(a+b) (aa + ab + ba + bb)^*$$

Note: Even though these two expressions seem to be different, the language corresponding to those two expressions is the same. So, a variety of regular expressions can be obtained for a language and all are equivalent.

### Obtain NFA from the regular expression

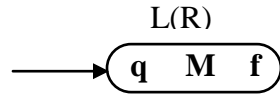
**Theorem:** Let  $R$  be a regular expression. Then there exists a finite automaton  $M = (Q, \Sigma, \delta, q_0, A)$  which accepts  $L(R)$ .

Proof: By definition,  $\phi$ ,  $\epsilon$  and  $a$  are regular expressions. So, the corresponding machines to recognize these expressions are shown in figure 3.1.a, 3.1.b and 3.1.c respectively.



**Fig 3.1 NFAs to accept  $\phi$ ,  $\varepsilon$  and  $a$**

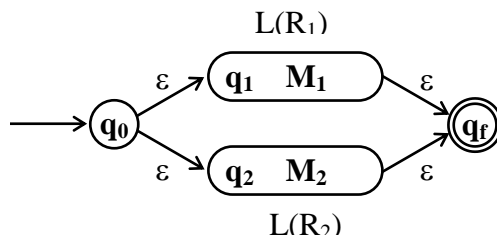
The schematic representation of a regular expression  $R$  to accept the language  $L(R)$  is shown in figure 3.2. where  $q$  is the start state and  $f$  is the final state of machine  $M$ .



**Fig 3.2 Schematic representation of FA accepting  $L(R)$**

In the definition of a regular expression it is clear that if  $R$  and  $S$  are regular expression, then  $R+S$  and  $R.S$  and  $R^*$  are regular expressions which clearly uses three operators '+', '.' and '\*'. Let us take each case separately and construct equivalent machine. Let  $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, f_1)$  be a machine which accepts the language  $L(R_1)$  corresponding to the regular expression  $R_1$ . Let  $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, f_2)$  be a machine which accepts the language  $L(R_2)$  corresponding to the regular expression  $R_2$ .

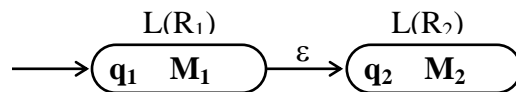
**Case 1:**  $R = R_1 + R_2$ . We can construct an NFA which accepts either  $L(R_1)$  or  $L(R_2)$  which can be represented as  $L(R_1 + R_2)$  as shown in figure 3.3.



**Fig. 3.3 To accept the language  $L(R_1 + R_2)$**

It is clear from figure 3.3 that the machine can either accept  $L(R_1)$  or  $L(R_2)$ . Here,  $q_0$  is the start state of the combined machine and  $q_f$  is the final state of combined machine  $M$ .

**Case 2:**  $R = R_1 . R_2$ . We can construct an NFA which accepts  $L(R_1)$  followed by  $L(R_2)$  which can be represented as  $L(R_1 . R_2)$  as shown in figure 3.4.

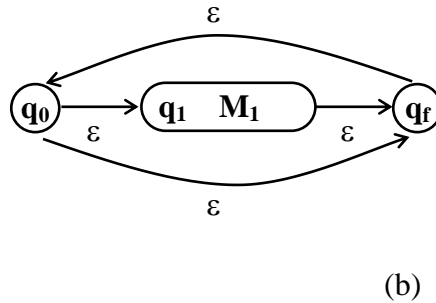
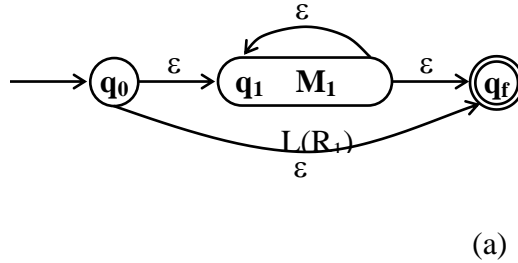


**Fig. 3.4 To accept the language  $L(R_1 . R_2)$**

It is clear from figure 3.4 that the machine after accepting  $L(R_1)$  moves from state  $q_1$  to  $f_1$ . Since there is a  $\varepsilon$ -transition, without any input there will be a transition from state  $f_1$  to state  $q_2$ . In state  $q_2$ , upon accepting  $L(R_2)$ , the machine moves to  $f_2$  which is the final state. Thus,  $q_1$  which is the start state of machine  $M_1$  becomes the start state of the combined machine  $M$ .

and  $f_2$  which is the final state of machine  $M_2$ , becomes the final state of machine  $M$  and accepts the language  $L(R_1.R_2)$ .

**Case 3:**  $R = (R_1)^*$ . We can construct an NFA which accepts either  $L(R_1)^*$  as shown in figure 3.5.a. It can also be represented as shown in figure 3.5.b.



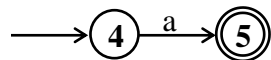
**Fig. 3.5 To accept the language  $L(R_1)^*$**

It is clear from figure 3.5 that the machine can either accept  $\epsilon$  or any number of  $L(R_1)$ s thus accepting the language  $L(R_1)^*$ . Here,  $q_0$  is the start state  $q_f$  is the final state.

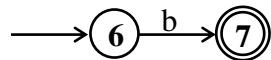
Obtain an NFA which accepts strings of a's and b's starting with the string ab.

The regular expression corresponding to this language is  $ab(a+b)^*$ .

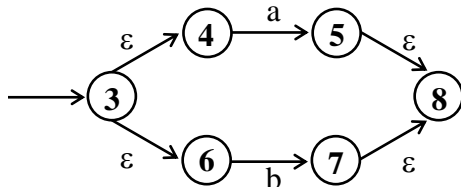
Step 1: The machine to accept 'a' is shown below.



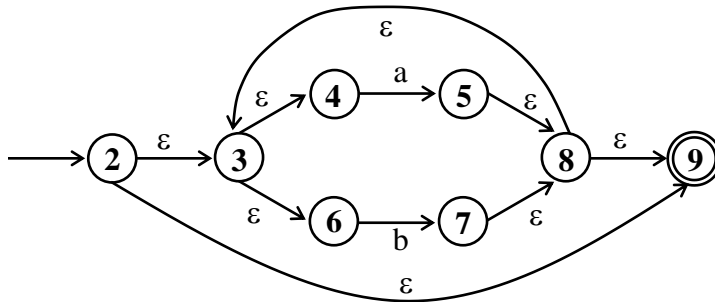
Step 2: The machine to accept 'b' is shown below.



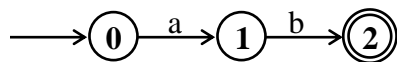
Step 3: The machine to accept  $(a + b)$  is shown below.



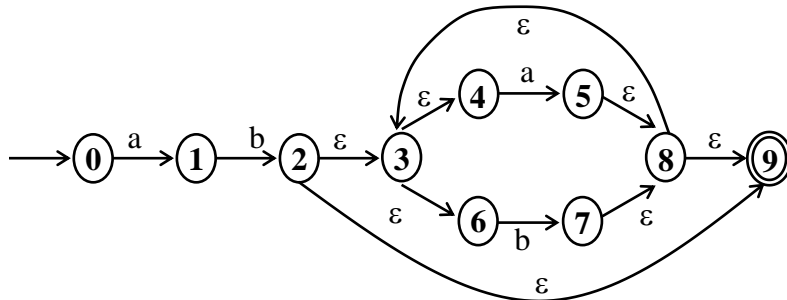
Step 4: The machine to accept  $(a+b)^*$  is shown below.



Step 5: The machine to accept  $ab$  is shown below.



Step 6: The machine to accept  $ab(a+b)^*$  is shown below.

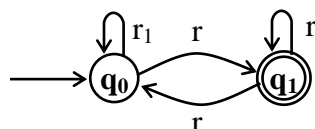


**Fig. 3.6 To accept the language  $L(ab(a+b)^*)$**

### Obtain the regular expression from FA

**Theorem:** Let  $M = (Q, \Sigma, \delta, q_0, A)$  be an FA recognizing the language  $L$ . Then there exists an equivalent regular expression  $R$  for the regular language  $L$  such that  $L = L(R)$ .

The general procedure to obtain a regular expression from FA is shown below. Consider the generalized graph



**Fig. 3.9 Generalized transition graph**

where  $r_1$ ,  $r_2$ ,  $r_3$  and  $r_4$  are the regular expressions and correspond to the labels for the edges. The regular expression for this can take the form:

$$r = r_1^* r_2 (r_4 + r_3 r_1^* r_2)^* \quad (3.1)$$

Note:

4. Any graph can be reduced to the graph shown in figure 3.9. Then substitute the regular expressions appropriately in the equation 3.1 and obtain the final regular expression.

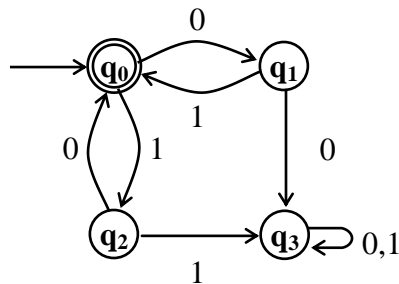
5. If  $r_3$  is not there in figure 3.9, the regular expression can be of the form

$$r = r_1^* r_2 r_4^* \quad (3.2)$$

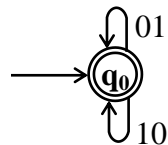
6. If  $q_0$  and  $q_1$  are the final states then the regular expression can be of the form

$$r = r_1^* + r_1^* r_2 r_4^* \quad (3.3)$$

Obtain a regular expression for the FA shown below:



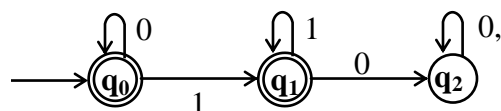
The figure can be reduced as shown below:



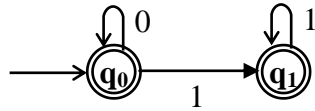
It is clear from this figure that the machine accepts strings of 01's and 10's of any length and the regular expression can be of the form

$$(01 + 10)^*$$

What is the language accepted by the following FA



Since, state  $q_2$  is the dead state, it can be removed and the following FA is obtained.



The state  $q_0$  is the final state and at this point it can accept any number of 0's which can be represented using notation as

$$0^*$$

$q_1$  is also the final state. So, to reach  $q_1$  one can input any number of 0's followed by 1 and followed by any number of 1's and can be represented as

$$0^* 1 1^*$$

So, the final regular expression is obtained by adding  $0^*$  and  $0^* 1 1^*$ . So, the regular expression is

$$\begin{aligned}
 \text{R.E} &= 0^* + 0^* 1 1^* \\
 &= 0^* ( \epsilon + 1 1^* ) \\
 &= 0^* ( \epsilon + 1^+ ) \\
 &= 0^* ( 1^* ) = 0^* 1^*
 \end{aligned}$$

It is clear from the regular expression that language consists of any number of 0's (possibly  $\epsilon$ ) followed by any number of 1's (possibly  $\epsilon$ ).

## Applications of Regular Expressions

**Pattern Matching** refers to a set of objects with some common properties. We can match an identifier or a decimal number or we can search for a string in the text.

An application of regular expression in UNIX editor *ed*.

In UNIX operating system, we can use the editor *ed* to search for a specific pattern in the text. For example, if the command specified is

$$/acb^*c/$$

then the editor searches for a string which starts with *ac* followed by zero or more b's and followed by the symbol *c*. Note that the editor *ed* accepts the regular expression and searches for that particular pattern in the text. As the input can vary dynamically, it is challenging to write programs for string patterns of these kinds.



## Properties of regular languages

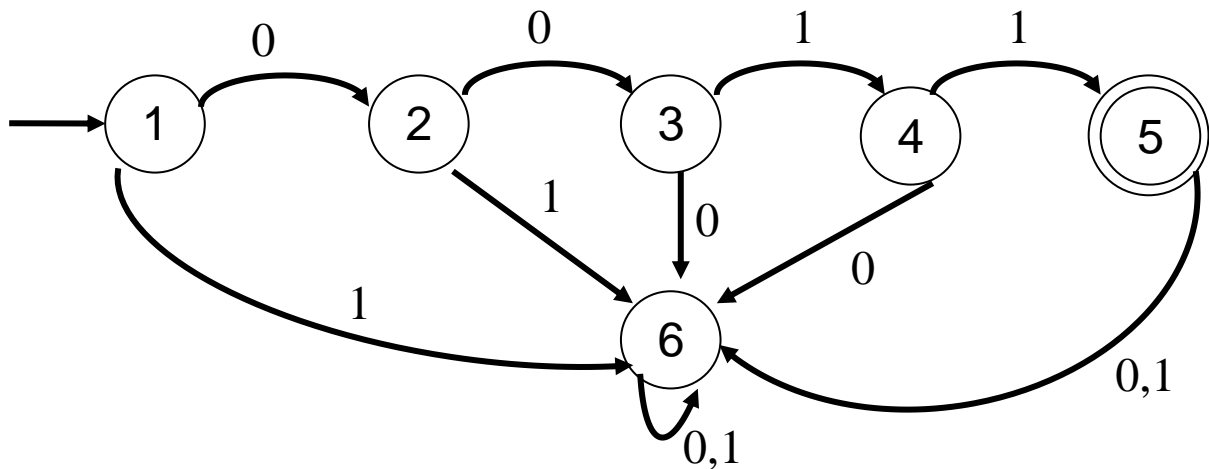
- Pumping Lemma  
Used to prove certain languages like  $L = \{0^n 1^n \mid n \geq 1\}$  are not regular.
- Closure properties of regular languages  
Used to build recognizers for languages that are constructed from other languages by certain operations.  
Ex. Automata for intersection of two regular languages
- Decision properties of regular languages
  - Used to find whether two automata define the same language
  - Used to minimize the states of DFA  
eg. Design of switching circuits.

### Pumping Lemma for regular languages ( Explanation)

Let  $L = \{0^n 1^n \mid n \geq 1\}$

There is no regular expression to define L.  $00^*11^*$  is not the regular expression defining L.

Let  $L = \{0^2 1^2\}$



State 6 is a trap state, state 3 remembers that two 0's have come and from there state 5 remembers that two 1's are accepted.

This implies DFA has no memory to remember arbitrary 'n'. In other words if we have to remember n, which varies from 1 to  $\infty$ , we have to have infinite states, which is not possible with a finite state machine, which has finite number of states.

## Pumping Lemma (PL) for Regular Languages

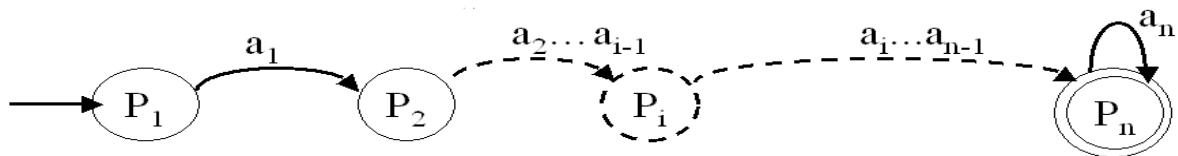
### Theorem:

Let  $L$  be a regular language. Then there exists a constant ' $n$ ' (which depends on  $L$ ) such that for every string  $w$  in  $L$  such that  $|w| \geq n$ , we can break  $w$  into three strings,  $w=xyz$ , such that:

1.  $|y| > 0$
2.  $|xy| \leq n$
3. For all  $k \geq 0$ , the string  $xy^kz$  is also in  $L$ .

### PROOF:

Let  $L$  be regular defined by an FA having ' $n$ ' states. Let  $w = a_1, a_2, a_3, \dots, a_n$  and is in  $L$ .  
 $|w| = n \geq n$ . Let the start state be  $P_1$ . Let  $w = xyz$  where  $x = a_1, a_2, a_3, \dots, a_{n-1}$ ,  $y = a_n$  and  $z = \epsilon$ .



$$\left. \begin{array}{l} \delta(P_1, a_1) = P_2 \\ \delta(P_2, a_2) = P_3 \\ \vdots \\ \delta(P_n, a_n) = P_{n+1} \end{array} \right\} \begin{array}{l} \text{But there are only } n \text{ states. } \Rightarrow \text{ there} \\ \text{must be a loop. Let there be a loop in} \\ P_n \text{ State.} \\ \text{Let } x = a_1, \dots, a_{n-1} \\ y = a_n \\ z = \epsilon \end{array}$$

Therefore  $xy^kz = a_1 \dots a_{n-1} (a_n)^k \epsilon$

- $k=0$      $a_1 \dots a_{n-1}$  is accepted  
 $k=1$      $a_1 \dots a_n$  is accepted  
 $k=2$      $a_1 \dots a_{n+1}$  is accepted  
 $k=10$     $a_1 \dots a_{n+9}$  is accepted and so on.

Uses of Pumping Lemma: - This is to be used to show that, certain languages are not regular. It should never be used to show that some language is regular. If you want to show that language is regular, write separate expression, DFA or NFA.

General Method of proof: -

- (i) Select  $w$  such that  $|w| \geq n$
- (ii) Select  $y$  such that  $|y| \geq 1$
- (iii) Select  $x$  such that  $|xy| \leq n$
- (iv) Assign remaining string to  $z$
- (v) Select  $k$  suitably to show that, resulting string is not in  $L$ .

Example 1.

To prove that  $L = \{w \mid w \in a^n b^n, \text{ where } n \geq 1\}$  is not regular

Proof:

Let  $L$  be regular. Let  $n$  is the constant (PL Definition). Consider a word  $w$  in  $L$ . Let  $w = a^n b^n$ , such that  $|w| = 2n$ . Since  $2n > n$  and  $L$  is regular it must satisfy PL.

Consider  $w =$   $\overbrace{aa \dots a}^n \overbrace{bb \dots b}^n$

$\left| \begin{array}{c} \leftarrow xy \rightarrow \quad \leftarrow z \rightarrow \end{array} \right|$

$xy$  contain only  $a$ 's. (Because  $|xy| \leq n$ ).

Let  $|y| = l$ , where  $l > 0$  (Because  $|y| > 0$ ).

Then, the break up of  $x$ ,  $y$  and  $z$  can be as follows

$$w = \overbrace{a^{n-1}}^x \overbrace{a^l}^y \overbrace{b^n}^z$$

from the definition of PL,  $w = xy^kz$ , where  $k=0,1,2,\dots,\infty$ , should belong to  $L$ .

That is  $a^{n-1} (a^l)^k b^n \in L$ , for all  $k=0,1,2,\dots,\infty$

Put  $k=0$ . we get  $a^{n-1} b^n \notin L$ .

Contradiction. Hence the Language is not regular.

Example 2.

To prove that  $L = \{w \mid w \text{ is a palindrome on } \{a,b\}^*\}$  is not regular. i.e.,  $L = \{aaba, aba, abbbba, \dots\}$

Proof:

Let  $L$  be regular. Let  $n$  is the constant (PL Definition). Consider a word  $w$  in  $L$ . Let  $w = a^n b a^n$ , such that  $|w| = 2n+1$ . Since  $2n+1 > n$  and  $L$  is regular it must satisfy PL.

Consider  $w = \overbrace{aa \dots a}^n b \overbrace{aa \dots a}^n$

$\leftarrow \overbrace{xy}^{\text{---}} \rightarrow \quad \leftarrow \overbrace{z}^{\text{---}} \rightarrow$

xy contain only a's. (Because  $|xy| \leq n$ ).

Let  $|y|=l$ , where  $l > 0$  (Because  $|y| > 0$ ).

That is, the break up of x, y and z can be as follows

$$w = \overbrace{a^{n-1}}^x \overbrace{a^1}^y \overbrace{ba^n}^z$$

from **the definition** of PL  $w=xy^kz$ , where  $k=0,1,2,\dots,\infty$ , should belong to L.

That is  $a^{n-1} (a^1)^k ba^n \in L$ , for all  $k=0,1,2,\dots,\infty$ .

Put  $k=0$ . we get  $a^{n-1} b a^n \notin L$ , because, it is not a palindrome. Contradiction, hence the language is not regular

Example 3.

To prove that  $L=\{\text{all strings of 1's whose length is prime}\}$  is not regular. i.e.,  $L=\{1^2, 1^3, 1^5, 1^7, 1^{11}, \dots\}$

**Proof:** Let L be regular. Let  $w = 1^p$  where p is prime and  $|w| = p = n + 2$

Let  $y = m$ .

by PL  $xy^kz \in L$

$$\begin{aligned}
 |xy^kz| &= |xz| + |y^k| && \text{Let } k = p-m \\
 &= (p-m) + m(p-m) \\
 &= (p-m)(1+m) \text{ ----- this can not be prime} \\
 &&& \text{if } p-m \geq 2 \text{ or } 1+m \geq 2
 \end{aligned}$$

1.  $(1+m) \geq 2$  because  $m \geq 1$
2. Limiting case  $p=n+2$   
 $(p-m) \geq 2$  since  $m \leq n$

Example 4.

To prove that  $L=\{0^i \mid i \text{ is integer and } i > 0\}$  is not regular. i.e.,  $L=\{0^2, 0^4, 0^9, 0^{16}, 0^{25}, \dots\}$

**Proof:** Let L be regular. Let  $w = 0^{n^2}$  where  $|w| = n^2 \geq n$

by PL  $xy^kz \in L$ , for all  $k = 0,1,\dots$

Select  $k = 2$

$$\begin{aligned}
 |xy^2z| &= |xyz| + |y| \\
 &= n^2 + \text{Min } 1 \text{ and Max } n
 \end{aligned}$$

Therefore  $n^2 < |xy^2z| \leq n^2 + n$

$$n^2 < |xy^2z| < n^2 + n + 1 + n$$

$$n^2 < |xy^2z| < (n + 1)^2$$

adding  $1 + n$  ( Note that less than or equal to is replaced by less than sign)

Say  $n = 5$  this implies that string can have length  $> 25$  and  $< 36$  which is not of the form  $0^{i^2}$ .

Exercises for students: -

a) Show that following languages are not regular

(i)  $L = \{a^n b^m \mid n, m \geq 0 \text{ and } n < m\}$

(ii)  $L = \{a^n b^m \mid n, m \geq 0 \text{ and } n > m\}$

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

(iv)  $L = \{a^n \mid n \text{ is a perfect square}\}$

(v)  $L = \{a^n \mid n \text{ is a perfect cube}\}$

b) Apply pumping lemma to following languages and understand why we cannot complete proof

(i)  $L = \{a^n aba \mid n \geq 0\}$

(ii)  $L = \{a^n b^m \mid n, m \geq 0\}$

## **Closure Properties of Regular Languages**

1. The union of two regular languages is regular.
2. The intersection of two regular languages is regular.
3. The complement of a regular language is regular.
4. The difference of two regular languages is regular.
5. The reversal of a regular language is regular.
6. The closure (star) of a regular language is regular.
7. The concatenation of regular languages is regular.
8. A homomorphism (substitution of strings for symbols) of a regular language is regular.
9. The inverse homomorphism of a regular language is regular

### **Closure under Union**

**Theorem:** If L and M are regular languages, then so is  $L \cup M$ .

Ex1.

$$L1 = \{a, a^3, a^5, \dots\}$$

$$L2 = \{a^2, a^4, a^6, \dots\}$$

$$L1 \cup L2 = \{a, a^2, a^3, a^4, \dots\}$$

$$RE = a(a)^*$$

Ex2.

$$L1 = \{ab, a^2b^2, a^3b^3, a^4b^4, \dots\}$$

$$L2 = \{ab, a^3b^3, a^5b^5, \dots\}$$

$$L1 \cup L2 = \{ab, a^2b^2, a^3b^3, a^4b^4, a^5b^5, \dots\}$$

$$RE = ab(ab)^*$$

### **Closure Under Complementation**

**Theorem :** If L is a regular language over alphabet S, then  $L^c = \Sigma^* - L$  is also a regular language.

Ex1.

$$L1 = \{a, a^3, a^5, \dots\}$$

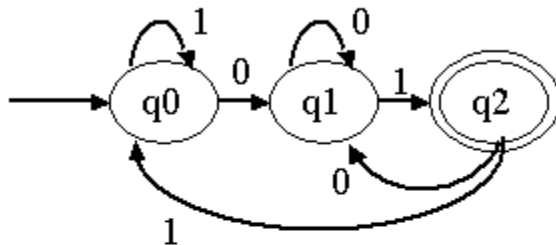
$$\Sigma^* - L1 = \{e, a^2, a^4, a^6, \dots\}$$

$$RE = (aa)^*$$

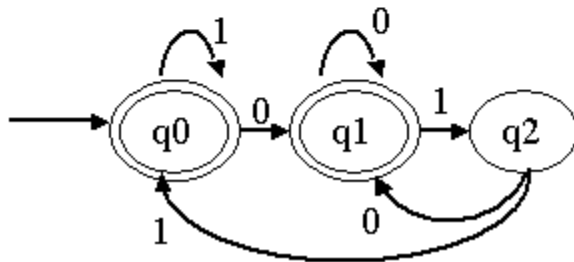
Ex2.

Consider a DFA, A that accepts all and only the strings of 0's and 1's that end in 01. That is  $L(A) = (0+1)^*01$ . The complement of  $L(A)$  is therefore all string of 0's and 1's that do not end in 01

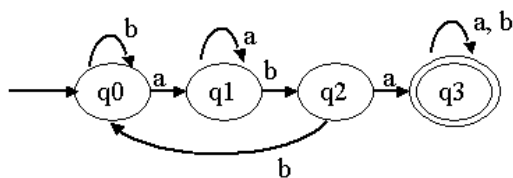
$$L(A) = (0+1)^*01$$



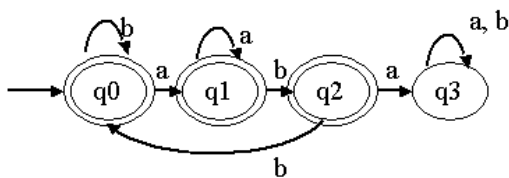
$$\overline{L(A)} = \{0,1\}^* - L(A)$$



$$L(A) = (a+b)^*aba(a+b)^*$$



$$\overline{L(A)} = \{a,b,c\}^* - L(A)$$



**Theorem:** - If  $L$  is a regular language over alphabet  $\Sigma$ , then,  $\overline{L} = \Sigma^* - L$  is also a regular language

**Proof:** - Let  $L = L(A)$  for some DFA.  $A = (Q, \Sigma, \delta, q_0, F)$ . Then  $\overline{L} = L(B)$ , where  $B$  is the DFA  $(Q, \Sigma, \delta, q_0, Q-F)$ . That is,  $B$  is exactly like  $A$ , but the accepting states of  $A$  have become non-accepting states of  $B$ , and vice versa, then  $w$  is in  $L(B)$  if and only if  $\delta^*(q_0, w)$  is in  $Q-F$ , which occurs if and only if  $w$  is not in  $L(A)$ .

### Closure Under Intersection

**Theorem :** If  $L$  and  $M$  are regular languages, then so is  $L \cap M$ .

Ex1.

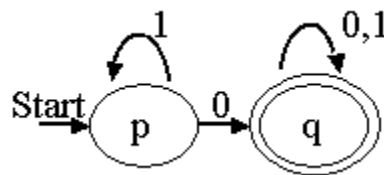
$L1 = \{a, a^2, a^3, a^4, a^5, a^6, \dots\}$   
 $L2 = \{a^2, a^4, a^6, \dots\}$   
 $L1 \cap L2 = \{a^2, a^4, a^6, \dots\}$   
 $RE = aa(aa)^*$

Ex2

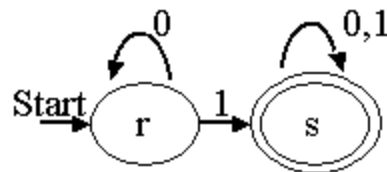
$L1 = \{ab, a^3b^3, a^5b^5, a^7b^7, \dots\}$   
 $L2 = \{a^2b^2, a^4b^4, a^6b^6, \dots\}$   
 $L1 \cap L2 = \emptyset$   
 $RE = \emptyset$

Ex3.

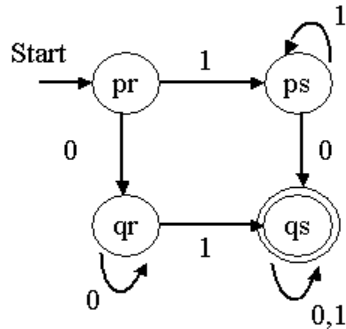
Consider a DFA that accepts all those strings that have a 0.



Consider a DFA that accepts all those strings that have a 1.



The product of above two automata is given below.

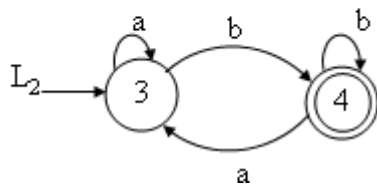
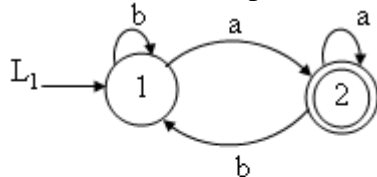




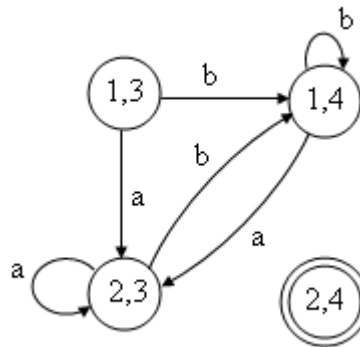
This automaton accepts the intersection of the first two languages: Those languages that have both a 0 and a 1. Then pr represents only the initial condition, in which we have seen neither 0 nor 1. Then state qr means that we have seen only once 0's, while state ps represents the condition that we have seen only 1's. The accepting state qs represents the condition where we have seen both 0's and 1's.

#### Ex 4 (on intersection)

Write a DFA to accept the intersection of  $L_1=(a+b)^*a$  and  $L_2=(a+b)^*b$  that is for  $L_1 \cap L_2$ .

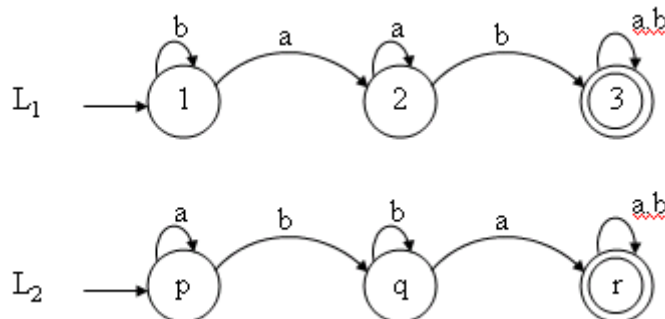


DFA for  $L_1 \cap L_2 = \emptyset$  (as no string has reached to final state (2,4))

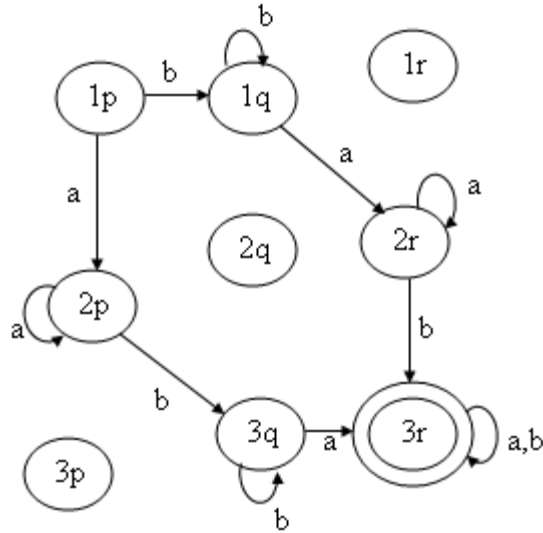


#### Ex5 (on intersection)

Find the DFA to accept the intersection of  $L_1=(a+b)^*ab(a+b)^*$  and  $L_2=(a+b)^*ba(a+b)^*$  that is for  $L_1 \cap L_2$



DFA for  $L_1 \cap L_2$



## Closure Under Difference

Theorem : If L and M are regular languages, then so is  $L - M$ .

Ex.

$$L1 = \{a, a^3, a^5, a^7, \dots\}$$

$$L2 = \{a^2, a^4, a^6, \dots\}$$

$$L1 - L2 = \{a, a^3, a^5, a^7, \dots\}$$

$$RE = a(a)^*$$

## Reversal

Theorem : If L is a regular language, so is  $L^R$

Ex.

$$L = \{001, 10, 111, 01\}$$

$$L^R = \{100, 01, 111, 10\}$$

To prove that regular languages are closed under reversal.

Let  $L = \{001, 10, 111\}$ , be a language over  $\Sigma = \{0, 1\}$ .

$L^R$  is a language consisting of the reversals of the strings of L.

That is  $L^R = \{100, 01, 111\}$ .

If L is regular we can show that  $L^R$  is also regular.

Proof.

As L is regular it can be defined by an FA,  $M = (Q, \Sigma, \delta, q_0, F)$ , having only one final state.

If there are more than one final states, we can use  $\epsilon$ - transitions from the final states going to a common final state.

Let FA,  $M^R = (Q^R, \Sigma^R, \delta^R, q_0^R, F^R)$  defines the language  $L^R$ ,

Where  $Q^R = Q$ ,  $\Sigma^R = \Sigma$ ,  $q_0^R = F$ ,  $F^R = q_0$ , and  $\delta^R(p, a) \rightarrow q$ , iff  $\delta(q, a) \rightarrow p$

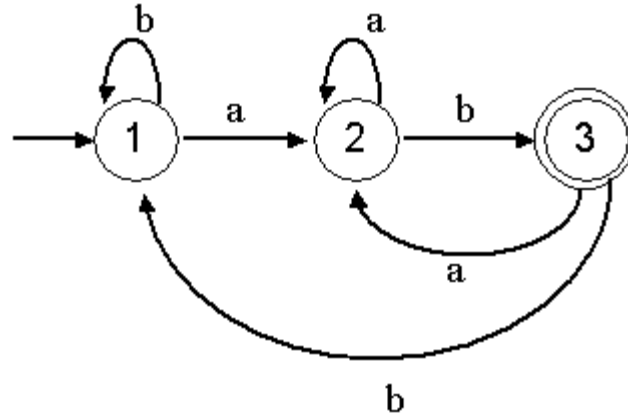
Since  $M^R$  is derivable from  $M$ ,  $L^R$  is also regular.

The proof implies the following method

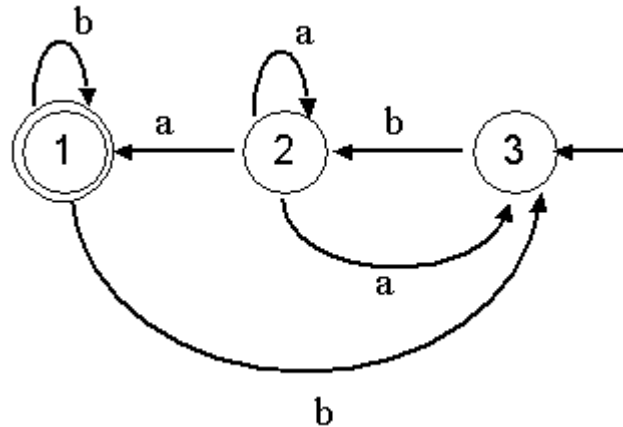
1. Reverse all the transitions.
2. Swap initial and final states.
3. Create a new start state  $p_0$  with transition on  $\epsilon$  to all the accepting states of original DFA

Example

Let  $r = (a+b)^* ab$  define a language  $L$ . That is  
 $L = \{ab, aab, bab, aaab, \dots\}$ . The FA is as given below



The FA for  $L^R$  can be derived from FA for  $L$  by swapping initial and final states and changing the direction of each edge. It is shown in the following figure.



## Homomorphism

A string homomorphism is a function on strings that works by substituting a particular string for each symbol.

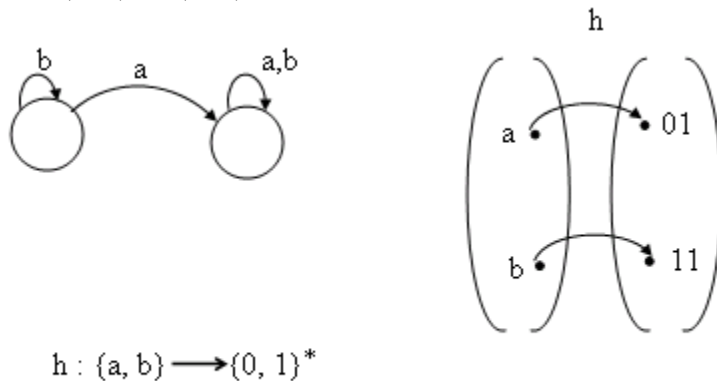
Theorem : If  $L$  is a regular language over alphabet  $\Sigma$ , and  $h$  is a homomorphism on  $\Sigma$ , then  $h(L)$  is also regular.

Ex.

The function  $h$  defined by  $h(0)=ab$   $h(1)=c$  is a homomorphism.

$h$  applied to the string 00110 is ababccab

$$L1 = (a+b)^* a (a+b)^*$$



$$h : \{a, b\} \longrightarrow \{0, 1\}^*$$

Ex.

Ex.

Ex.

$$h_1(a) = 01$$

$$h_2(a) = 101$$

$$h_2(a) = 01$$

$$h_1(b) = 11$$

$$h_2(b) = 010$$

$$h_3(a) = 101$$

Resulting :

$$h_1(L) = (01 + 11)^* 01 (01 + 11)^*$$

$$h_2(L) = (101 + 010)^* 101 (101 + 010)^*$$

$$h_3(L) = (01 + 101)^* 01 (01 + 101)^*$$

## Inverse Homomorphism

Theorem : If  $h$  is a homomorphism from alphabet  $S$  to alphabet  $T$ , and  $L$  is a regular language over  $T$ , then  $h^{-1}(L)$  is also a regular language.

Ex. Let  $L$  be the language of regular expression  $(00+1)^*$ .

Let  $h$  be the homomorphism defined by  $h(a)=01$  and  $h(b)=10$ . Then  $h^{-1}(L)$  is the language of regular expression  $(ba)^*$ .

### Decision Properties of Regular Languages

1. Is the language described empty?
2. Is a particular string  $w$  in the described language?
3. Do two descriptions of a language actually describe the same language?

This question is often called “equivalence” of languages.

### Converting Among Representations

#### **Converting NFA's to DFA's**

Time taken for either an NFA or  $\epsilon$ -NFA to DFA can be exponential in the number of states of the NFA. Computing  $\epsilon$ -Closure of  $n$  states takes  $O(n^3)$  time. Computation of DFA takes  $O(n^3)$  time where number of states of DFA can be  $2^n$ . The running time of NFA to DFA conversion including  $\epsilon$  transition is  $O(n^3 2^n)$ . Therefore the bound on the running time is  $O(n^3 s)$  where  $s$  is the number of states the DFA actually has.

#### **DFA to NFA Conversion**

Conversion takes  $O(n)$  time for an  $n$  state DFA.

#### **Automaton to Regular Expression Conversion**

For DFA where  $n$  is the number of states, conversion takes  $O(n^3 4^n)$  by substitution method and by state elimination method conversion takes  $O(n^3)$  time. If we convert an NFA to DFA and then convert the DFA to a regular expression it takes the time  $O(n^3 4^n 2^n)$ .

#### **Regular Expression to Automaton Conversion**

Regular expression to  $\epsilon$ -NFA takes linear time –  $O(n)$  on a regular expression of length  $n$ . Conversion from  $\epsilon$ -NFA to NFA takes  $O(n^3)$  time.

#### **Testing Emptiness of Regular Languages**

Suppose  $R$  is regular expression, then

1.  $R = R_1 + R_2$ . Then  $L(R)$  is empty if and only if both  $L(R_1)$  and  $L(R_2)$  are empty.
2.  $R = R_1 R_2$ . Then  $L(R)$  is empty if and only if either  $L(R_1)$  or  $L(R_2)$  is empty.
3.  $R = R_1^*$ . Then  $L(R)$  is not empty. It always includes at least  $\epsilon$ .
4.  $R = (R_1)$ . Then  $L(R)$  is empty if and only if  $L(R_1)$  is empty since they are the same language.

### **Testing Emptiness of Regular Languages**

Suppose  $R$  is regular expression, then

1.  $R = R_1 + R_2$ . Then  $L(R)$  is empty if and only if both  $L(R_1)$  and  $L(R_2)$  are empty.

2.  $R = R_1 R_2$ . Then  $L(R)$  is empty if and only if either  $L(R_1)$  or  $L(R_2)$  is empty.
3.  $R = (R_1)^*$ . Then  $L(R)$  is not empty. It always includes at least  $\epsilon$ .
4.  $R = (R_1)$ . Then  $L(R)$  is empty if and only if  $L(R_1)$  is empty since they are the same language.

### **Testing Membership in a Regular Language**

Given a string  $w$  and a Regular Language  $L$ , is  $w$  in  $L$ .

If  $L$  is represented by a DFA, simulate the DFA processing the string of input symbol  $w$ , beginning in start state. If DFA ends in accepting state the answer is 'Yes', else it is 'no'. This test takes  $O(n)$  time.

If the representation is NFA, if  $w$  is of length  $n$ , NFA has  $s$  states, running time of this algorithm is  $O(ns^2)$ .

If the representation is  $\epsilon$ -NFA,  $\epsilon$ -closure has to be computed, then processing of each input symbol  $a$ , has 2 stages, each of which requires  $O(s^2)$  time.

If the representation of  $L$  is a Regular Expression of size  $s$ , we can convert to an  $\epsilon$ -NFA with almost  $2s$  states, in  $O(s)$  time. Simulation of the above takes  $O(ns^2)$  time on an input  $w$  of length  $n$ .

### **Minimization of Automata ( Method 1)**

Let  $p$  and  $q$  are two states in DFA. Our goal is to understand when  $p$  and  $q$  ( $p \neq q$ ) can be replaced by a single state.

Two states  $p$  and  $q$  are said to be distinguishable, if there is at least one string,  $w$ , such that one of  $\delta^*(p, w)$  and  $\delta^*(q, w)$  is accepting and the other is not accepting.

#### **Algorithm 1:**

List all unordered pair of states  $(p, q)$  for which  $p \neq q$ . Make a sequence of passes through these pairs. On first pass, mark each pair of which exactly one element is in  $F$ . On each subsequent pass, mark any pair  $(r, s)$  if there is an  $a \in \Sigma$  for which  $\delta(r, a) = p$ ,  $\delta(s, a) = q$ , and  $(p, q)$  is already marked. After a pass in which no new pairs are marked, stop. The marked pair  $(p, q)$  are distinguishable.

#### **Examples:**

1. Let  $L = \{ \epsilon, a^2, a^4, a^6, \dots \}$  be a regular language over  $\Sigma = \{a, b\}$ . The FA is shown in Fig 1.

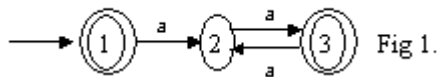
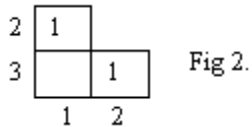


Fig 2. gives the list of all unordered pairs of states  $(p, q)$  with  $p \neq q$ .



The boxes (1,2) and (2,3) are marked in the first pass according to the algorithm 1.

In pass 2 no boxes are marked because,  $\delta(1,a) \rightarrow \phi$  and  $\delta(3,a) \rightarrow 2$ . That is  $(1,3) \xrightarrow{a} (\phi, 2)$ , where  $\phi$  and 3 are non final states.

$\delta(1,b) \rightarrow \phi$  and  $\delta(3,b) \rightarrow \phi$ . That is  $(1,3) \xrightarrow{b} (\phi, \phi)$ , where  $\phi$  is a non-final state. This implies that (1,3) are equivalent and can be replaced by a single state A.

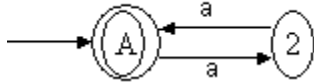
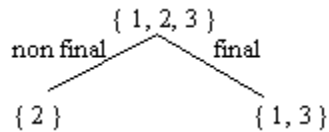


Fig 3. Minimal Automata corresponding to FA in Fig 1

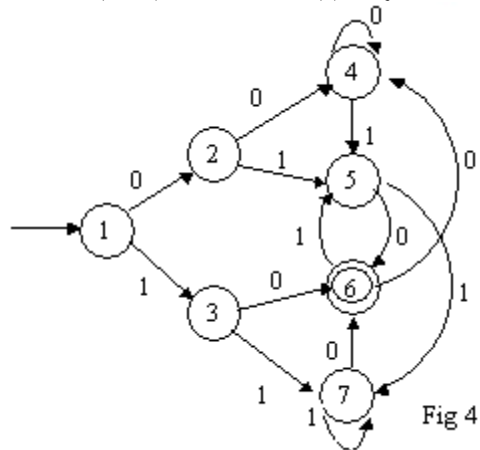
### Minimization of Automata (Method 2)



Consider set {1,3}.  $(1,3) \xrightarrow{a} (2,2)$  and  $(1,3) \xrightarrow{b} (\phi, \phi)$ . This implies state 1 and 3 are equivalent and can not be divided further. This gives us two states 2,A. The resultant FA is shown in Fig 3.

### Example 2. (Method1):

Let  $r = (0+1)^*10$ , then  $L(r) = \{10, 010, 00010, 110, \dots\}$ . The FA is given below



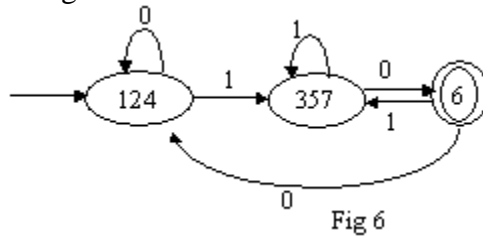
Following fig shows all unordered pairs (p,q) with  $p \neq q$

2						
3	2	2				
4			2			
5	2	2		2		
6	1	1	1	1	1	
7	2	2		2		1
	1	2	3	4	5	6

In pass 2,  
 $(3,1) \xrightarrow{0} (6,2)$   
 $(3,2) \xrightarrow{0} (6,4)$   
 $(5,1) \xrightarrow{0} (6,2)$   
 $(5,2) \xrightarrow{0} (6,4)$   
 $(7,4) \xrightarrow{0} (6,4)$   
 $(7,2) \xrightarrow{0} (6,4)$   
 and so on.

The pairs marked 1 are those of which exactly one element is in F; They are marked on pass 1. The pairs marked 2 are those marked on the second pass. For example (5,2) is one of these, since  $(5,2) \rightarrow (6,4)$ , and the pair (6,4) was marked on pass 1.

From this we can make out that 1, 2, and 4 can be replaced by a single state 124 and states 3, 5, and 7 can be replaced by the single state 357. The resultant minimal FA is shown in Fig. 6



The transitions of fig 4 are mapped to fig 6 as shown below

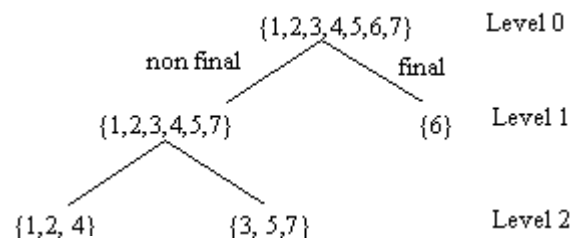
Original DFA

Minimal DFA

1	$\xrightarrow{0}$	2	124	$\xrightarrow{0}$	124
2	$\xrightarrow{1}$	5	124	$\xrightarrow{1}$	357
3	$\xrightarrow{0}$	6	357	$\xrightarrow{0}$	6

and so on.

### Example 2. (Method1):



$(2,3) \xrightarrow{0} (4,6)$  this implies that 2 and 3 belongs to different group hence they are split in level 2. similarly it can be easily shown for the pairs (4,5) (1,7) and (2,5) and so on.



## Session 13 by Prof.B.I.Khodanpur: 21.4.05

### Lecture Notes for Class 4 (21/4/2005)

#### Context Free Grammar

Context Free grammar or CGF, G is represented by four components that is  $G = (V, T, P, S)$ , where V is the set of variables, T the terminals, P the set of productions and S the start symbol.

Example: The grammar  $G_{\text{pal}}$  for palindromes is represented by

$$G_{\text{pal}} = (\{P\}, \{0, 1\}, A, P)$$

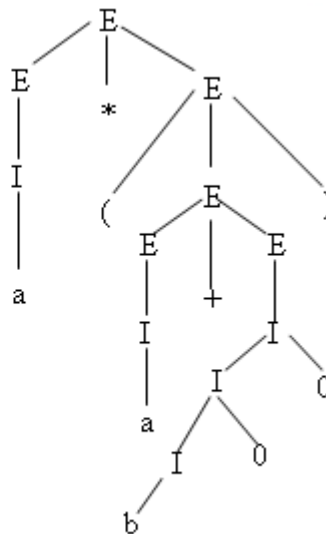
Where A represents the set of five productions

1.  $P \rightarrow \epsilon$
2.  $P \rightarrow 0$
3.  $P \rightarrow 1$
4.  $P \rightarrow 0P0$
5.  $P \rightarrow 1P1$

#### Derivation using Grammar

Consider a context-free grammar for simple expressions

1.  $E \rightarrow I$
2.  $E \rightarrow E + E$
3.  $E \rightarrow E * E$
4.  $E \rightarrow (E)$
5.  $I \rightarrow a$
6.  $I \rightarrow b$
7.  $I \rightarrow Ia$
8.  $I \rightarrow Ib$
9.  $I \rightarrow IO$
10.  $I \rightarrow II$

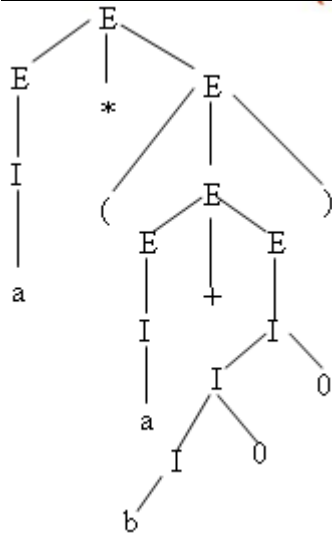


#### Example 1: Leftmost Derivation

The inference that  $a * (a + b00)$  is in the language of variable E can be reflected in a derivation of that string, starting with the string E. Here is one such derivation:

$$\begin{aligned} E &\rightarrow E * E \rightarrow I * E \rightarrow a * E \rightarrow \\ a * (E) &\rightarrow a * (E + E) \rightarrow a * (I + E) \rightarrow a * (a + E) \rightarrow \\ a * (a + I) &\rightarrow a * (a + IO) \rightarrow a * (a + I00) \rightarrow a * (a + b00) \end{aligned}$$

### Leftmost Derivation - Tree



### Example 2: Rightmost Derivations

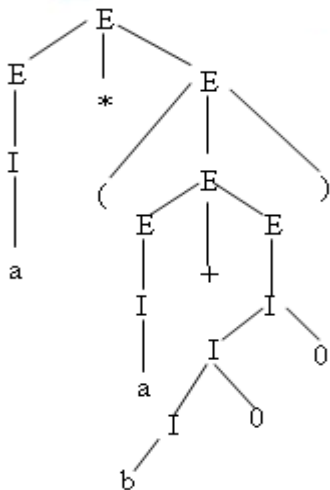
The derivation of Example 1 was actually a leftmost derivation. Thus, we can describe the same derivation by:

$$\begin{aligned}
 E &\Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E) \Rightarrow \\
 E * (E + I) &\Rightarrow E * (E + I0) \Rightarrow E * (E + I00) \Rightarrow E * (E + b00) \Rightarrow \\
 E * (I + b00) &\Rightarrow E * (a + b00) \Rightarrow I * (a + b00) \Rightarrow a * (a + b00)
 \end{aligned}$$

We can also summarize the leftmost derivation by saying

$E \Rightarrow a * (a + b00)$ , or express several steps of the derivation by expressions such as  $E * E \Rightarrow a * (E)$ .

### Rightmost Derivation - Tree



There is a rightmost derivation that uses the same replacements for each variable, although it makes the replacements in different order. This rightmost derivation is:

$$\begin{aligned}
E &\rightarrow E * E \rightarrow E * (E) \rightarrow E * (E + E) \rightarrow \\
E * (E + I) &\rightarrow E * (E + I0) \rightarrow E * (E + I00) \rightarrow E * (E + b00) \rightarrow \\
E * (I + b00) &\rightarrow E * (a + b00) \rightarrow I * (a + b00) \rightarrow a * (a + b00)
\end{aligned}$$

This derivation allows us to conclude  $E \rightarrow a * (a + b00)$

Consider the Grammar for string  $(a+b)^*c$

$$\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow ( E ) \mid a \mid b \mid c
\end{aligned}$$

Leftmost Derivation

$$\begin{aligned}
E &\rightarrow T \rightarrow T * F \rightarrow F * F \rightarrow (E) * F \rightarrow (E + T) * F \rightarrow (T + T) * F \rightarrow (F + T) * F \rightarrow (a + T) * F \rightarrow (a + F) * F \\
&\rightarrow (a + b) * F \rightarrow (a + b) * c
\end{aligned}$$

Rightmost derivation

$$\begin{aligned}
E &\rightarrow T \rightarrow T * F \rightarrow T * c \rightarrow F * c \rightarrow (E) * c \rightarrow (E + T) * c \rightarrow (E + F) * c \\
&\rightarrow (E + b) * c \rightarrow (T + b) * c \rightarrow (F + b) * c \rightarrow (a + b) * c
\end{aligned}$$

### Example 2:

Consider the Grammar for string  $(a,a)$

$$\begin{aligned}
S &\rightarrow (L) \mid a \\
L &\rightarrow L, S \mid S
\end{aligned}$$

Leftmost derivation

$$S \rightarrow (L) \rightarrow (L, S) \rightarrow (S, S) \rightarrow (a, S) \rightarrow (a, a)$$

Rightmost Derivation

$$S \rightarrow (L) \rightarrow (L, S) \rightarrow (L, a) \rightarrow (S, a) \rightarrow (a, a)$$

### The Language of a Grammar

If  $G(V, T, P, S)$  is a CFG, the language of  $G$ , denoted by  $L(G)$ , is the set of terminal strings that have derivations from the start symbol.

$$L(G) = \{ w \text{ in } T^* \mid S \rightarrow w \}$$

### Sentential Forms

Derivations from the start symbol produce strings that have a special role called “sentential forms”. That is if  $G = (V, T, P, S)$  is a CFG, then any string in  $(V \cup T)^*$  such that  $S \rightarrow \alpha$  is a sentential form. If  $S \rightarrow \alpha$ , then  $\alpha$  is a left – sentential form, and if  $S \rightarrow \alpha$ , then  $\alpha$  is a right – sentential form. Note that the language  $L(G)$  is those sentential forms that are in  $T^*$ ; that is they consist solely of terminals.

For example,  $E * (I + E)$  is a sentential form, since there is a derivation

$$E \rightarrow E * E \rightarrow E * (E) \rightarrow E * (E + E) \rightarrow E * (I + E)$$

However this derivation is neither leftmost nor rightmost, since at the last step, the middle E is replaced.

As an example of a left – sentential form, consider  $a * E$ , with the leftmost derivation.

$E \rightarrow E * E \rightarrow I * E \rightarrow a * E$

Additionally, the derivation

$E \rightarrow E * E \rightarrow E * (E) \rightarrow E * (E + E)$

Shows that

$E * (E + E)$  is a right – sentential form.

### Ambiguity

A context – free grammar  $G$  is said to be ambiguous if there exists some  $w \in L(G)$  which has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more left most or rightmost derivations.

### Ex:-

Consider the grammar  $G=(V,T,E,P)$  with  $V=\{E,I\}$ ,  $T=\{a,b,c,+,*,(,)\}$ , and productions.

$E \rightarrow I$ ,

$E \rightarrow E+E$ ,

$E \rightarrow E * E$ ,

$E \rightarrow (E)$ ,

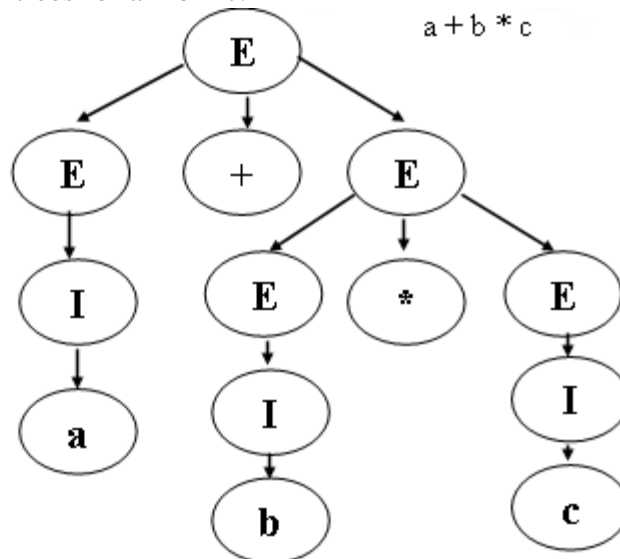
$I \rightarrow a|b|c$

Consider two derivation trees for  $a + b * c$ .

### **Tree I**

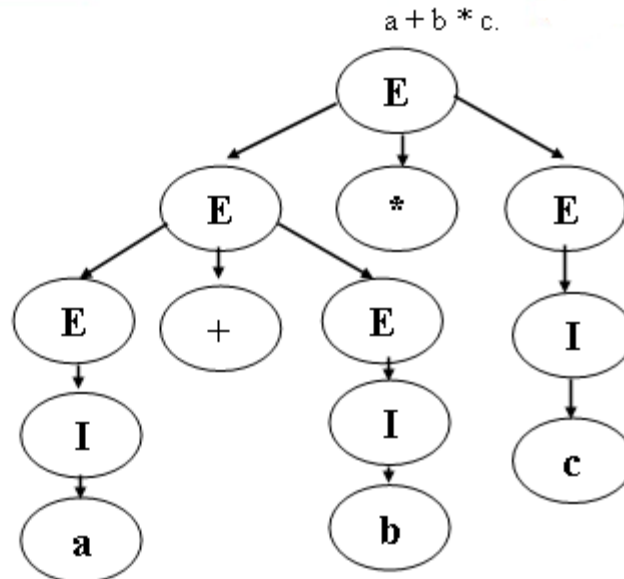
Let  $a=5$ ,  $b=6$ ,  $c=7$

The value for Tree I will be 47



## Tree II

Let  $a=5$ ,  $b=6$ ,  $c=7$   
The value for Tree II  
will be 77



Now unambiguous grammar for the above

Example:

$E \rightarrow T, T \rightarrow F, F \rightarrow I, E \rightarrow E + T, T \rightarrow T * F,$

$F \rightarrow (E), I \rightarrow a|b|c$

### Inherent Ambiguity

A CFL L is said to be inherently ambiguous if all its grammars are ambiguous

Example:

Consider the Grammar for string aabbccdd

$S \rightarrow AB | C$

$A \rightarrow aAb | ab$

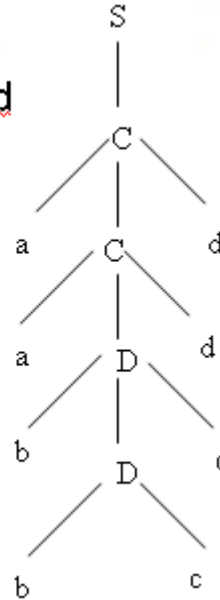
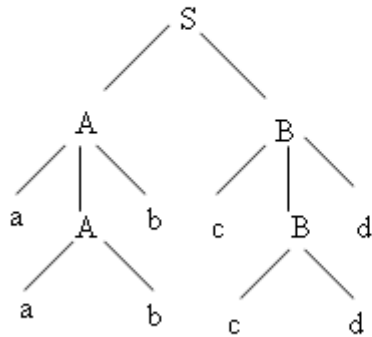
$B \rightarrow cBd | cd$

$C \rightarrow aCd | aDd$

$D \rightarrow bDc | bc$

Parse tree for string aabbccdd

## Parse tree for string aabbccdd



### Applications of Context – Free Grammars

- Parsers
- The YACC Parser Generator
- Markup Languages
- XML and Document type definitions

### The YACC Parser Generator

$E \rightarrow I \mid E+E \mid E * E \mid (E)$

$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

Exp : Id {...}  
      | Exp '+' Exp {...}  
      | Exp '\*' Exp {...}  
      | '(' Exp ')' {...}  
      ;

Id : 'a' {...}  
      | 'b' {...}  
      | Id 'a' {...}  
      | Id 'b' {...}  
      | Id '0' {...}  
      | Id '1' {...}  
      ;

### XML and Document type definitions.

1.  $A \rightarrow E_1 E_2$ .

$A \rightarrow BC$

$B \rightarrow E_1$

$C \rightarrow E_2$

2.  $A \rightarrow E_1 \mid E_2$ .

$A \rightarrow E_1$

$A \rightarrow E_2$

3.  $A \rightarrow (E_1)^*$

$A \rightarrow BA$

$A \rightarrow \varepsilon$

$B \rightarrow E_1$

4.  $A \rightarrow (E_1)^+$

$A \rightarrow BA$

$A \rightarrow B$

$B \rightarrow E_1$

5.  $A \rightarrow (E_1)?$

$A \rightarrow \varepsilon$

$A \rightarrow E_1$

### EXERCISE QUESTIONS

1) Design context-free grammar for the following cases

a)  $L = \{ 0^n 1^n \mid n \geq 1 \}$

b)  $L = \{ a^i b^j c^k \mid i \neq j \text{ or } j \neq k \}$

2) The following grammar generates the language of RE

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

$S \rightarrow A|B$

$A \rightarrow 0A|\varepsilon$

$B \rightarrow 0B|1B|\varepsilon$

Give leftmost and rightmost derivations of the following strings

a) 00101    b) 1001    c) 00011

- 3) Consider the grammar

$$S \rightarrow aS \mid aSbS \mid \epsilon$$

Show that deviation for the string  $aab$  is ambiguous

- 4) Suppose  $h$  is the homomorphism from the alphabet  $\{0,1,2\}$  to the alphabet  $\{a,b\}$  defined by  $h(0) = a$ ;  $h(1) = ab$  &  $h(2) = ba$

- a) What is  $h(0120)$  ?
- b) What is  $h(21120)$  ?
- c) If  $L$  is the language  $L(01^*2)$ , what is  $h(L)$  ?
- d) If  $L$  is the language  $L(0+12)$ , what is  $h(L)$  ?
- e) If  $L$  is the language  $L(a(ba)^*)$ , what is  $h^{-1}(L)$  ?



## 27.0.05

### Context Free Grammar

Context Free grammar or CGF,  $G$  is represented by four components that is  $G=(V,T,P,S)$ , where  $V$  is the set of variables,  $T$  the terminals,  $P$  the set of productions and  $S$  the start symbol.

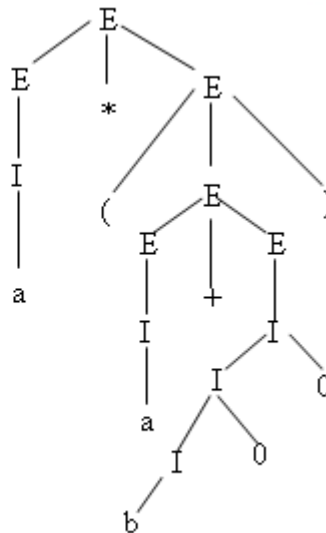
Example: The grammar  $G_{\text{pal}}$  for palindromes is represented by  
 $G_{\text{pal}} = (\{P\}, \{0,1\}, A, P)$   
where  $A$  represents the set of five productions

1.  $P \rightarrow \epsilon$
2.  $P \rightarrow 0$
3.  $P \rightarrow 1$
4.  $P \rightarrow 0P0$
5.  $P \rightarrow 1P1$

### Derivation using Grammar

Consider a context-free grammar for simple expressions

1.  $E \rightarrow I$
2.  $E \rightarrow E + E$
3.  $E \rightarrow E * E$
4.  $E \rightarrow (E)$
5.  $I \rightarrow a$
6.  $I \rightarrow b$
7.  $I \rightarrow Ia$
8.  $I \rightarrow Ib$
9.  $I \rightarrow IO$
10.  $I \rightarrow II$



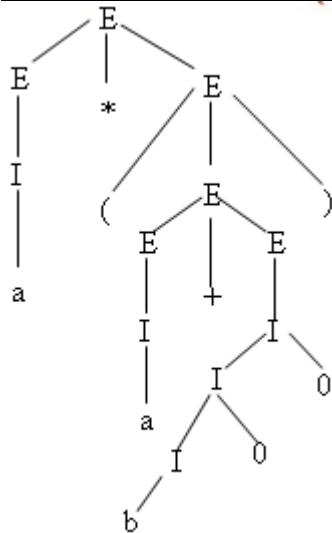
### Example 1: Leftmost Derivation

The inference that  $a * (a+b00)$  is in the language of variable  $E$  can be reflected in a derivation of that string, starting with the string  $E$ . Here is one such derivation:

$$\begin{aligned} E &\Rightarrow E * E \Rightarrow I * E \Rightarrow a * E \Rightarrow \\ a * (E) &\Rightarrow a * (E + E) \Rightarrow a * (I + E) \Rightarrow a * (a + E) \Rightarrow \end{aligned}$$

$$a * (a + I) \rightarrow a * (a + I0) \rightarrow a * (a + I00) \rightarrow a * (a + b00)$$

### Leftmost Derivation - Tree



### Example 2: Rightmost Derivations

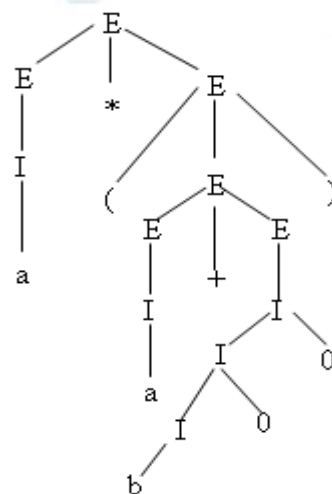
The derivation of Example 1 was actually a leftmost derivation. Thus, we can describe the same derivation by:

$$\begin{aligned} E &\rightarrow E * E \rightarrow E * (E) \rightarrow E * (E + E) \rightarrow \\ E * (E + I) &\rightarrow E * (E + I0) \rightarrow E * (E + I00) \rightarrow E * (E + b00) \rightarrow \\ E * (I + b00) &\rightarrow E * (a + b00) \rightarrow I * (a + b00) \rightarrow a * (a + b00) \end{aligned}$$

We can also summarize the leftmost derivation by saying

$$\begin{aligned} E &\rightarrow a * (a + b00), \text{ or express several steps of the derivation by expressions such as } E * \\ E &\rightarrow a * (E). \end{aligned}$$

### Rightmost Derivation - Tree



There is a rightmost derivation that uses the same replacements for each variable, although it makes the replacements in different order. This rightmost derivation is:

$E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E) \Rightarrow$   
 $E * (E + I) \Rightarrow E * (E + IO) \Rightarrow E * (E + IOO) \Rightarrow E * (E + bOO) \Rightarrow$   
 $E * (I + bOO) \Rightarrow E * (a + bOO) \Rightarrow I * (a + bOO) \Rightarrow a * (a + bOO)$   
 This derivation allows us to conclude  $E \Rightarrow a * (a + bOO)$

Consider the Grammar for string  $(a+b)^*c$

$E \rightarrow E + T \mid T$   
 $T \rightarrow T * F \mid F$   
 $F \rightarrow ( E ) \mid a \mid b \mid c$

Leftmost Derivation

$E \rightarrow T \Rightarrow T * F \Rightarrow F * F \Rightarrow (E) * F \Rightarrow (E+T) * F \Rightarrow (T+T) * F \Rightarrow (F+T) * F \Rightarrow (a+T) * F \Rightarrow (a+F) * F$   
 $\Rightarrow (a+b) * F \Rightarrow (a+b)^*c$

Rightmost derivation

$E \rightarrow T \Rightarrow T * F \Rightarrow T * c \Rightarrow F * c \Rightarrow (E) * c \Rightarrow (E+T) * c \Rightarrow (E+F) * c$   
 $\Rightarrow (E+b) * c \Rightarrow (T+b) * c \Rightarrow (F+b) * c \Rightarrow (a+b)^*c$

### Example 2:

Consider the Grammar for string  $(a,a)$

$S \rightarrow (L) \mid a$   
 $L \rightarrow L, S \mid S$

Leftmost derivation

$S \rightarrow (L) \rightarrow (L,S) \rightarrow (S,S) \rightarrow (a,S) \rightarrow (a,a)$

Rightmost Derivation

$S \rightarrow (L) \rightarrow (L,S) \rightarrow (L,a) \rightarrow (S,a) \rightarrow (a,a)$

### The Language of a Grammar

If  $G(V,T,P,S)$  is a CFG, the language of  $G$ , denoted by  $L(G)$ , is the set of terminal strings that have derivations from the start symbol.

$$L(G) = \{w \text{ in } T^* \mid S \Rightarrow w\}$$

### Sentential Forms

Derivations from the start symbol produce strings that have a special role called “sentential forms”. That is if  $G = (V, T, P, S)$  is a CFG, then any string in  $(V \cup T)^*$  such that  $S \Rightarrow \alpha$  is a sentential form. If  $S \Rightarrow \alpha$ , then  $\alpha$  is a left – sentential form, and if  $S \Rightarrow \alpha$ , then  $\alpha$  is a right – sentential form. Note that the language  $L(G)$  is those sentential forms that are in  $T^*$ ; that is they consist solely of terminals.

For example,  $E * (I + E)$  is a sentential form, since there is a derivation

$E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E) \Rightarrow E * (I + E)$

However this derivation is neither leftmost nor rightmost, since at the last step, the middle  $E$  is replaced.

As an example of a left – sentential form, consider  $a * E$ , with the leftmost derivation.

$E \rightarrow E * E \rightarrow I * E \rightarrow a * E$

Additionally, the derivation

$E \rightarrow E * E \rightarrow E * (E) \rightarrow E * (E + E)$

Shows that

$E * (E + E)$  is a right – sentential form.

### Ambiguity

A context – free grammar  $G$  is said to be ambiguous if there exists some  $w \in L(G)$  which has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more left most or rightmost derivations.

### Ex:-

Consider the grammar  $G=(V,T,E,P)$  with  $V=\{E,I\}$ ,  $T=\{a,b,c,+,*,(,)\}$ , and productions.

$E \rightarrow I$ ,

$E \rightarrow E + E$ ,

$E \rightarrow E * E$ ,

$E \rightarrow (E)$ ,

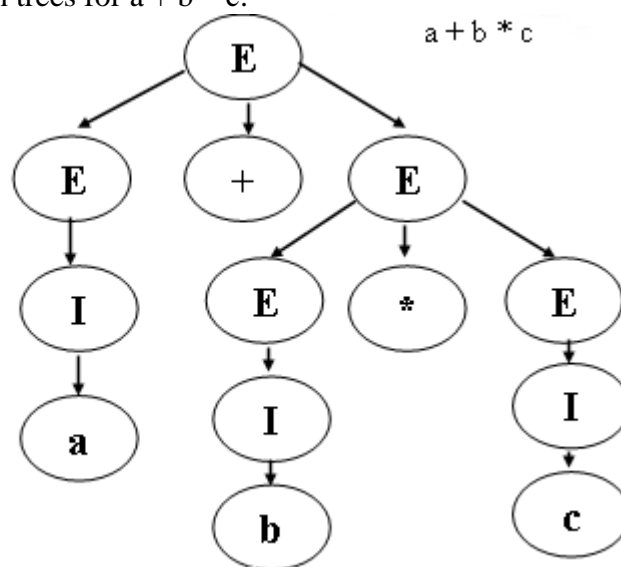
$I \rightarrow a|b|c$

Consider two derivation trees for  $a + b * c$ .

### **Tree I**

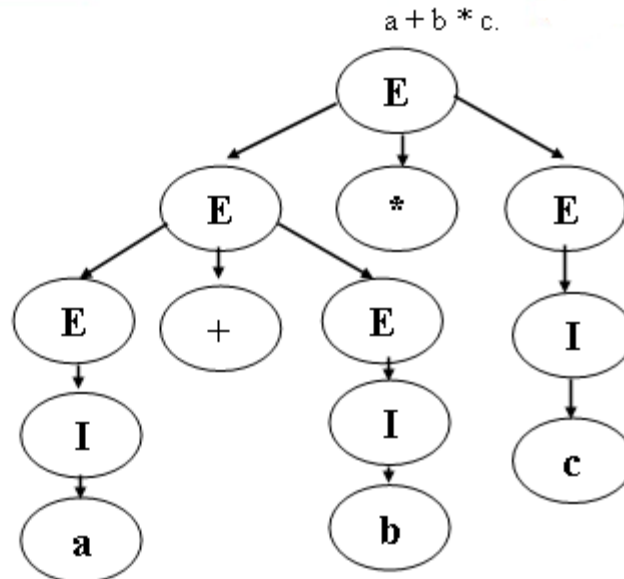
Let  $a=5$ ,  $b=6$ ,  $c=7$

The value for Tree I will be 47



## Tree II

Let  $a=5$ ,  $b=6$ ,  $c=7$   
The value for Tree II  
will be 77



Now unambiguous grammar for the above

Example:

$E \rightarrow T$ ,  $T \rightarrow F$ ,  $F \rightarrow I$ ,  $E \rightarrow E + T$ ,  $T \rightarrow T * F$ ,  
 $F \rightarrow (E)$ ,  $I \rightarrow a|b|c$

## Inherent Ambiguity

A CFL  $L$  is said to be inherently ambiguous if all its grammars are ambiguous

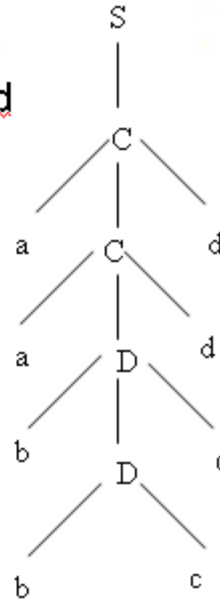
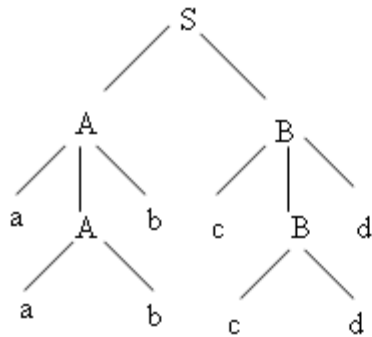
Example:

Consider the Grammar for string aabbccdd

$S \rightarrow AB \mid C$   
 $A \rightarrow aAb \mid ab$   
 $B \rightarrow cBd \mid cd$   
 $C \rightarrow aCd \mid aDd$   
 $D \rightarrow bDc \mid bc$

Parse tree for string aabbccdd

## Parse tree for string aabbccdd



## Applications of Context – Free Grammars

- Parsers
- The YACC Parser Generator
- Markup Languages
- XML and Document type definitions

## The YACC Parser Generator

$E \rightarrow E+E \mid E * E \mid (E)id$

```
%{ #include <stdio.h>
```

```
%}
```

```
%token ID id
```

```
%%
```

```
Exp : id { $$ = $1 ; printf ("result is %d\n", $1); }
```

```
      | Exp '+' Exp { $$ = $1 + $3; }
```

```
      | Exp '*' Exp { $$ = $1 * $3; }
```

```
      | '(' Exp ')' { $$ = $2; }
```

```
      ;
```

```
%%
```

```
int main (void) {
```

```
  return yyparse ( );
```

```
}
```

```
void yyerror (char *s) {
```

```
  fprintf (stderr, "%s\n", s);
```

```
}
```

```
%{
```

```
#include "y.tab.h"
```

```
% }
%%
[0-9]+      {yyval.ID = atoi(yytext); return id;}
[ \t \n]    ;
[+ * ( )]   {return yytext[0];}
.           {ECHO; yyerror ("unexpected character");}
%%
```

## Example 2:

```
% {
#include <stdio.h>
% }
%start line
%token <a_number> number
%type <a_number> exp term factor
%%
line : exp ';' {printf ("result is %d\n", $1);}
;
exp : term {$$ = $1;}
    | exp '+' term {$$ = $1 + $3;}
    | exp '-' term {$$ = $1 - $3;}
term : factor {$$ = $1;}
    | term '*' factor {$$ = $1 * $3;}
    | term '/' factor {$$ = $1 / $3;}
;
factor : number {$$ = $1;}
| '(' exp ')' {$$ = $2;}
;
%%
int main (void) {
return yyparse ( );
}
void yyerror (char *s) {
fprintf (stderr, "%s\n", s);
}
% {
#include "y.tab.h"
% }
%%
[0-9]+ {yyval.a_number = atoi(yytext); return number;}
[ \t \n] ;
[-+*/()]; {return yytext[0];}
. {ECHO; yyerror ("unexpected character");}
%%
```

## Markup Languages

# Functions

- Creating links between documents
- Describing the format of the document

## Example

The Things I *hate*

1. Moldy bread
2. People who drive too slow  
In the fast lane

HTML Source

```
<P> The things I <EM>hate</EM>:  
<OL>  
<LI> Moldy bread  
<LI>People who drive too slow  
In the fast lane  
</OL>
```

HTML Grammar

- Char           a | A | ...
  - Text           e | Char Text
  - Doc            e | Element Doc
  - Element       Text |  
                  <EM> Doc </EM>|  
                  <p> Doc |  
                  <OL> List </OL>| ...
5.     List-Item     <LI> Doc
6.     List           e | List-Item List     Start symbol

### XML and Document type definitions.

2.  $A \rightarrow E_1, E_2.$
- $A \rightarrow BC$   
 $B \rightarrow E_1$   
 $C \rightarrow E_2$
3.  $A \rightarrow E_1 \mid E_2.$
- $A \rightarrow E_1$   
 $A \rightarrow E_2$
4.  $A \rightarrow (E_1)^*$



- $A \rightarrow BA$   
 $A \rightarrow \varepsilon$   
 $B \rightarrow E1$
4.  $A \rightarrow (E1)^+$
- $A \rightarrow BA$   
 $A \rightarrow B$   
 $B \rightarrow E1$
5.  $A \rightarrow (E1)^?$
- $A \rightarrow \varepsilon$   
 $A \rightarrow E1$

## **EXERCISE QUESTIONS**

- 1) Design context-free grammar for the following cases
  - a)  $L = \{ 0^n 1^n \mid n \geq 1 \}$
  - b)  $L = \{ a^i b^j c^k \mid i \neq j \text{ or } j \neq k \}$
- 3) The following grammar generates the language of RE  $0^*1(0+1)^*$ 

$$S \rightarrow A|B$$

$$A \rightarrow 0A|\varepsilon$$

$$B \rightarrow 0B|1B|\varepsilon$$
 Give leftmost and rightmost derivations of the following strings
  - a) 00101    b) 1001    c) 00011
- 3) Consider the grammar
 
$$S \rightarrow aS|aSbS|\varepsilon$$
 Show that deviation for the string aab is ambiguous
- 4) Suppose  $h$  is the homomorphism from the alphabet  $\{0,1,2\}$  to the alphabet  $\{a,b\}$  defined by  $h(0) = a$ ;  $h(1) = ab$  &  $h(2) = ba$ 
  - a) What is  $h(0120)$  ?
  - b) What is  $h(21120)$  ?
  - c) If  $L$  is the language  $L(01^*2)$ , what is  $h(L)$  ?
  - d) If  $L$  is the language  $L(0+12)$ , what is  $h(L)$  ?
  - e) If  $L$  is the language  $L(a(ba)^*)$ , what is  $h^{-1}(L)$  ?

**Session by Ms. Jibi Abraham:**

**Introduction to Push Down Automata**

Just as finite-state automata correspond to regular languages, the context-free languages (CFL) have corresponding machines called pushdown automata (PDA).

Regular expressions are generators for regular languages and Finite Automata's are recognizers for them. Similarly for Context-free Languages, Context Free Grammars (CFG) are generators and Pushdown Automata (PDA) are recognizers.

PDA is more powerful than FA. An FA cannot recognize the language  $a^n b^n$ ,  $n \geq 0$ , because FA does not have any memory to remember the number of a's it has already seen, for equating with number of b's found. PDA is NFA with an added memory. Stack functions as the required memory. So, a PDA is an NFA with a stack. Figure 1 shows a diagrammatic representation of PDA. The Finite State Control (FSC) reads inputs, one symbol at a time. Based on the input symbol, current state and the top symbol on the stack, FSC does some state transitions and does some operations to the stack content. Stack could be kept unchanged, or some thing could be pushed into the stack and could be popped out of the stack.

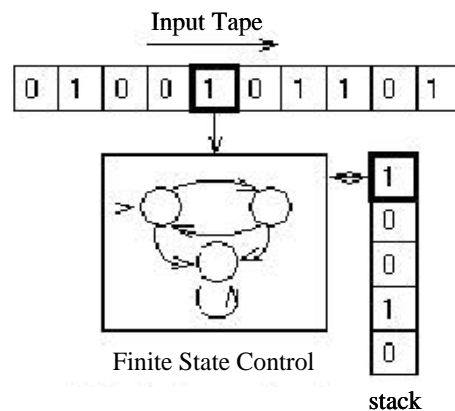


Figure 1: PushDown Automaton

**Formal Definition:**

A *nondeterministic pushdown automaton* or *npda* is a 7-tuple  $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$ , where

- $Q$  is a finite set of *states*,
- $\Sigma$  is a the *input alphabet*,
- $\Gamma$  is the *stack alphabet*,
- $\delta$  is a *transition function*, has the form  
 $\delta: Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow \text{finite subsets of } Q \times \Gamma^*$
- $q_0 \in Q$  is the *initial state*,
- $z \in \Gamma$  is the *stack start symbol*, and
- $F \subseteq Q$  is a set of *final states*.

Transition function: for any given state, input symbol and stack symbol, gives a new state and stack symbol; i.e. it has the form:  $(P, a, t) \rightarrow (Q, u)$

Basically, if,  $a \in \Sigma$ ,  $t \in \Gamma$  and  $P$  and  $Q$  are states. Means “read the symbol ‘a’ from the input, move from state  $P$  to state  $Q$ , and replace the symbol ‘t’ on top of the stack with the symbol ‘u’”.

If  $u = t$ , then stack is unchanged.

If  $u = \epsilon$ , then stack is popped

If  $u = wx$ , then  $t$  is replaced with  $x$  and  $w$  is pushed into the stack.

### Example 1:

Construct PDA for the language  $L = \{a^n b^n \mid a, b \in \Sigma, n \geq 0\}$ .

Start at state  $q_0$  and keep  $Z_0$  in the stack. The following transitions are possible:

1. If current state is  $q_0$ , and symbol on input tape is  $\epsilon$ , and stack top is  $Z_0$ , then move to  $q_2$  the final state.
2. If current state is  $q_0$ , and input tape symbol is  $a$ , and stack top  $Z_0$ , then stay in  $q_0$  and push ‘a’ to the stack.
3. If current state is  $q_0$ , input tape symbol is ‘a’, and stack top is  $a$ , stay in  $q_0$  and push ‘a’ to the stack.
4. If current state is  $q_0$ , input tape symbol is  $b$ , stack top is  $a$ , move to state  $q_1$  and pop the top symbol of the stack.
5. If current state is  $q_1$ , input tape symbol is  $b$ , stack top is  $a$ , stay in  $q_1$  and pop the top symbol of the stack
6. If current state is  $q_1$ , input tape symbol is  $\epsilon$  and stack top is  $Z_0$ , move to  $q_2$  the final state.

So we can define PDA as  $M = (\{q_0, q_1, q_2\}, \{a, b\}, \{a, b, Z_0\}, \delta, q_0, Z_0, \{q_2\})$ , where  $\delta$  is defined by following rules:

$$\delta(q_0, a, Z_0) = \{(q_0, aZ_0)\}$$

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

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

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

$$\delta(q_1, \epsilon, Z_0) = \{(q_2, \epsilon)\}$$

$$\delta(q_0, \epsilon, Z_0) = \{(q_2, \epsilon)\}$$

$$\delta(q, x, Y) = \phi \quad \text{for all other possibilities}$$

## Graphical Notation of PDA:

To understand the behavior of PDA clearer, the transition diagram of PDA can be used. Transition diagram of PDA is generalization of transition diagram of FA.

1. Node corresponds to states of PDA
2. Arrow labeled Start indicates start state
3. Doubly circled states are final states
4. Arc corresponds to transitions of PDA. If  $\delta(q, a, X) = \{(p, \alpha)\dots\}$  is an arc labeled  $(a, X/\alpha)$  from state  $q$  to state  $p$  means that an input tape head positioned at symbol  $a$  and stack top with  $X$ , moves automaton to state  $p$  and replaces the stack top with  $\alpha$ .

The transition diagram for the above example PDA is given in Figure 2.

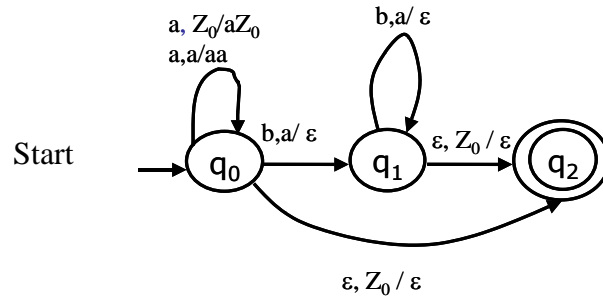


Figure 2: Transition diagram

## Instantaneous Description:

Instantaneous Description or configuration of a PDA describes its execution status at any time. Instantaneous Description is represented by a triplet  $(q, w, u)$ , where

1.  $q$  is the current state of the automaton,
2.  $w$  is the unread part of the input string,  $w \in \Sigma^*$
3.  $u$  is the stack contents, written as a string, with the leftmost symbol at the top of the stack.  
So  $u \in \Gamma^*$

## Moves of A PDA:

Let the symbol " $|$ -" indicates a move of the nPDA. There are two types of moves possible for a PDA.

### 1. Move by consuming input symbol

Suppose that  $\delta(q_1, a, x) = \{(q_2, y), \dots\}$ . Then the following move by consuming an input symbol is possible:

$$(q_1, aW, xZ) \vdash (q_2, W, yZ),$$

where  $W$  indicates the rest of the input string following the  $a$ , and  $Z$  indicates the rest of the stack contents underneath the  $x$ . This notation says that in moving from state  $q_1$  to state  $q_2$ , an input symbol ' $a$ ' is consumed from the input string  $aW$ , and the symbol ' $x$ ' at the top (left) of the stack  $xZ$  is replaced with symbol ' $y$ ', leaving  $yZ$  on the stack.

The above example PDA with a few example input strings, the moves are given below:

a) Moves for the input string  $aabb$ :

$$\begin{aligned} (q_0, aabb, Z_0) &\vdash (q_0, abb, aZ_0) \text{ as per transition rule } \delta(q_0, a, Z_0) = \{(q_0, aZ_0)\} \\ &\vdash (q_0, bb, aaZ_0) \text{ as per transition rule } \delta(q_0, a, a) = \{(q_0, aa)\} \\ &\vdash (q_1, b, aZ_0) \text{ as per transition rule } \delta(q_0, b, a) = \{(q_1, \epsilon)\} \\ &\vdash (q_1, \epsilon, Z_0) \text{ as per transition rule } \delta(q_0, b, a) = \{(q_1, \epsilon)\} \\ &\vdash (q_2, \epsilon, \epsilon) \text{ as per transition rule } \delta(q_1, \epsilon, Z_0) = \{(q_2, \epsilon)\} \end{aligned}$$

PDA reached a configuration of  $(q_2, \epsilon, \epsilon)$ . The input tape is empty, stack is empty and PDA has reached a final state. So the string is accepted.

b) Moves for the input string  $aaabb$ :

$$\begin{aligned} (q_0, aaabb, Z_0) &\vdash (q_0, aabb, aZ_0) \text{ as per transition rule } \delta(q_0, a, Z_0) = \{(q_0, aZ_0)\} \\ &\vdash (q_0, abb, aaZ_0) \text{ as per transition rule } \delta(q_0, a, a) = \{(q_0, aa)\} \\ &\vdash (q_0, bb, aaaZ_0) \text{ as per transition rule } \delta(q_0, a, a) = \{(q_0, aa)\} \\ &\vdash (q_1, b, aaZ_0) \text{ as per transition rule } \delta(q_0, b, a) = \{(q_1, \epsilon)\} \\ &\vdash (q_1, \epsilon, aZ_0) \text{ as per transition rule } \delta(q_0, b, a) = \{(q_1, \epsilon)\} \\ &\vdash \text{There is no defined move.} \end{aligned}$$

So the automaton stops and the string is not accepted.

c) Moves for the input string  $aabbb$ :

$$\begin{aligned} (q_0, aabbb, Z_0) &\vdash (q_0, abbb, aZ_0) \text{ as per transition rule } \delta(q_0, a, Z_0) = \{(q_0, aZ_0)\} \\ &\vdash (q_0, bbb, aaZ_0) \text{ as per transition rule } \delta(q_0, a, a) = \{(q_0, aa)\} \\ &\vdash (q_1, bb, aZ_0) \text{ as per transition rule } \delta(q_0, b, a) = \{(q_1, \epsilon)\} \\ &\vdash (q_1, b, Z_0) \text{ as per transition rule } \delta(q_0, b, a) = \{(q_1, \epsilon)\} \\ &\vdash \text{There is no defined move.} \end{aligned}$$

So the automaton stops and the string is not accepted.

## 2. $\epsilon$ - move

Suppose that  $\delta(q_1, \epsilon, x) = \{(q_2, y), \dots\}$ . Then the following move without consuming an input symbol is possible:

$$(q_1, aW, xZ) \vdash (q_2, aW, yZ),$$

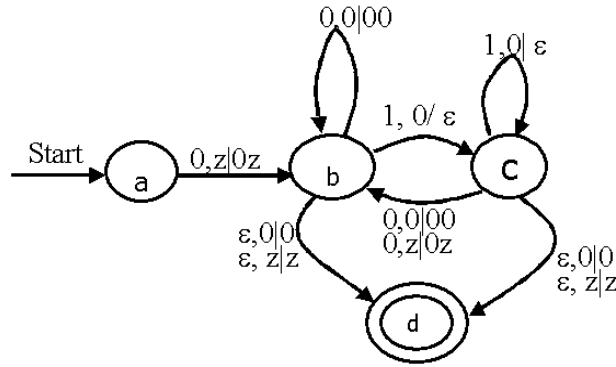
This notation says that in moving from state  $q_1$  to state  $q_2$ , an input symbol 'a' is not consumed from the input string  $aW$ , and the symbol 'x' at the top (left) of the stack  $xZ$  is replaced with symbol 'y', leaving  $yZ$  on the stack. In this move, the tape head position will not move forward. This move is usually used to represent non-determinism.

The relation  $\vdash^*$  is the reflexive-transitive closure of  $\vdash$  used to represent zero or more moves of PDA. For the above example,  $(q_0, aabb, Z_0) \vdash^* (q_2, \epsilon, \epsilon)$ .

### Example 2:

Design a PDA to accept the set of all strings of 0's and 1's such that no prefix has more 1's than 0's.

Solution: The transition diagram could be given as figure 3.



**Figure 3: Transition diagram**

$M = (\{a, b, c, d\}, \{0,1\}, \{0,1,Z\}, \delta, a, Z, \{d\})$ , where  $\delta$  is given by:

$$\delta(a, 0, Z) = \{(b, 0Z)\}$$

$$\delta(b, 0, 0) = \{(b, 00)\}$$

$$\delta(b, 1, 0) = \{(c, \epsilon)\}$$

$$\delta(c, 0, 0) = \{(b, 00)\}$$

$$\delta(c, 1, 0) = \{(c, \epsilon)\}$$

$$\delta(b, \epsilon, 0) = \{(d, 0)\}$$

$$\delta(b, \epsilon, Z) = \{(d, Z)\}$$

$$\delta(c, 0, Z) = \{(b, 0Z)\}$$

$$\delta(c, \epsilon, 0) = \{(d, 0)\}$$

$$\delta(c, \epsilon, Z) = \{(d, Z)\}$$

For all other moves, PDA stops.

Moves for the input string 0010110 is given by:

$(a, 0010110, Z) \vdash (b, 010110, 0Z) \vdash (b, 10110, 00Z) \vdash (c, 0110, 0Z) \vdash (b, 110, 00Z) \vdash$   
 $(c, 10, 0Z) \vdash (c, 0, Z) \vdash (b, \epsilon, 0Z) \vdash (d, \epsilon, 0Z).$

So the input string is accepted by the PDA.

Moves for 011

$(a, 011, Z) \vdash (b, 11, 0Z) \vdash (c, 1, Z) \vdash$  no move, so PDA stops

### Exercises:

Construct PDA:

1. For the language  $L = \{wcw^R \mid w \in \{a, b\}^*, c \in \Sigma\}$
2. Accepting the set of all strings over  $\{a, b\}$  with equal number of a's and b's. Show the moves for abbaba
3. For the language  $L = \{a^n b^{2n} \mid a, b \in \Sigma, n \geq 0\}$
4. Accepting the language of balanced parentheses, (consider any type of parentheses)

**Session by Ms. Jibi Abraham:**

**Finite Automata & Formal Languages**

**Push Down Automata**

**Languages of PDA**

**1. Languages of PDA**

There are 2 ways of accepting an input string PDA

- a. Accept by Final state

After consuming the input, PDA enters a final state. The content of the stack is irrelevant.

- b. Accept by empty stack

After consuming the input, stack of the PDA will be empty. The current state could be final or non-final state.

Both methods are equivalent. It is possible to convert a PDA accept by final state to another PDA accept by empty stack and also the vice versa. Usually the languages that a PDA accept by final state and PDA by empty stack are different. For example the language  $\{L = a^n b^m \mid n \geq m\}$ , the appropriate PDA could be by final state. After consuming the input, the stack may not be empty.

**Accept by Final state:**

Let  $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$  be a PDA. Then  $L(P)$ , the language accepted by P by the final state is

$$\{w \mid (q_0, w, Z_0) \xrightarrow{*} (q, \varepsilon, \alpha)\}, \text{ where } q \in F \text{ and } \alpha \in \Gamma^*$$

Example:  $L = \{ww^R \mid w \text{ is in } (0 + 1)^*\}$ , the language for even length palindrome. Acceptable input strings are like 00, 1111, 0110, 101101, and 110011. In the string 0110, the difficulty is how to decide the middle of the input string? The 3rd 1 can be part of  $w$  or can be part of  $w^R$ . The PDA could be constructed as below.

$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, q_2)$ , where  $\delta$  is defined by:

$$\delta(q_0, 0, Z_0) = \{(q_0, 0Z_0)\}$$

$$\delta(q_0, 1, Z_0) = \{(q_0, 1Z_0)\}$$

$$\delta(q_0, 0, 0) = \{(q_0, 00)\}$$

$$\delta(q_0, 1, 1) = \{(q_0, 11)\}$$

$$\delta(q_0, 0, 1) = \{(q_0, 01)\}$$

$$\delta(q_0, 1, 0) = \{(q_0, 10)\}$$

$$\delta(q_0, \varepsilon, Z_0) = \{(q_1, Z_0)\}$$

$$\delta(q_0, \varepsilon, 0) = \{(q_1, 0)\}$$



$$\begin{aligned}\delta(q_0, \varepsilon, 1) &= \{(q_1, 1)\} \\ \delta(q_1, 0, 0) &= \{(q_1, \varepsilon)\} \\ \delta(q_1, 1, 1) &= \{(q_1, \varepsilon)\} \\ \delta(q_1, \varepsilon, Z_0) &= \{(q_2, Z_0)\}\end{aligned}$$

$$(q_0, ww^R, Z_0) \vdash^* (q_0, w^R, w^R Z_0) \vdash (q_1, w^R, w^R Z_0) \vdash^* (q_1, \varepsilon, Z_0) \vdash (q_2, \varepsilon, Z_0)$$

The transition diagram for the PDA is given in figure 1.

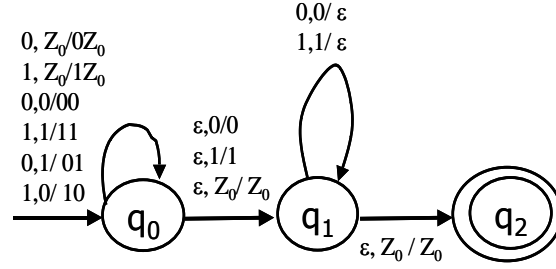


Figure 1: Transition Diagram for  $L = \{ww^R\}$

The moves of the PDA for the input string 101101 are given figure 2.

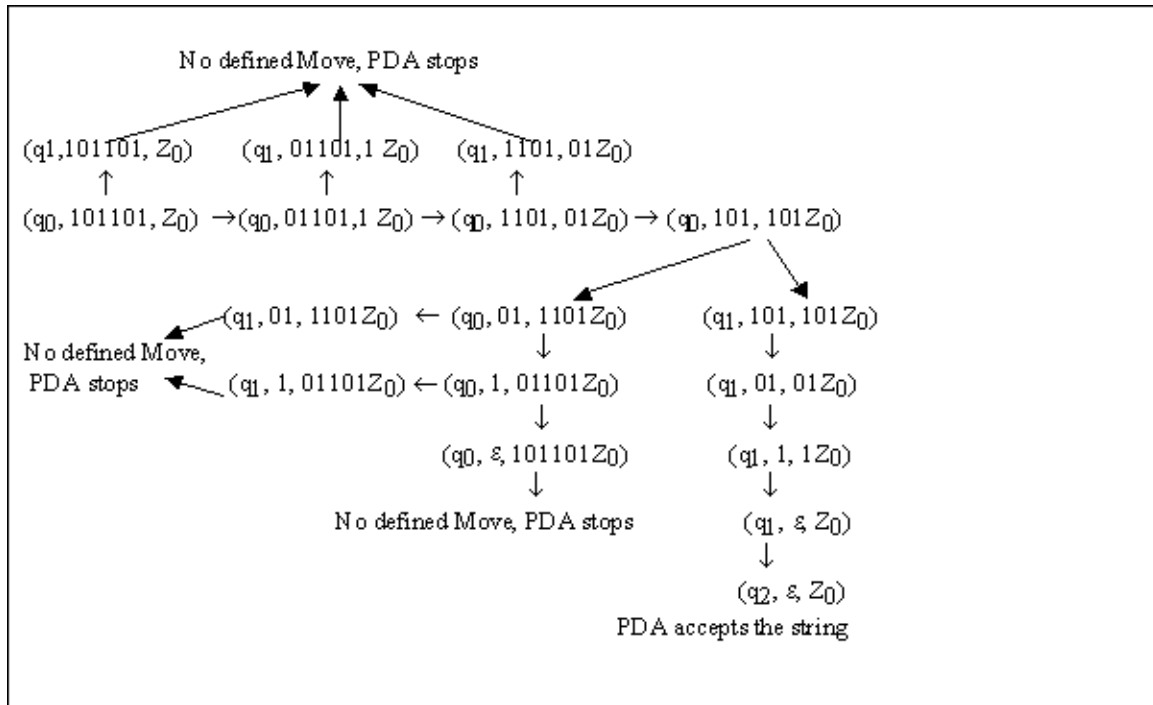


Figure 2: moves of PDA for string 101101

### Accept by empty stack:

Let PDA  $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ . We define the language accepted by empty stack by

$$N(P) = \{w \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)\}, \text{ where } q \in Q$$

### Example:

Construct PDA to accept by empty stack for the language  $L = \{ww^R \mid w \text{ is in } (0+1)^*\}$

Instead of the transition  $\delta(q_1, \varepsilon, Z_0) = \{(q_2, Z_0)\}$  give  $\delta(q_1, \varepsilon, Z_0) = \{(q_2, \varepsilon)\}$  to get accept by empty stack. The set of accepting states are irrelevant. This example also shows that  $L(P) = N(P)$

### Example:

Construct PDA to accept by empty stack for the language  $L = \{0^i 1^j \mid 0 \leq i \leq j\}$

The transition diagram for the PDA is given in Figure 3.

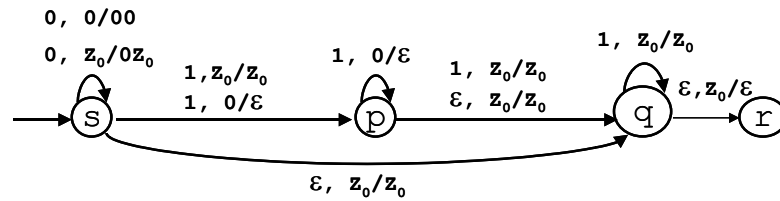


Figure 3: transition diagram of  $0^i 1^j \mid 0 \leq i \leq j$

## 2. Conversion between the two forms:

### a. From Empty Stack to Final State:

Theorem: If  $L = N(P_N)$  for some PDA  $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$ , then there is a PDA  $P_F$  such that  $L = L(P_F)$

Proof:

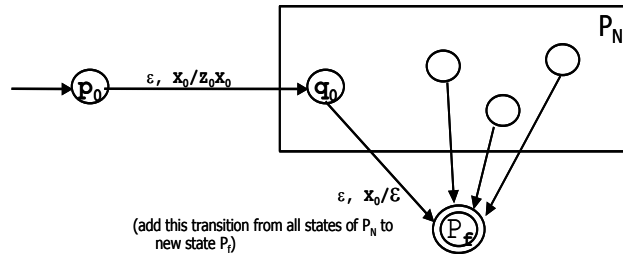


Figure 4:  $P_F$  simulates  $P_N$

The method of conversion is given in figure 4.

We use a new symbol  $X_0$ , which must be not symbol of  $\Gamma$  to denote the stack start symbol for  $P_F$ . Also add a new start state  $p_0$  and final state  $p_f$  for  $P_F$ . Let  $P_F = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{P_f\})$ , where  $\delta_F$  is defined by

$\delta_F(p_0, \varepsilon, X_0) = \{(q_0, Z_0 X_0)\}$  to push  $X_0$  to the bottom of the stack

$\delta_F(q, a, y) = \delta_N(q, a, y)$   $a \in \Sigma$  or  $a = \varepsilon$  and  $y \in \Gamma$ , same for both  $P_N$  and  $P_F$ .

$\delta_F(q, \varepsilon, X_0) = \{(P_f, \varepsilon)\}$  to accept the string by moving to final state.

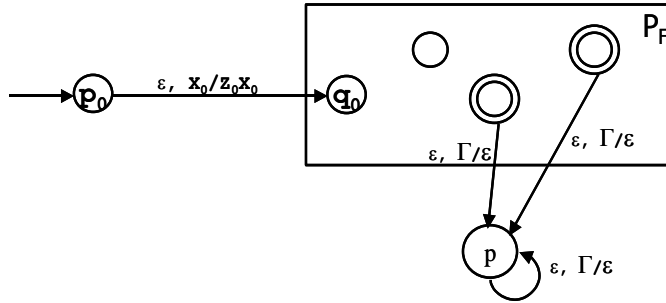
The moves of  $P_F$  to accept a string  $w$  can be written like:

$$(p_0, w, X_0) \vdash_{P_F} (p_0, w, Z_0 X_0) \vdash^*_{P_F} (q, \varepsilon, X_0) \vdash (P_f, \varepsilon, \varepsilon)$$

### b. From Final State to Empty Stack:

Theorem: If  $L = L(P_F)$  for some PDA  $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$ , then there is a PDA  $P_N$  such that  $L = N(P_N)$

Proof:



**Figure 5:  $P_N$  simulates  $P_F$**

The method of conversion is given in figure 5.

To avoid  $P_F$  accidentally emptying its stack, initially change the stack start content from  $Z_0$  to  $Z_0 X_0$ . Also add a new start state  $p_0$  and final state  $p$  for  $P_N$ . Let  $P_N = (Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0)$ , where  $\delta_N$  is defined by:

$\delta_N(p_0, \varepsilon, X_0) = \{(q_0, Z_0 X_0)\}$  to change the stack content initially

$\delta_N(q, a, y) = \delta_F(q, a, y)$ ,  $a \in \Sigma$  or  $a = \varepsilon$  and  $y \in \Gamma$ , same for both

$\delta_N(q, \varepsilon, y) = \{(p, \varepsilon)\}$ ,  $q \in F$ ,  $y \in \Gamma$  or  $y = X_0$ , same for both

$\delta_N(p, \varepsilon, y) = \{(p, \varepsilon)\}$ ,  $y \in \Gamma$  or  $y = X_0$ , to pop the remaining stack contents.

The moves of  $P_N$  to accept a string  $w$  can be written like:

$$(p_0, w, X_0) \vdash_{P_N} (q_0, w, Z_0 X_0) \vdash^*_{P_N} (q, \varepsilon, X_0) \vdash (p, \varepsilon, \varepsilon)$$

**Example:**

Construct PDA to accept by final state the language of all strings of 0's and 1's such that number of 1's is less than number of 0's. Also convert the PDA to accept by empty stack.

Solution:

PDA by final state is given by  $M = (\{q_0, q_1\}, \{0, 1\}, \{0, 1, Z\}, \delta, q_0, Z, \{q_1\})$ , where  $\delta$  is given by:

$$\delta(q_0, 0, Z) = \{(q_0, 0Z)\}$$

$$\delta(q_0, 1, Z) = \{(q_0, 1Z)\}$$

$$\delta(q_0, 0, 0) = \{(q_0, 00)\}$$

$$\delta(q_0, 0, 1) = \{(q_0, \varepsilon)\}$$

$$\delta(q_0, 1, 1) = \{(q_0, 11)\}$$

$$\delta(q_0, 1, 0) = \{(q_0, \varepsilon)\}$$

$$\delta(q_0, \varepsilon, Z) = \{(q_1, Z)\}$$

$$\delta(q_0, \varepsilon, 0) = \{(q_1, 0)\}$$

For all other moves, PDA stops.

PDA by empty stack is given by  $M = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, Z\}, \delta', q_0, Z)$ , where  $\delta'$  is the union of  $\delta$  and the transitions given below:

$$\delta(q_1, \varepsilon, Z) = \{(q_2, \varepsilon)\}$$

$$\delta(q_1, \varepsilon, 0) = \{(q_2, \varepsilon)\}$$

$$\delta(q_2, \varepsilon, 0) = \{(q_2, \varepsilon)\}$$

$$\delta(q_2, \varepsilon, Z) = \{(q_2, \varepsilon)\}$$

**Exercises:**

Design nPDA to accept the language:

1.  $\{a^i b^j c^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\}$
2.  $\{a^i b^j c^{i+j} \mid i, j \geq 0\}$
3.  $\{a^i b^{i+j} c^j \mid i \geq 0, j \geq 1\}$

# Finite Automata & Formal Languages

## Push Down Automata

### Equivalence of PDA and CFG

#### I. Equivalence of PDA and CFG

The aim is to prove that the following three classes of languages are same:

1. Context Free Language defined by CFG
2. Language accepted by PDA by final state
3. Language accepted by PDA by empty stack

It is possible to convert between any 3 classes. The representation is shown in figure 1.

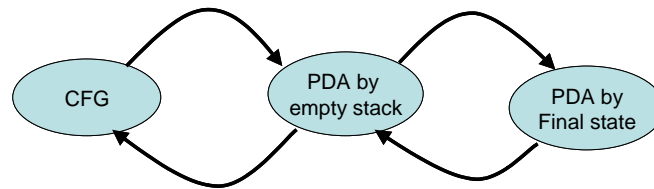


Figure 1: Equivalence of PDA and CFG

From CFG to PDA:

Given a CFG  $G$ , we construct a PDA  $P$  that simulates the leftmost derivations of  $G$ . The stack symbols of the new PDA contain all the terminal and non-terminals of the CFG. There is only one state in the new PDA; all the rest of the information is encoded in the stack. Most transitions are on  $\epsilon$ , one for each production. New transitions are added, each one corresponding to terminals of  $G$ . For every intermediate sentential form  $uA\alpha$  in the leftmost derivation of  $w$  (initially  $w = uv$  for some  $v$ ),  $M$  will have  $A\alpha$  on its stack after reading  $u$ . At the end (case  $u = w$ ) the stack will be empty.

Let  $G = (V, T, Q, S)$  be a CFG. The PDA which accepts  $L(G)$  by empty stack is given by:

$P = (\{q\}, T, V \cup T, \delta, q, S)$  where  $\delta$  is defined by:

1. For each variable  $A$  include a transition,  
 $\delta(q, \epsilon, A) = \{(q, b) \mid A \rightarrow b \text{ is a production of } Q\}$
2. For each terminal  $a$ , include a transition  
 $\delta(q, a, a) = \{(q, \epsilon)\}$

CFG to PDA conversion is another way of constructing PDA. First construct CFG, and then convert CFG to PDA.

**Example:**

Convert the grammar with following production to PDA accepted by empty stack:

$$\begin{aligned} S &\rightarrow 0S1 \mid A \\ A &\rightarrow 1A0 \mid S \mid \varepsilon \end{aligned}$$

Solution:

$P = (\{q\}, \{0, 1\}, \{0, 1, A, S\}, \delta, q, S)$ , where  $\delta$  is given by:

$$\begin{aligned} \delta(q, \varepsilon, S) &= \{(q, 0S1), (q, A)\} \\ \delta(q, \varepsilon, A) &= \{(q, 1A0), (q, S), (q, \varepsilon)\} \\ \delta(q, 0, 0) &= \{(q, \varepsilon)\} \\ \delta(q, 1, 1) &= \{(q, \varepsilon)\} \end{aligned}$$

**From PDA to CFG:**

Let  $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$  be a PDA. An equivalent CFG is  $G = (V, \Sigma, R, S)$ , where  $V = \{S, [pXq]\}$ , where  $p, q \in Q$  and  $X \in \Gamma$ , productions of  $R$  consists of

1. For all states  $p$ ,  $G$  has productions  $S \rightarrow [q_0Z_0p]$
2. Let  $\delta(q, a, X) = \{(r, Y_1Y_2 \dots Y_k)\}$  where  $a \in \Sigma$  or  $a = \varepsilon$ ,  $k$  can be 0 or any number and  $r_1r_2 \dots r_k$  are list of states.  $G$  has productions
$$[qXr_k] \rightarrow a[rY_1r_1][r_1Y_2r_2] \dots [r_{k-1}Y_kr_k]$$

If  $k = 0$  then  $[qXr] \rightarrow a$

**Example:**

Construct PDA to accept if-else of a C program and convert it to CFG. (This does not accept if –if –else-else statements).

Let the PDA  $P = (\{q\}, \{i, e\}, \{X, Z\}, \delta, q, Z)$ , where  $\delta$  is given by:

$$\delta(q, i, Z) = \{(q, XZ)\}, \delta(q, e, X) = \{(q, \varepsilon)\} \text{ and } \delta(q, \varepsilon, Z) = \{(q, \varepsilon)\}$$

Solution:

Equivalent productions are:

$$\begin{aligned} S &\rightarrow [qZq] \\ [qZq] &\rightarrow i[qXq][qZq] \end{aligned}$$

$$[qXq] \rightarrow e$$

$$[qZq] \rightarrow \varepsilon$$

If  $[qZq]$  is renamed to A and  $[qXq]$  is renamed to B, then the CFG can be defined by:

$$G = (\{S, A, B\}, \{i, e\}, \{S \rightarrow A, A \rightarrow iBA \mid \varepsilon, B \rightarrow e\}, S)$$

**Example:**

Convert PDA to CFG. PDA is given by  $P = (\{p, q\}, \{0, 1\}, \{X, Z\}, \delta, q, Z)$ , Transition

function  $\delta$  is defined by:

$$\delta(q, 1, Z) = \{(q, XZ)\}$$

$$\delta(q, 1, X) = \{(q, XX)\}$$

$$\delta(q, \varepsilon, X) = \{(q, \varepsilon)\}$$

$$\delta(q, 0, X) = \{(p, X)\}$$

$$\delta(p, 1, X) = \{(p, \varepsilon)\}$$

$$\delta(p, 0, Z) = \{(q, Z)\}$$

Solution:

Add productions for start variable

$$S \rightarrow [qZq] \mid [qZp]$$

For  $\delta(q, 1, Z) = \{(q, XZ)\}$

$$[qZq] \rightarrow 1[qXq][qZq]$$

$$[qZq] \rightarrow 1[qXp][pZq]$$

$$[qZp] \rightarrow 1[qXq][qZp]$$

$$[qZp] \rightarrow 1[qXp][pZp]$$

For  $\delta(q, 1, X) = \{(q, XX)\}$

$$[qXq] \rightarrow 1[qXq][qXq]$$

$$[qXq] \rightarrow 1[qXp][pXq]$$

$$[qXp] \rightarrow 1[qXq][qXp]$$

$$[qXp] \rightarrow 1[qXp][pXp]$$

For  $\delta(q, \varepsilon, X) = \{(q, \varepsilon)\}$

$$[qXq] \rightarrow \varepsilon$$

For  $\delta(q, 0, X) = \{(p, X)\}$

$$[qXq] \rightarrow 0[pXq]$$

$$[qXp] \rightarrow 0[pXp]$$

For  $\delta(p, 1, X) = \{(p, \varepsilon)\}$

$$[pXp] \rightarrow 1$$

For  $\delta(p, 0, Z) = \{(q, Z)\}$

$$[pZq] \rightarrow 0[qZq]$$

$$[pZp] \rightarrow 0[qZp]$$

Renaming the variables  $[qZq]$  to A,  $[qZp]$  to B,  $[pZq]$  to C,  $[pZp]$  to D,  $[qXq]$  to E  $[qXp]$  to F,  $[pXp]$  to G and  $[pXq]$  to H, the equivalent CFG can be defined by:

$G = (\{S, A, B, C, D, E, F, G, H\}, \{0,1\}, R, S)$ . The productions of R also are to be renamed accordingly.

### Exercises:

a. Convert to PDA, CFG with productions:

$$1. A \rightarrow aAA, A \rightarrow aS \mid bS \mid a$$

$$2. S \rightarrow SS \mid (S) \mid \varepsilon$$

$$3. S \rightarrow aAS \mid bAB \mid aB, A \rightarrow bBB \mid aS \mid a, B \rightarrow bA \mid a$$

b. Convert to CFG, PDA with transition function:

$$\delta(q, 0, Z) = \{(q, XZ)\}$$

$$\delta(q, 0, X) = \{(q, XX)\}$$

$$\delta(q, 1, X) = \{(q, X)\}$$

$$\delta(q, \varepsilon, X) = \{(p, \varepsilon)\}$$

$$\delta(p, 1, X) = \{(p, XX)\}$$

$$\delta(p, \varepsilon, X) = \{(p, \varepsilon)\}$$

$$\delta(p, 1, Z) = \{(p, \varepsilon)\}$$

## II. Deterministic PDA:

NPDA provides non-determinism to PDA. Deterministic PDA's (DPDA) are very useful for use in programming languages. For example Parsers used in YACC are DPDA's.

### Definition:

A PDA  $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$  is deterministic if and only if,

$$1. \delta(q, a, X) \text{ has at most one member for } q \in Q, a \in \Sigma \text{ or } a = \varepsilon \text{ and } X \in \Gamma$$

$$2. \text{If } \delta(q, a, X) \text{ is not empty for some } a \in \Sigma, \text{ then } \delta(q, \varepsilon, X) \text{ must be empty}$$

DPDA is less powerful than nPDA. The Context Free Languages could be recognized by

nPDA. The class of language DPDA accept is in between than of Regular language and

CFL. NPDA can be constructed for accepting language of palindromes, but not by DPDA.



**Example:**

Construct DPDA which accepts the language  $L = \{wcw^R \mid w \in \{a, b\}^*, c \in \Sigma\}$ .

The transition diagram for the DPDA is given in figure 2.

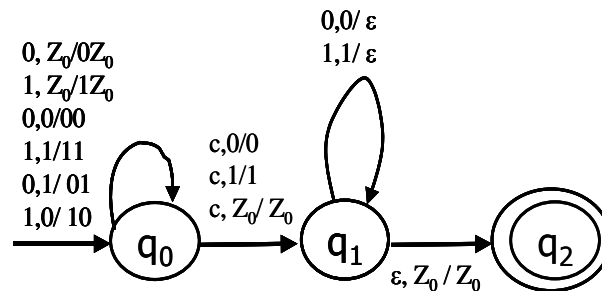


Figure 2: DPDA  $L = \{wcw^R\}$

**Exercises:**

Construct DPDA for the following:

1. Accepting the language of balanced parentheses. (Consider any type of parentheses)
2. Accepting strings with number of a's is more than number of b's
3. Accepting  $\{0^n 1^m \mid n \geq m\}$

**DPDA and Regular Languages:**

The class of languages DPDA accepts is in between regular languages and CFLs. The DPDA languages include all regular languages. The two modes of acceptance are not same for DPDA.

To accept with final state:

If  $L$  is a regular language,  $L = L(P)$  for some DPDA  $P$ . PDA surely includes a stack, but the DPDA used to simulate a regular language does not use the stack. The stack is inactive always. If  $A$  is the FA for accepting the language  $L$ , then  $\delta_P(q, a, Z) = \{(p, Z)\}$  for all  $p, q \in Q$  such that  $\delta_A(q, a) = p$ .

To accept with empty stack:

Every regular language is not  $N(P)$  for some DPDA  $P$ . A language  $L = N(P)$  for some DPDA  $P$  if and only if  $L$  has prefix property. Definition of prefix property of  $L$  states that if  $x, y \in L$ , then  $x$  should not be a prefix of  $y$ , or vice versa. Non-Regular language  $L = WcW^R$  could be accepted by DPDA with empty stack, because if you take any  $x, y \in L(WcW^R)$ ,  $x$  and  $y$

satisfy the prefix property. But the language,  $L=\{0^*\}$  could be accepted by DPDA with final state, but not with empty stack, because strings of this language do not satisfy the prefix property. So  $N(P)$  are properly included in CFL  $L$ , ie.  $N(P) \subseteq L$

### **DPDA and Ambiguous grammar:**

DPDA is very important to design of programming languages because languages DPDA accept are unambiguous grammars. But all unambiguous grammars are not accepted by DPDA. For example  $S \rightarrow 0S0|1S1| \epsilon$  is an unambiguous grammar corresponds to the language of palindromes. This language is accepted by only nPDA. If  $L = N(P)$  for DPDA  $P$ , then surely  $L$  has unambiguous CFG.

If  $L = L(P)$  for DPDA  $P$ , then  $L$  has unambiguous CFG. To convert  $L(P)$  to  $N(P)$  to have prefix property by adding an end marker  $\$$  to strings of  $L$ . Then convert  $N(P)$  to CFG  $G'$ .

From  $G'$  we have to construct  $G$  to accept  $L$  by getting rid of  $\$$ . So add a new production

$\$ \rightarrow \epsilon$  as a variable of  $G$ .

# Simplification of CFG

The goal is to take an arbitrary Context Free Grammar  $G = (V, T, P, S)$  and perform transformations on the grammar that preserve the language generated by the grammar but reach a specific format for the productions. A CFG can be simplified by eliminating

## 1. Useless symbols

Those variables or terminals that do not appear in any derivation of a terminal string starting from Start variable,  $S$ .

## 2. $\epsilon$ - Productions

$A \rightarrow \epsilon$ , where  $A$  is a variable

## 3. Unit production

$A \rightarrow B$ ,  $A$  and  $B$  are variables

# 1. Eliminate useless symbols:

**Definition:** Symbol  $X$  is useful for a grammar  $G = (V, T, P, S)$  if there is  $S \xRightarrow{*} \alpha X \beta \xRightarrow{*} w$ ,  $w \in \Gamma^*$ . If  $X$  is not useful, then it is useless.

Omitting useless symbols from a grammar does not change the language generated

**Example:**

$S \rightarrow aSb \mid \epsilon \mid A$ ,  $A \rightarrow aA$ . Here  $A$  is a useless symbol  
 $S \rightarrow A$ ,  $A \rightarrow aA \mid \epsilon$ ,  $B \rightarrow bB$ . Here  $B$  is a useless symbol

Symbol  $X$  is useful if both the conditions given below are satisfied in that order itself.

1.  $X$  is generating variable. ie if  $X \xRightarrow{*} w$ , where  $w \in \Gamma^*$
2.  $X$  is reachable. ie if  $S \xRightarrow{*} \alpha X \beta$

**Theorem:**

Let  $G = (V, T, P, S)$  be a CFG and assume that  $L(G) \neq \phi$ , then  $G_1 = (V_1, T_1, P_1, S)$  be a grammar without useless symbols by

1. Eliminating non generating symbols
2. Eliminating symbols that are non reachable

Elimination has to be performed only in the order of 1 followed by 2. Otherwise the grammar produced will not be completely useless symbols eliminated.

**Example:**

Consider a grammar with productions:  $S \rightarrow AB|a$ ,  $A \rightarrow a$ . If the order of eliminations is 1 followed by 2 gives  $S \rightarrow a$ . If the order of eliminations is 2 followed by 1 gives  $S \rightarrow a$ ,  $A \rightarrow a$ . Here A is still useless symbol, so completely useless symbols are not eliminated.

## Eliminate non-generating symbols:

Generating symbols follow to one of the categories below:

1. Every symbol of T is generating
2. If  $A \rightarrow \alpha$  and  $\alpha$  is already generating, then A is generating

Non-generating symbols = V - generating symbols.

**Example:**

$G = (\{S, A, B\}, \{a\}, S \rightarrow AB | a, A \rightarrow a, S)$

Solution:

B is a non-generating symbol. After eliminating B, we get

$G1 = (\{S, A\}, \{a\}, \{S \rightarrow a, A \rightarrow a\}, S)$

**Example:**

Eliminate the non-generating symbols from  $S \rightarrow aS | A | C$ ,  $A \rightarrow a$ ,  $B \rightarrow aa$ ,  $C \rightarrow aCb$

Solution:

C is a non-generating symbol. After eliminating C gets,  $S \rightarrow aS | A$ ,  $A \rightarrow a$ ,  $B \rightarrow aa$

**Eliminate symbols that are non reachable:**

Dependency Graph: For the production  $C \rightarrow xDy$ , the dependency graph is given below.

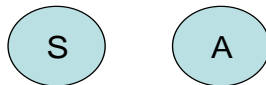


Draw dependency graph for all productions. If there is no edge reaching a variable  $X$  from Start symbol  $S$ , then  $X$  is non reachable.

**Example:**

Eliminate non-reachable symbols from  $G = (\{S, A\}, \{a\}, \{S \rightarrow a, A \rightarrow a\}, S)$

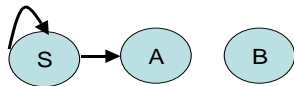
Solution:



Draw the dependency graph as given above. A is non-reachable from S. After eliminating A,  $G_1 = (\{S\}, \{a\}, \{S \rightarrow a\}, S)$

**Example:**

Eliminate non-reachable symbols from  $S \rightarrow aS \mid A, A \rightarrow a, B \rightarrow aa$



Draw the dependency graph as given above. B is non-reachable from S. After eliminating B, we get the grammar with productions  $S \rightarrow aS \mid A, A \rightarrow a$

**Example:**

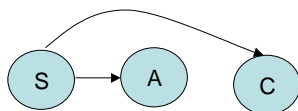
Eliminate useless symbols from the grammar with productions  $S \rightarrow AB \mid CA, B \rightarrow BC \mid AB, A \rightarrow a, C \rightarrow AB \mid b$

Step 1: Eliminate non-generating symbols

$$V_1 = \{A, C, S\}$$

$$P_1 = \{S \rightarrow CA, A \rightarrow a, C \rightarrow b\}$$

Step 2: Eliminate symbols that are non reachable



Draw the dependency graph as given above. All Variables are reachable. So the final variables and productions are same  $V_1$  and  $P_1$ .

$$V_2 = \{A, C, S\}$$

$$P_2 = \{S \rightarrow CA, A \rightarrow a, C \rightarrow b\}$$

### Exercises:

Eliminate useless symbols from the grammar

$$1. P = \{S \rightarrow aAa, A \rightarrow Sb \mid bCC, C \rightarrow abb, E \rightarrow aC\}$$

$$2. P = \{S \rightarrow aBa \mid BC, A \rightarrow aC \mid BCC, C \rightarrow a, B \rightarrow bcc, D \rightarrow E, E \rightarrow d\}$$

$$3. P = \{S \rightarrow aAa, A \rightarrow bBB, B \rightarrow ab, C \rightarrow aB\}$$

$$4. P = \{S \rightarrow aS \mid AB, A \rightarrow bA, B \rightarrow AA\}$$

### III. Eliminate $\varepsilon$ - Productions:

Most theorems and methods about grammars  $G$  assume  $L(G)$  does not contain  $\varepsilon$ . So if  $\varepsilon$  is not there in  $L(G)$ , then we have to find out an equivalent  $G$  without  $\varepsilon$ -productions. Example for a grammar  $G$  with  $\varepsilon$  - productions is

$$S \rightarrow ABA, A \rightarrow aA \mid \varepsilon, B \rightarrow bB \mid \varepsilon$$

The procedure to find out an equivalent  $G$  with out  $\varepsilon$ -productions

1. Find nullable variables
2. Add productions with nullable variables removed.
3. Remove  $\varepsilon$ -productions and duplicates

**Step 1:** Find set of nullable variables

**Nullable variables:** Variables that can be replaced by null ( $\varepsilon$ ). If  $A \xRightarrow{*} \varepsilon$  then  $A$  is a nullable variable.

In the grammar with productions  $S \rightarrow ABA, A \rightarrow aA \mid \varepsilon, B \rightarrow bB \mid \varepsilon$ ,  $A$  is nullable because of the production  $A \rightarrow \varepsilon$ .  $B$  is nullable because of the production  $B \rightarrow \varepsilon$ .  $S$  is nullable because both  $A$  and  $B$  are nullable.

Algorithm to find nullable variables is given below:

V: set of variables  
 $N_0 \leftarrow \{A \mid A \text{ in } V, \text{ production } A \rightarrow \epsilon\}$   
 Repeat  
      $N_i \leftarrow N_{i-1} \cup \{A \mid A \text{ in } V, A \rightarrow \alpha, \alpha \text{ in } N_{i-1}\}$   
 until  $N_i = N_{i-1}$

**Step 2:** Add productions with nullable variables removed

For each production of the form  $A \rightarrow w$ , create all possible productions of the form  $A \rightarrow w'$ , where  $w'$  is obtained from  $w$  by removing one or more occurrences of nullable variables.

Example:

In the grammar with productions  $S \rightarrow ABA$ ,  $A \rightarrow aA \mid \epsilon$ ,  $B \rightarrow bB \mid \epsilon$ ,  $A$ ,  $B$  and  $S$  are nullable variables. So after removing nullable variables we get the productions

$S \rightarrow ABA \mid BA \mid AA \mid AB \mid A \mid B \mid \epsilon$   
 $A \rightarrow aA \mid \epsilon \mid a$   
 $B \rightarrow bB \mid \epsilon \mid b$

**Step 3:** Remove  $\epsilon$ -productions and duplicates

The desired grammar consists of the original productions together with the productions constructed in step 2, minus any productions of the form  $A \rightarrow \epsilon$ .

**Example:**

For the above example we get the final grammar with productions

$S \rightarrow ABA \mid BA \mid AA \mid AB \mid A \mid B$   
 $A \rightarrow aA \mid a$   
 $B \rightarrow bB \mid b$

**Example:**

Find out the grammar without  $\epsilon$  - Productions

$G = (\{S, A, B, D\}, \{a\}, \{S \rightarrow aS \mid AB, A \rightarrow \epsilon, B \rightarrow \epsilon, D \rightarrow b\}, S)$

Solution:

Nullable variables = {S, A, B}

New Set of productions:

$S \rightarrow aS \mid a$

$S \rightarrow AB \mid A \mid B$

$D \rightarrow b$

$G_1 = (\{S, B, D\}, \{a\}, \{S \rightarrow aS \mid a \mid AB \mid A \mid B, D \rightarrow b\}, S)$

### Exercise:

Eliminate  $\epsilon$  - productions from the grammar

1.  $S \rightarrow a \mid Xb \mid aYa, X \rightarrow Y \mid \epsilon, Y \rightarrow b \mid X$
2.  $S \rightarrow Xa, X \rightarrow aX \mid bX \mid \epsilon$
3.  $S \rightarrow XY, X \rightarrow Zb, Y \rightarrow bW, Z \rightarrow AB, W \rightarrow Z, A \rightarrow aA \mid bB \mid \epsilon, B \rightarrow Ba \mid Bb \mid \epsilon$
4.  $S \rightarrow ASB \mid \epsilon, A \rightarrow aAS \mid a, B \rightarrow SbS \mid A \mid bb$

But if you have to get a grammar without  $\epsilon$  - productions and useless symbols, follow the sequence given below:

1. Eliminate  $\epsilon$  - productions and obtain  $G_1$
2. Eliminate useless symbols from  $G_1$  and obtain  $G_2$

### Example:

Eliminate  $\epsilon$  - productions and useless symbols from the grammar

$S \rightarrow a \mid aA \mid B \mid C, A \rightarrow aB \mid \epsilon, B \rightarrow aA, C \rightarrow aCD, D \rightarrow ddd$

Step 1: Eliminate  $\epsilon$  - productions

**Nullable = {A}**

$P_1 = \{S \rightarrow a \mid aA \mid B \mid C, A \rightarrow aB, B \rightarrow aA \mid a, C \rightarrow aCD, D \rightarrow ddd\}$

Step 2: Eliminating useless symbols

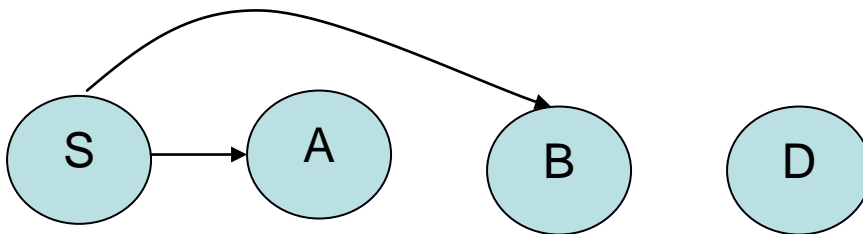
Step 2a: Eliminate non-generating symbols



Generating = {D, B, A, S}

P2 = {S  $\rightarrow$  a | aA | B, A  $\rightarrow$  aB, B  $\rightarrow$  aA | a, D  $\rightarrow$  ddd}

Step 2b: Eliminate non-reachable symbols



C is non-reachable, eliminating C gets

P3 = S  $\rightarrow$  a | aA | B, A  $\rightarrow$  aB, B  $\rightarrow$  aA | a}

## I. Eliminate unit productions:

**Definition:**

Unit production is of form  $A \rightarrow B$ , where A and B are variables.

Unit productions could complicate certain proofs and they also introduce extra steps into derivations that technically need not be there. The algorithm for eliminating unit productions from the set of production P is given below:

1. Add all non unit productions of P to P1
2. For each unit production  $A \rightarrow B$ , add to P1 all productions  $A \rightarrow \alpha$ , where  $B \rightarrow \alpha$  is a non-unit production in P.
3. Delete all the unit productions

**Example:**

Eliminate unit productions from  $S \rightarrow ABA \mid BA \mid AA \mid AB \mid A \mid B$ ,  $A \rightarrow aA \mid a$ ,  $B \rightarrow bB \mid b$

**Solution:**

The unit productions are  $S \rightarrow A$ ,  $S \rightarrow B$ . So A and B are derivable. Now add productions from derivable.

$S \rightarrow ABA \mid BA \mid AA \mid AB \mid A \mid B \mid aA \mid a \mid bB \mid b$   
 $A \rightarrow aA \mid a$   
 $B \rightarrow bB \mid b$

Remove unit productions from above productions to get

$S \rightarrow ABA \mid BA \mid AA \mid AB \mid aA \mid a \mid bB \mid b$   
 $A \rightarrow aA \mid a$   
 $B \rightarrow bB \mid b$

**Example:**

Eliminate unit productions from  $S \rightarrow Aa \mid B$ ,  $A \rightarrow a \mid bc \mid B$ ,  $B \rightarrow A \mid bb$

**Solution:**

Unit productions are  $S \rightarrow B$ ,  $A \rightarrow B$  and  $B \rightarrow A$ . So A and B are derivable. Add productions from derivable and eliminate unit productions to get,

$$S \rightarrow Aa \mid bb \mid a \mid bc$$

$$A \rightarrow a \mid bc \mid bb$$

$$B \rightarrow bb \mid a \mid bc$$

**Simplified Grammar:**

If you have to get a grammar without  $\varepsilon$  - productions, useless symbols and unit productions, follow the sequence given below:

1. Eliminate  $\varepsilon$  - productions from G and obtain  $G_1$
2. Eliminate unit productions from  $G_1$  and obtain  $G_2$
3. Eliminate useless symbols from  $G_2$  and obtain  $G_3$

**Example:**

Eliminate useless symbols,  $\varepsilon$  -productions and unit productions from the grammar with productions:

$$S \rightarrow a \mid aA \mid B \mid C, A \rightarrow aB \mid \varepsilon, B \rightarrow aA, C \rightarrow cCD, D \rightarrow ddd$$

Step 1: Eliminate  $\varepsilon$  -productions

$$\text{Nullable} = \{A\}$$

$$P_1 = \{S \rightarrow a \mid aA \mid B \mid C, A \rightarrow aB, B \rightarrow aA \mid a, C \rightarrow cCD, D \rightarrow ddd\}$$

Step 2: Eliminate unit productions

Unit productions are  $S \rightarrow B$  and  $S \rightarrow C$ . So Derivable variables are B and C.

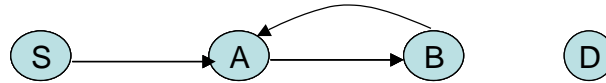
$$P_2 = \{S \rightarrow a \mid aA \mid cCD, A \rightarrow aB, B \rightarrow aA \mid a, C \rightarrow cCD, D \rightarrow ddd\}$$

Step 3: Eliminate useless symbols

After eliminate non-generating symbol C we get

$$P_3 = \{S \rightarrow a \mid aA, A \rightarrow aB, B \rightarrow aA \mid a, D \rightarrow ddd\}$$

After eliminate symbols that are non reachable



$$P_4 = \{S \rightarrow a \mid aA, A \rightarrow aB, B \rightarrow aA \mid a\}$$

So the equivalent grammar  $G_1 = (\{S, A, B\}, \{a\}, \{S \rightarrow a \mid aA, A \rightarrow aB, B \rightarrow aA \mid a\}, S)$

## II. Chomsky Normal Form (CNF)

Every nonempty CFL without  $\varepsilon$ , has a grammar in CNF with productions of the form:

1.  $A \rightarrow BC$ , where  $A, B, C \in V$
2.  $A \rightarrow a$ , where  $A \in V$  and  $a \in T$

Algorithm to produce a grammar in CNF:

1. Eliminate useless symbols,  $\varepsilon$ -productions and unit productions from the grammar
2. Elimination of terminals on RHS of a production
  - a) Add all productions of the form  $A \rightarrow BC$  or  $A \rightarrow a$  to  $P_1$
  - b) Consider a production  $A \rightarrow X_1X_2\dots X_n$  with some terminals of RHS. If  $X_i$  is a terminal say  $a_i$ , then add a new variable  $C_{ai}$  to  $V_1$  and a new production  $C_{ai} \rightarrow a_i$  to  $P_1$ . Replace  $X_i$  in A production of P by  $C_{ai}$
  - c) Consider  $A \rightarrow X_1X_2\dots X_n$ , where  $n \geq 3$  and all  $X_i$ 's are variables. Introduce new productions  $A \rightarrow X_1C_1, C_1 \rightarrow X_2C_2, \dots, C_{n-2} \rightarrow X_{n-1}X_n$  to  $P_1$  and  $C_1, C_2, \dots, C_{n-2}$  to  $V_1$

### Example:

Convert to CNF:  $S \rightarrow aAD, A \rightarrow aB \mid bAB, B \rightarrow b, D \rightarrow d$

Solution:

Step1: Simplify the grammar

Grammar is already simplified.

Step2a: Elimination of terminals on RHS

Change  $S \rightarrow aAD$  to  $S \rightarrow C_aAD, C_a \rightarrow a$

$A \rightarrow aB$  to  $A \rightarrow C_aB$

$A \rightarrow bAB$  to  $A \rightarrow C_bAB, C_b \rightarrow b$

Step2b: Reduce RHS with 2 variables

Change  $S \rightarrow C_aAD$  to  $S \rightarrow C_aC_1, C_1 \rightarrow AD$

$A \rightarrow C_bAB$  to  $A \rightarrow C_bC_2, C_2 \rightarrow AB$

Grammar converted to CNF is given below:

$G_1 = (\{S, A, B, D, C_a, C_b, C_1, C_2\}, \{a, b\}, S \rightarrow C_aC_1, A \rightarrow C_aB \mid C_bC_2, C_a \rightarrow a, C_b \rightarrow b, C_1 \rightarrow AD, C_2 \rightarrow AB, S)$

### Example:

Convert the grammar with following productions to CNF:

$P = \{S \rightarrow ASB \mid \varepsilon, A \rightarrow aAS \mid a, B \rightarrow SbS \mid A \mid bb\}$

Solution:

Step1: Simplify the grammar

Step 1a: Eliminate  $\varepsilon$ -productions: Consists of  $S \rightarrow \varepsilon$

Eliminating  $\varepsilon$ -productions from P to get:

$P_1 = \{S \rightarrow ASB \mid AB, A \rightarrow aAS \mid aA \mid a, B \rightarrow SbS \mid Sb \mid bS \mid b \mid A \mid bb\}$

Step 1b: Eliminate unit productions:  $B \rightarrow A$

$P_2 = \{S \rightarrow ASB \mid AB, A \rightarrow aAS \mid aA \mid a, B \rightarrow SbS \mid Sb \mid bS \mid b \mid bb \mid aAS \mid aA \mid a\}$

Step 1c: Eliminate useless symbols: no useless symbols

Step2: Convert to CNF

$P_3 = \{S \rightarrow AC_1 \mid AB, A \rightarrow C_aC_2 \mid C_aA \mid a,$   
 $B \rightarrow SC_3 \mid SC_b \mid C_bS \mid b \mid C_bC_b \mid C_aC_2 \mid C_aA \mid a,$   
 $C_a \rightarrow a, C_b \rightarrow b, C_1 \rightarrow SB, C_2 \rightarrow AS, C_3 \rightarrow C_bS\}$

### Exercises:

Convert the following grammar with productions to CNF:

1.  $S \rightarrow aSa \mid bSb \mid a \mid b \mid aa \mid bb$
2.  $S \rightarrow bA \mid aB, A \rightarrow bAA \mid aS \mid a, B \rightarrow aBB \mid bS \mid b$

3.  $S \rightarrow Aba, A \rightarrow aab, B \rightarrow AC$
4.  $S \rightarrow 0A0 \mid 1B1 \mid BB, A \rightarrow C, B \rightarrow S \mid A, C \rightarrow S \mid \varepsilon$
5.  $S \rightarrow aAa \mid bBb \mid \varepsilon, A \rightarrow C \mid a, B \rightarrow C \mid b, C \rightarrow CDE \mid \varepsilon, D \rightarrow A \mid B \mid ab$

### III. Greibach Normal Form (GNF):

Every nonempty language without  $\varepsilon$  is  $L(G)$  for some grammar  $G$  with productions are of the form

$$A \rightarrow a\alpha,$$

where  $a \in T$  and  $\alpha$  is a string of zero or more variables. Conversion to GNF is a complex process. Converting CFG in GNF to PDA gets a PDA without  $\varepsilon$ -rules.

Example:

Transform into Greibach normal form the grammar with productions

$$S \rightarrow 0S1 \mid 01$$

Solution:

$$S \rightarrow 0SA \mid 0A, A \rightarrow 1$$

### Usage of GNF:

Construction of PDA from a GNF grammar can be made more meaningful with GNF. Assume that the PDA has two states: start state  $q_0$  and accepting state  $q_1$ . An  $S$  rule of the form  $S \rightarrow aA_1A_2\dots A_n$  generates a transition that processes the terminal  $a$ , pushes the variables  $A_1A_2\dots A_n$  on the stack and enters  $q_1$ . The remainder of the computation uses the input symbol and the stack top to determine the appropriate transition.

## The Pumping Lemma for CFL

The *pumping lemma for regular languages* states that every sufficiently long string in a regular language contains a short sub-string that can be pumped. That is, inserting as many copies of the sub-string as we like always yields a string in the regular language.

The *pumping lemma for CFL's* states that there are always two short sub-strings close together that can be repeated, both the same number of times, as often as we like.

For example, consider a CFL  $L = \{a^n b^n \mid n \geq 1\}$ . Equivalent CNF grammar is having productions  $S \rightarrow AC \mid AB$ ,  $A \rightarrow a$ ,  $B \rightarrow b$ ,  $C \rightarrow SB$ . The parse tree for the string  $a^4 b^4$  is given in figure 1. Both leftmost derivation and rightmost derivation have same parse tree because the grammar is unambiguous.

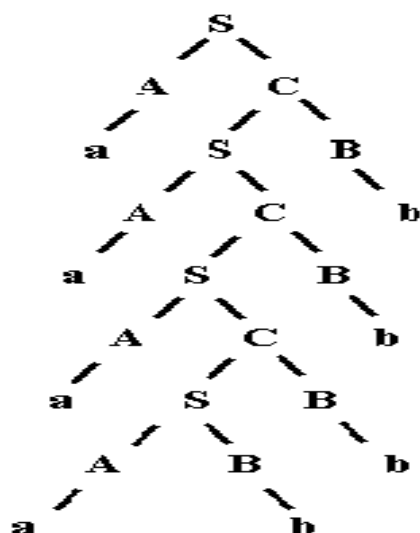


Figure 1: Parse tree for  $a^4 b^4$

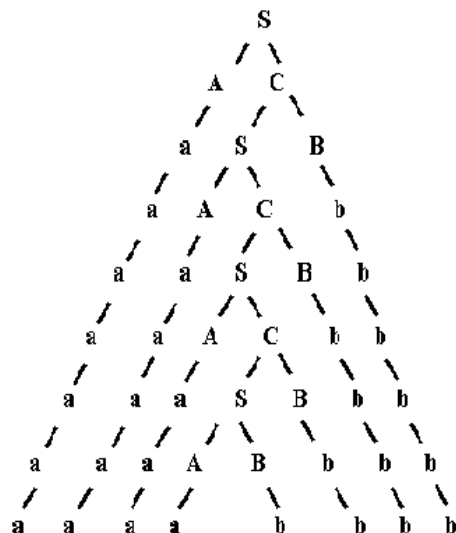


Figure 2: Extended Parse tree for  $a^4 b^4$

Extend the tree by duplicating the terminals generated at each level on all lower levels. The extended parse tree for the string  $a^4 b^4$  is given in figure 2. Number of symbols at each level is at most twice of previous level. 1 symbols at level 0, 2 symbols at 1, 4 symbols at 2 ...  $2^i$  symbols at level  $i$ . To have  $2^n$  symbols at bottom level, tree must be having at least depth of  $n$  and level of at least  $n+1$ .

### Pumping Lemma Theorem:

Let  $L$  be a CFL. Then there exists a constant  $k \geq 0$  such that if  $z$  is any string in  $L$  such that  $|z|$

$\geq k$ , then we can write  $z = uvwxy$  such that

1.  $|vwx| \leq k$  (that is, the middle portion is not too long).
2.  $vx \neq \varepsilon$  (since  $v$  and  $x$  are the pieces to be “pumped”, at least one of the strings we pump must not be empty).
3. For all  $i \geq 0$ ,  $uv^iwx^iy$  is in  $L$ .

**Proof:**

The parse tree for a grammar  $G$  in CNF will be a binary tree. Let  $k = 2^{n+1}$ , where  $n$  is the number of variables of  $G$ . Suppose  $z \in L(G)$  and  $|z| \geq k$ . Any parse tree for  $z$  must be of depth at least  $n+1$ . The longest path in the parse tree is at least  $n+1$ , so this path must contain at least  $n+1$  occurrences of the variables. By pigeonhole principle, some variables occur more than once along the path. Reading from bottom to top, consider the first pair of same variable along the path. Say  $X$  has 2 occurrences. Break  $z$  into  $uvwxy$  such that  $w$  is the string of terminals generated at the lower occurrence of  $X$  and  $vwx$  is the string generated by upper occurrence of  $X$ .

**Example parse tree:**

For the above example  $S$  has repeated occurrences, and the parse tree is shown in figure 3.  $w = ab$  is the string generated by lower occurrence of  $S$  and  $vwx = aabb$  is the string generated by upper occurrence of  $S$ . So here  $u=aa$ ,  $v=a$ ,  $w=ab$ ,  $x=b$ ,  $y=bb$ .

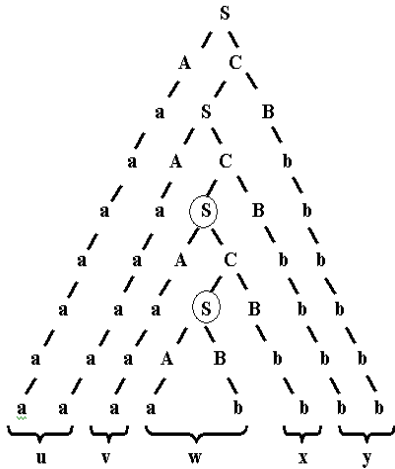


Figure 3: Parse tree for  $a^4b^4$  with repeated occurrences of  $S$  circled.

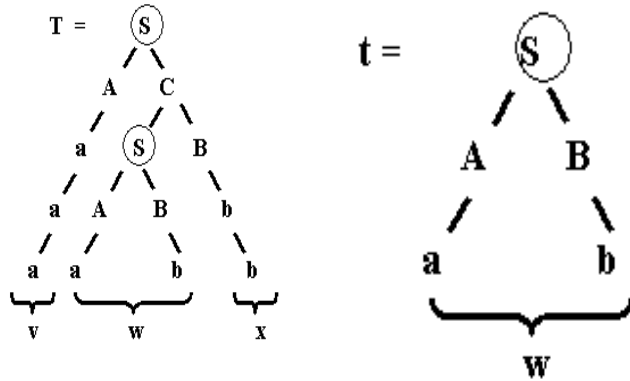


Figure 4: sub- trees

Let  $T$  be the subtree rooted at upper occurrence of  $S$  and  $t$  be subtree rooted at lower occurrence of  $S$ . These parse trees are shown in figure 4. To get  $uv^2wx^2y \in L$ , cut out  $t$  and replace it with copy of  $T$ . The parse tree for  $uv^2wx^2y \in L$  is given in figure 5. Cutting out  $t$  and replacing it with copy of  $T$  as many times to get a valid parse tree for  $uv^iwx^iy$  for  $i \geq 1$ .



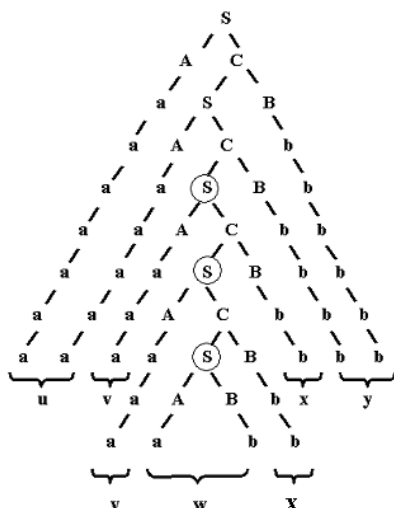


Figure 5: Parse tree for  $uv^2wx^2y$

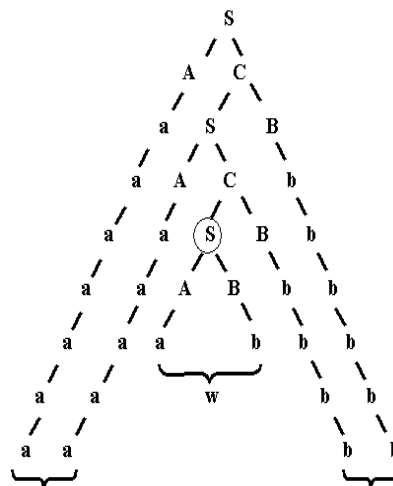


Figure 6: Parse tree for  $uwy \in L$

To get  $uwy \in L$ , cut  $T$  out of the original tree and replace it with  $t$  to get a parse tree of  $uv^0wx^0y = uwy$  as shown in figure 6.

### Pumping Lemma game:

1. To show that a language  $L$  is not a CFL, assume  $L$  is context free.
2. Choose an “appropriate” string  $z$  in  $L$
3. Express  $z = uvwxy$  following rules of pumping lemma
4. Show that  $uv^kwx^ky$  is not in  $L$ , for some  $k$
5. The above contradicts the Pumping Lemma
6. Our assumption that  $L$  is context free is wrong

### Example:

Show that  $L = \{a^ib^ic^i \mid i \geq 1\}$  is not CFL

Solution:

Assume  $L$  is CFL. Choose an appropriate  $z = a^n b^n c^n = uvwxy$ . Since  $|vwx| \leq n$  then  $vwx$  can either consists of

1. All  $a$ 's or all  $b$ 's or all  $c$ 's
2. Some  $a$ 's and some  $b$ 's
3. Some  $b$ 's and some  $c$ 's

Case 1:  $vwx$  consists of all  $a$ 's

If  $z = a^2b^2c^2$  and  $u = \epsilon$ ,  $v = a$ ,  $w = \epsilon$ ,  $x = a$  and  $y = b^2c^2$  then,  $uv^2wx^2y$  will be  $a^4b^2c^2 \neq L$

Case 2:  $vwx$  consists of some  $a$ 's and some  $b$ 's

If  $z = a^2b^2c^2$  and  $u = a$ ,  $v = a$ ,  $w = \epsilon$ ,  $x = b$ ,  $y = bc^2$ , then  $uv^2wx^2y$  will be  $a^3b^3c^2 \neq L$

Case 3:  $vwx$  consists of some  $b$ 's and some  $c$ 's

If  $z = a^2b^2c^2$  and  $u = a^2b$ ,  $v = b$ ,  $w = c$ ,  $x = \epsilon$ ,  $y = c$ , then  $uv^2wx^2y$  will be  $a^2b^3c^2 \neq L$

If you consider any of the above 3 cases,  $uv^2wx^2y$  will not be having an equal number of  $a$ 's,  $b$ 's and  $c$ 's. But Pumping Lemma says  $uv^2wx^2y \in L$ . Can't contradict the pumping lemma! Our original assumption must be wrong. So  $L$  is not context-free.

### Example:

Show that  $L = \{ww \mid w \in \{0, 1\}^*\}$  is not CFL

Solution:

Assume  $L$  is CFL. It is sufficient to show that  $L_1 = \{0^m1^n0^m1^n \mid m, n \geq 0\}$ , where  $n$  is pumping lemma constant, is a CFL. Pick any  $z = 0^n1^n0^n1^n = uvwxy$ , satisfying the conditions  $|vwx| \leq n$  and  $vx \neq \epsilon$ .

This language we prove by taking the case of  $i = 0$ , in the pumping lemma satisfying the condition  $uv^iwx^iy$  for  $i \geq 0$ .

$z$  is having a length of  $4n$ . So if  $|vwx| \leq n$ , then  $|uwy| \geq 3n$ . According to pumping lemma,  $uwy \in L$ . Then  $uwy$  will be some string in the form of  $tt$ , where  $t$  is repeating. If so,  $n \mid |t| \geq 3n/2$ .

**Suppose  $vwx$  is within first  $n$  0's:** let  $vx$  consists of  $k$  0's. Then  $uwy$  begins with  $0^{n-k}1^n$   $|uwy| = 4n-k$ . If  $uwy$  is some repeating string  $tt$ , then  $|t| = 2n-k/2$ .  $t$  does end in 0 but  $tt$  ends with 1. So second  $t$  is not a repetition of first  $t$ .

Example:  $z = 0^31^30^31^3$ ,  $vx = 0^2$  then  $uwy = tt = 01^30^31^3$ , so first  $t = 01^30$  and second  $t = 0^21^3$ . Both  $t$ 's are not same.

**Suppose  $vwx$  consists of 1<sup>st</sup> block of 0's and first block of 1's:**  $vx$  consists of only 0's if  $x = \epsilon$ , then  $uwy$  is not in the form  $tt$ . If  $vx$  has at least one 1, then  $|t|$  is at least  $3n/2$  and first  $t$  ends with a 0, not a 1.

Very similar explanations could be given for the cases of  $vwx$  consists of first block of 1's and  $vwx$  consists of 1<sup>st</sup> block of 1's and 2<sup>nd</sup> block of 0's. In all cases  $uwy$  is expected to be in the form of  $tt$ . But first  $t$  and second  $t$  are not the same string. So  $uwy$  is not in  $L$  and  $L$  is not context free.

**Example:**

Show that  $L = \{0^i 1^j 2^i 3^j \mid i \geq 1, j \geq 1\}$  is not CFL

Solution:

Assume  $L$  is CFL. Pick  $z = uvwxy = 0^n 1^n 2^n 3^n$  where  $|vwx| \leq n$  and  $vx \neq \epsilon$ .  $vwx$  can consist of a substring of one of the symbols or straddles of two adjacent symbols.

Case 1:  $vwx$  consists of a substring of one of the symbols

Then  $uwy$  has  $n$  of 3 different symbols and fewer than  $n$  of 4<sup>th</sup> symbol. Then  $uwy$  is not in  $L$ .

Case 2:  $vwx$  consists of 2 adjacent symbols say 1 & 2

Then  $uwy$  is missing some 1's or 2's and  $uwy$  is not in  $L$ .

If we consider any combinations of above cases, we get  $uwy$ , which is not CFL. This contradicts the assumption. So  $L$  is not a CFL.

**Exercises:**

Using pumping lemma for CFL prove that below languages are not context free

1.  $\{0^p \mid p \text{ is a prime}\}$
2.  $\{a^n b^n c^i \mid i \leq n\}$

# Closure Properties of CFL

Many operations on Context Free Languages (CFL) guarantee to produce CFL. A few do not produce CFL. *Closure properties* consider operations on CFL that are guaranteed to produce a CFL. The CFL's are closed under *substitution*, *union*, *concatenation*, *closure (star)*, *reversal*, *homomorphism* and *inverse homomorphism*. CFL's are not closed under *intersection* (but the intersection of a CFL and a regular language is always a CFL), *complementation*, and *set-difference*.

## **I. Substitution:**

By substitution operation, each symbol in the strings of one language is replaced by an entire CFL language.

### **Example:**

$S(0) = \{a^n b^n \mid n \geq 1\}$ ,  $S(1) = \{aa, bb\}$  is a substitution on alphabet  $\Sigma = \{0, 1\}$ .

### **Theorem:**

If a substitution  $s$  assigns a CFL to every symbol in the alphabet of a CFL  $L$ , then  $s(L)$  is a CFL.

Proof:

Let  $G = (V, \Sigma, P, S)$  be grammar for the CFL  $L$ . Let  $G_a = (V_a, T_a, P_a, S_a)$  be the grammar corresponding to each terminal  $a \in \Sigma$  and  $V \cap V_a = \emptyset$ . Then  $G' = (V', T', P', S)$  is a grammar for  $s(L)$  where

- $V' = V \cup V_a$
- $T' =$  union of  $T_a$ 's all for  $a \in \Sigma$
- $P'$  consists of
  - All productions in any  $P_a$  for  $a \in \Sigma$
  - The productions of  $P$ , with each terminal  $a$  is replaced by  $S_a$  everywhere  $a$  occurs.

### **Example:**

$L = \{0^n 1^n \mid n \geq 1\}$ , generated by the grammar  $S \rightarrow 0S1 \mid 01$ ,  $s(0) = \{a^n b^m \mid m \leq n\}$ , generated by the grammar  $S \rightarrow aSb \mid A$ ;  $A \rightarrow aA \mid ab$ ,  $s(1) = \{ab, abc\}$ , generated by the grammar  $S \rightarrow abA$ ,  $A \rightarrow c \mid \epsilon$ . Rename second and third  $S$ 's to  $S_0$  and  $S_1$ , respectively. Rename second  $A$  to  $B$ . Resulting grammars are:

$$S \rightarrow 0S1 \mid 01$$

$$S_0 \rightarrow aS_0b \mid A; A \rightarrow aA \mid ab$$

$$S_1 \rightarrow abB; B \rightarrow c \mid \varepsilon$$

In the first grammar replace 0 by  $S_0$  and 1 by  $S_1$ . The resulted grammar after substitution is:

$$S \rightarrow S_0SS_1 \mid S_0S_1 \quad S_0 \rightarrow aS_0b \mid A; A \rightarrow aA \mid ab \quad S_1 \rightarrow abB; B \rightarrow c \mid \varepsilon$$

## II. Application of substitution:

### a. Closure under union of CFL's $L_1$ and $L_2$ :

Use  $L = \{a, b\}$ ,  $s(a) = L_1$  and  $s(b) = L_2$ . Then  $s(L) = L_1 \cup L_2$ .

How to get grammar for  $L_1 \cup L_2$  ?

Add new start symbol  $S$  and rules  $S \rightarrow S_1 \mid S_2$

The grammar for  $L_1 \cup L_2$  is  $G = (V, T, P, S)$  where  $V = \{V_1 \cup V_2 \cup S\}$ ,  $S \notin (V_1 \cup V_2)$  and  $P = \{P_1 \cup P_2 \cup \{S \rightarrow S_1 \mid S_2\}\}$

#### Example:

$L_1 = \{a^n b^n \mid n \geq 0\}$ ,  $L_2 = \{b^n a^n \mid n \geq 0\}$ . Their corresponding grammars are

$$G_1: S_1 \rightarrow aS_1b \mid \varepsilon, G_2: S_2 \rightarrow bS_2a \mid \varepsilon$$

The grammar for  $L_1 \cup L_2$  is

$$G = (\{S, S_1, S_2\}, \{a, b\}, \{S \rightarrow S_1 \mid S_2, S_1 \rightarrow aS_1b \mid \varepsilon, S_2 \rightarrow bS_2a\}, S).$$

### b. Closure under concatenation of CFL's $L_1$ and $L_2$ :

Let  $L = \{ab\}$ ,  $s(a) = L_1$  and  $s(b) = L_2$ . Then  $s(L) = L_1L_2$

How to get grammar for  $L_1L_2$ ?

Add new start symbol and rule  $S \rightarrow S_1S_2$

The grammar for  $L_1L_2$  is  $G = (V, T, P, S)$  where  $V = V_1 \cup V_2 \cup \{S\}$ ,  $S \notin V_1 \cup V_2$  and  $P = P_1 \cup P_2 \cup \{S \rightarrow S_1S_2\}$

#### Example:

$L_1 = \{a^n b^n \mid n \geq 0\}$ ,  $L_2 = \{b^n a^n \mid n \geq 0\}$  then  $L_1 L_2 = \{a^n b^{n+m} a^m \mid n, m \geq 0\}$

Their corresponding grammars are

$$G_1: S_1 \rightarrow aS_1b \mid \varepsilon, G_2: S_2 \rightarrow bS_2a \mid \varepsilon$$

The grammar for  $L_1 L_2$  is

$$G = (\{S, S_1, S_2\}, \{a, b\}, \{S \rightarrow S_1 S_2, S_1 \rightarrow aS_1b \mid \varepsilon, S_2 \rightarrow bS_2a\}, S).$$

**c. Closure under Kleene's star (closure  $*$  and positive closure  $^+$ ) of CFL's  $L_1$ :**

Let  $L = \{a\}^*$  (or  $L = \{a\}^+$ ) and  $s(a) = L_1$ . Then  $s(L) = L_1^*$  (or  $s(L) = L_1^+$ ).

**Example:**

$$L_1 = \{a^n b^n \mid n \geq 0\} \quad (L_1)^* = \{a^{\{n_1\}} b^{\{n_1\}} \dots a^{\{n_k\}} b^{\{n_k\}} \mid k \geq 0 \text{ and } n_i \geq 0 \text{ for all } i\}$$

$$L_2 = \{a^{\{n^2\}} \mid n \geq 1\}, (L_2)^* = a^*$$

How to get grammar for  $(L_1)^*$ :

Add new start symbol  $S$  and rules  $S \rightarrow SS_1 \mid \varepsilon$ .

The grammar for  $(L_1)^*$  is

$$G = (V, T, P, S), \text{ where } V = V_1 \cup \{S\}, S \notin V_1, P = P_1 \cup \{S \rightarrow SS_1 \mid \varepsilon\}$$

**d. Closure under homomorphism of CFL  $L_i$  for every  $a_i \in \Sigma$ :**

Suppose  $L$  is a CFL over alphabet  $\Sigma$  and  $h$  is a homomorphism on  $\Sigma$ . Let  $s$  be a substitution that replaces every  $a \in \Sigma$ , by  $h(a)$ . ie  $s(a) = \{h(a)\}$ . Then  $h(L) = s(L)$ . ie  $h(L) = \{h(a_1) \dots h(a_k) \mid k \geq 0\}$  where  $h(a_i)$  is a homomorphism for every  $a_i \in \Sigma$ .

**III. Closure under Reversal:**

$L$  is a CFL, so  $L^R$  is a CFL. It is enough to reverse each production of a CFL for  $L$ , i.e., to substitute each production  $A \rightarrow \alpha$  by  $A \rightarrow \alpha^R$ .

**IV. Intersection:**

The CFL's are not closed under intersection

**Example:**

The language  $L = \{0^n 1^n 2^n \mid n \geq 1\}$  is not context-free. But  $L_1 = \{0^n 1^n 2^i \mid n \geq 1, i \geq 1\}$  is a CFL and  $L_2 = \{0^i 1^n 2^n \mid n \geq 1, i \geq 1\}$  is also a CFL. But  $L = L_1 \cap L_2$ .

Corresponding grammars for  $L_1$ :  $S \rightarrow AB$ ;  $A \rightarrow 0A1 \mid 01$ ;  $B \rightarrow 2B \mid 2$  and corresponding grammars for  $L_2$ :  $S \rightarrow AB$ ;  $A \rightarrow 0A \mid 0$ ;  $B \rightarrow 1B2 \mid 12$ .

However,  $L = L_1 \cap L_2$ , thus intersection of CFL's is not CFL

### a. Intersection of CFL and Regular Language:

**Theorem:** If  $L$  is CFL and  $R$  is a regular language, then  $L \cap R$  is a CFL.

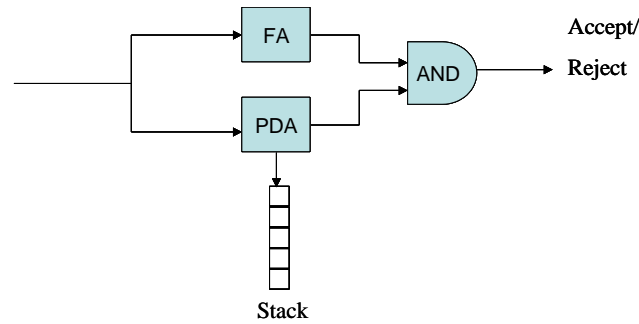


Figure 1: PDA for  $L \cap R$

Proof:

$P = (Q_p, \Sigma, \Gamma, \delta_p, q_p, Z_0, F_p)$  be PDA to accept  $L$  by final state. Let  $A = (Q_A, \Sigma, \delta_A, q_A, F_A)$  for DFA to accept the Regular Language  $R$ . To get  $L \cap R$ , we have to run a Finite Automata in parallel with a push down automata as shown in figure 1. Construct PDA  $P' = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$  where

- $Q = (Q_p \times Q_A)$
- $q_0 = (q_p, q_A)$
- $F = (F_p \times F_A)$
- $\delta$  is in the form  $\delta((q, p), a, X) = ((r, s), g)$  such that
  1.  $s = \delta_A(p, a)$
  2.  $(r, g)$  is in  $\delta_p(q, a, X)$

That is for each move of PDA  $P$ , we make the same move in PDA  $P'$  and also we carry along the state of DFA  $A$  in a second component of  $P'$ .  $P'$  accepts a string  $w$  if and only if both  $P$  and  $A$  accept  $w$ . ie  $w$  is in  $L \cap R$ . The moves  $((q_p, q_A), w, Z) \vdash^* P' ((q, p), \epsilon, \gamma)$  are possible if and only if  $(q_p, w, Z) \vdash^* P (q, \epsilon, \gamma)$  moves and  $p = \delta^*(q_A, w)$  transitions are possible.

### b. CFL and RL properties:

**Theorem:** The following are true about CFL's  $L$ ,  $L_1$ , and  $L_2$ , and a regular language  $R$ .

1. **Closure of CFL's under set-difference with a regular language.** ie  $L - R$  is a CFL.

Proof:

$R$  is regular and regular language is closed under complement. So  $R^C$  is also regular. We know that  $L - R = L \cap R^C$ . We have already proved the closure of intersection of a CFL and a regular language. So CFL is closed under set difference with a Regular language.

2. **CFL is not closed under complementation**

$L^C$  is not necessarily a CFL

Proof:

Assume that CFLs were closed under complement. ie if  $L$  is a CFL then  $L^C$  is a CFL. Since CFLs are closed under union,  $L_1^C \cup L_2^C$  is a CFL. By our assumption  $(L_1^C \cup L_2^C)^C$  is a CFL. But  $(L_1^C \cup L_2^C)^C = L_1 \cap L_2$ , which we just showed isn't necessarily a CFL. Contradiction! . So our assumption is false. CFL is not closed under complementation.

3. **CFLs are not closed under set-difference.** ie  $L_1 - L_2$  is not necessarily a CFL.

Proof:

Let  $L_1 = \Sigma^* - L$ .  $\Sigma^*$  is regular and is also CFL. But  $\Sigma^* - L = L^C$ . If CFLs were closed under set difference, then  $\Sigma^* - L = L^C$  would always be a CFL. But CFL's are not closed under complementation. So CFLs are not closed under set-difference.

## V. Inverse Homomorphism:

Recall that if  $h$  is a homomorphism, and  $L$  is any language, then  $h^{-1}(L)$ , called an *inverse homomorphism*, is the set of all strings  $w$  such that  $h(w) \in L$ . The CFL's are closed under inverse homomorphism.

### Theorem:

If  $L$  is a CFL and  $h$  is a homomorphism, then  $h^{-1}(L)$  is a CFL



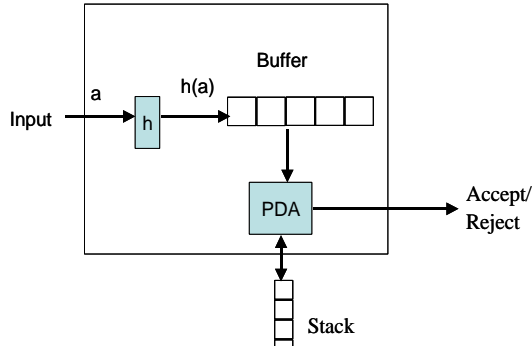


Figure 2: PDA to simulate inverse homomorphism

We can prove closure of CFL under inverse homomorphism by designing a new PDA as shown in figure 2. After input  $a$  is read,  $h(a)$  is placed in a buffer. Symbols of  $h(a)$  are used one at a time and fed to PDA being simulated. Only when the buffer is empty does the PDA read another of its input symbol and apply homomorphism to it.

Suppose  $h$  applies to symbols of alphabet  $\Sigma$  and produces strings in  $T^*$ . Let PDA  $P = (Q, T, \Gamma, \delta, q_0, Z_0, F)$  that accept CFL  $L$  by final state. We construct a new PDA  $P' = (Q', \Sigma, \Gamma, \delta', (q_0, \varepsilon), Z_0, (F \times \varepsilon))$  to accept  $h^{-1}(L)$ , where

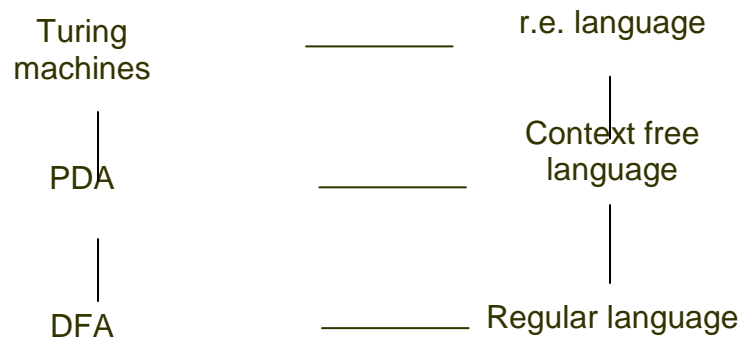
- $Q'$  is the set of pairs  $(q, x)$  such that
  - $q$  is a state in  $Q$
  - $x$  is a suffix of some string  $h(a)$  for some input string  $a$  in  $\Sigma$
- $\delta'$  is defined by
  - $\delta'((q, \varepsilon), a, X) = \{(q, h(a)), X\}$
  - If  $\delta(q, b, X) = \{(p, \gamma)\}$  where  $b \in T$  or  $b = \varepsilon$  then  $\delta'((q, bx), \varepsilon, X) = \{(p, x), \gamma\}$
- The start state of  $P'$  is  $(q_0, \varepsilon)$
- The accepting state of  $P'$  is  $(q, \varepsilon)$ , where  $q$  is an accepting state of  $P$ .

Once we accept the relationship between  $P$  and  $P'$ ,  $P$  accepts  $h(w)$  if and only if  $P'$  accepts  $w$ , because of the way the accepting states of  $P'$  are defined.

Thus  $L(P') = h^{-1}(L(P))$

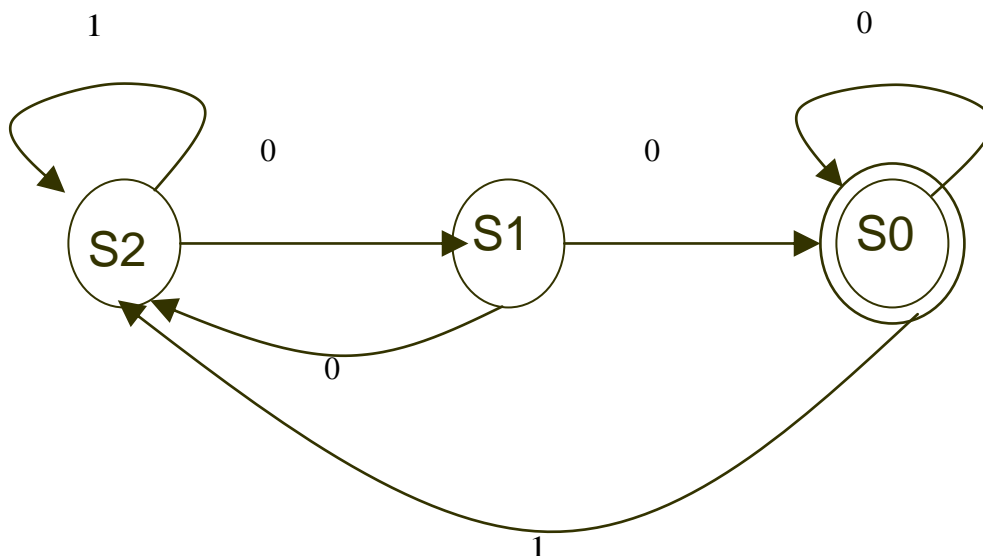
# Language Hierarchy and History of Turing Machine

## A Hierarchy of Formal Languages



## Regular language

A language is called a regular language if some finite automaton recognizes it.  
Ex: To recognize string that is a multiple of 4



But can regular language recognize strings of the form  $0^n1^n$  ? **No**

## Context Free Language

A language is called Context free language iff some pushdown automaton recognizes it.

Ex: To recognize a string of the form  $0^n1^n$

$$S \rightarrow 0S1 / \lambda$$

Limitation again

**Can context free language recognize strings of the form  $0^n1^n2^n$  ? No**

**Recursively Enumerable Language (r.e. language)**

A language is called r.e language if some Turing machine recognizes it.

Ex: To recognize strings of the form  $0^n1^n2^n$

$$\begin{aligned}(S_0, 0) &\rightarrow (S_1, X, R) \\(S_1, 0) &\rightarrow (S_1, 0, R) \\(S_1, 1) &\rightarrow (S_2, Y, R) \\(S_2, 1) &\rightarrow (S_2, 1, R) \\(S_2, 2) &\rightarrow (S_3, Z, L) \\(S_3, 1) &\rightarrow (S_3, 1, L) \\(S_3, Y) &\rightarrow (S_3, Y, L) \\(S_3, 0) &\rightarrow (S_3, 0, L) \\(S_3, X) &\rightarrow (S_0, X, R) \\(S_0, Y) &\rightarrow (S_A, Y, R)\end{aligned}$$

$S_A$  : Accepting state

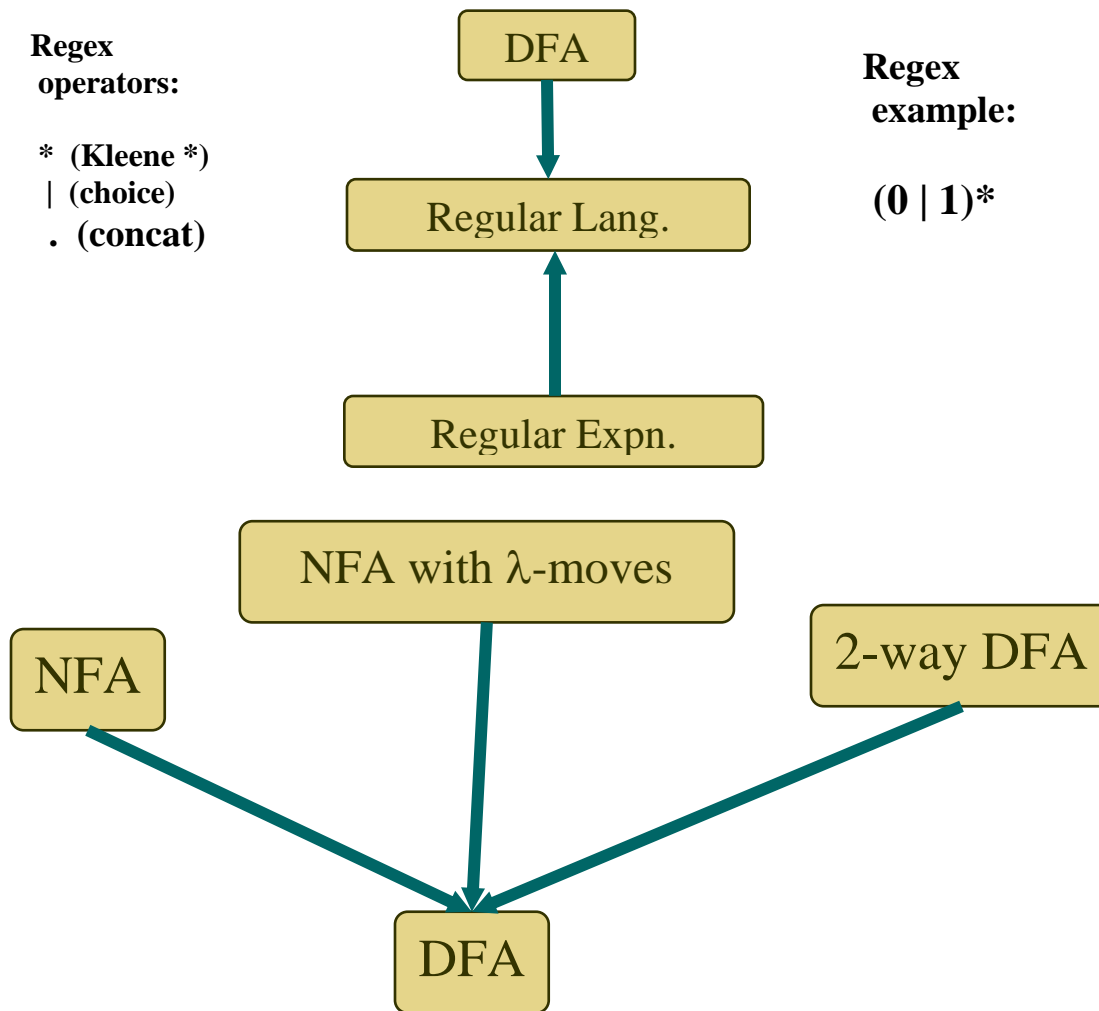
## Formal Machines Overview

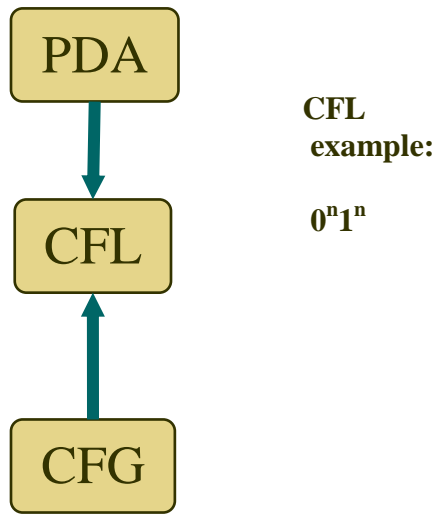
**Regex  
operators:**

\* (Kleene \*)  
| (choice)  
· (concat)

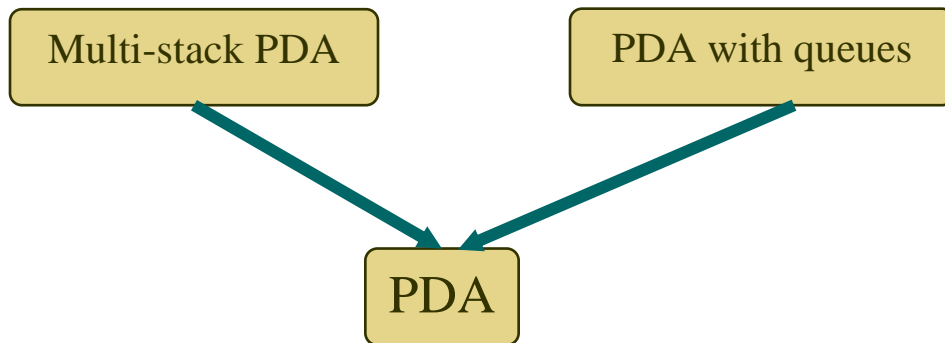
**Regex  
example:**

**(0 | 1)\***





But there are strings that cannot be recognized by PDAs. For example:  $a^n b^n c^n$



### Some Historical Notes

At the turn of the 20<sup>th</sup> century the German mathematician David Hilbert proposed the *Entscheidungsproblem*. Given a set  $X$ , and a universe of elements  $U$ , and a criteria for membership, is it possible to formulate a *general* procedure to decide whether a given element of  $U$  is a member of  $X$ ?

Hilbert's grand ideas were watered down by the *incompleteness theorem* proposed by the Austrian Kurt Gödel. His theorem states that in any mathematical system, there exist certain obviously true assertions that cannot be *proved* to be true by the system.

In 1936 the British cryptologist Alan Turing, addressed Hilbert's *Entscheidungsproblem* using a different approach. He proposed two kinds of mathematical

machines called the *a-machine* and the *c-machine* respectively, and showed the existence of some problems where membership cannot be determined.

But Turing's *a-machine* became famous for something else other than its original intentions. The *a-machine* was found to be *more expressive* than CFGs. It can recognize strings that cannot be modeled by CFLs like  $a^n b^n c^n$ . The *a-machines* came to be more popularly known as Turing Machines

The Princeton mathematician Alonzo Church recognized the power of *a-machines*. He invited Turing to Princeton to compare Turing Machines with his own  $\lambda$ -calculus.

Church and Turing proved the equivalence of Turing Machines and  $\lambda$ -calculus, and showed that they represent algorithmic computation. This is called the Church-Turing thesis.

## Turing Machines

### Definition:

A Turing Machine (TM) is an abstract, mathematical model that describes what can and cannot be computed. A Turing Machine consists of a tape of infinite length, on which input is provided as a finite sequence of symbols. A *head* reads the input tape. The Turing Machine starts at "start state"  $S_0$ . On reading an input symbol it optionally replaces it with another symbol, changes its internal state and moves one cell to the right or left.

### Notation for the Turing Machine :

TM =  $\langle S, T, S_0, \delta, H \rangle$  where,

$S$  is a set of TM states

$T$  is a set of tape symbols

$S_0$  is the start state

$H \subset S$  is a set of halting states

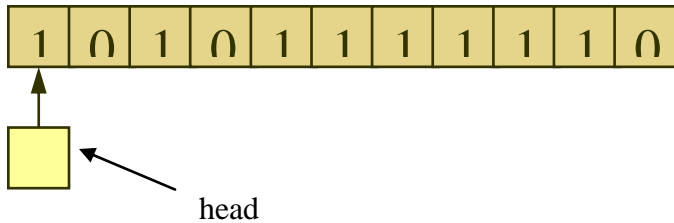
$\delta : S \times T \rightarrow S \times T \times \{L, R\}$   
is the transition function

$\{L, R\}$  is direction in which the head moves

L : Left      R: Right

input symbols on infinite length tape





The Turing machine model uses an infinite tape as its unlimited memory. (This is important because it helps to show that there are tasks that these machines cannot perform, even though unlimited memory and unlimited time is given.) The input symbols occupy some of the tape's cells, and other cells contain blank symbols.

Some of the characteristics of a Turing machine are:

1. The symbols can be both read from the tape and written on it.
2. The TM head can move in either directions – Left or Right.
3. The tape is of infinite length
4. The special states, Halting states and Accepting states, take immediate effect.

### Solved examples:

#### TM Example 1:

Turing Machine U+1:

Given a string of 1s on a tape (followed by an infinite number of 0s), add one more 1 at the end of the string.

Input : #111100000000.....



Output : #1111100000000.....

Initially the TM is in Start state  $S_0$ . Move right as long as the input symbol is 1. When a 0 is encountered, replace it with 1 and halt.

Transitions:

$$\begin{aligned} (S_0, 1) &\rightarrow (S_0, 1, R) \\ (S_0, 0) &\rightarrow (h, 1, \text{STOP}) \end{aligned}$$

#### TM Example 2 :

TM: X-Y

Given two unary numbers  $x$  and  $y$ , compute  $|x-y|$  using a TM. For purposes of simplicity we shall be using multiple tape symbols.

Ex:  $5(11111) - 3(111) = 2(11)$

#11111b1110000..... →  
#\_\_\_11b\_\_\_000...

a) Stamp out the first 1 of x and seek the first 1 of y.

$(S_0, 1) \rightarrow (S_1, \_, R)$   
 $(S_0, b) \rightarrow (h, b, STOP)$   
 $(S_1, 1) \rightarrow (S_1, 1, R)$   
 $(S_1, b) \rightarrow (S_2, b, R)$

b) Once the first 1 of y is reached, stamp it out. If instead the input ends, then y has finished. But in x, we have stamped out one extra 1, which we should replace. So, go to some state s5 which can handle this.

$(S_2, 1) \rightarrow (S_3, \_, L)$   
 $(S_2, \_) \rightarrow (S_2, \_, R)$   
 $(S_2, 0) \rightarrow (S_5, 0, L)$

c) State s3 is when corresponding 1s from both x and y have been stamped out. Now go back to x to find the next 1 to stamp. While searching for the next 1 from x, if we reach the head of tape, then stop.

$(S_3, \_) \rightarrow (S_3, \_, L)$   
 $(S_3, b) \rightarrow (S_4, b, L)$   
 $(S_4, 1) \rightarrow (S_4, 1, L)$   
 $(S_4, \_) \rightarrow (S_0, \_, R)$   
 $(S_4, \#) \rightarrow (h, \#, STOP)$

d) State s5 is when y ended while we were looking for a 1 to stamp. This means we have stamped out one extra 1 in x. So, go back to x, and replace the blank character with 1 and stop the process.

$(S_5, \_) \rightarrow (S_5, \_, L)$   
 $(S_5, b) \rightarrow (S_6, b, L)$   
 $(S_6, 1) \rightarrow (S_6, 1, L)$   
 $(S_6, \_) \rightarrow (h, 1, STOP)$



## Design of Turing Machines

and

## Universal Turing Machine

### Solved examples:

TM Example 1: Design a Turing Machine to recognize  $0^n 1^n 2^n$   
ex: #000111222\_ \_ \_ \_ \_ .....

Step 1: Stamp the first 0 with X, then seek the first 1 and stamp it with Y, and then seek the first 2 and stamp it with Z and then move left.

$$\begin{aligned}(S_0, 0) &\rightarrow (S_1, X, R) \\ (S_1, 0) &\rightarrow (S_1, 0, R) \\ (S_1, 1) &\rightarrow (S_2, Y, R) \\ (S_2, 1) &\rightarrow (S_2, 1, R) \\ (S_2, 2) &\rightarrow (S_3, Z, L)\end{aligned}$$

$S_0$  = Start State, seeking 0, stamp it with X

$S_1$  = Seeking 1, stamp it with Y

$S_2$  = Seeking 2, stamp it with Z

Step 2: Move left until an X is reached, then move one step right.

$$\begin{aligned}(S_3, 1) &\rightarrow (S_3, 1, L) \\ (S_3, Y) &\rightarrow (S_3, Y, L) \\ (S_3, 0) &\rightarrow (S_3, 0, L) \\ (S_3, X) &\rightarrow (S_0, X, R)\end{aligned}$$

$S_3$  = Seeking X, to repeat the process.

Step 3: Move right until the end of the input denoted by blank( \_ ) is reached passing through X Y Z s only, then the accepting state  $S_A$  is reached.

$$\begin{aligned}(S_0, Y) &\rightarrow (S_4, Y, R) \\ (S_4, Y) &\rightarrow (S_4, Y, R) \\ (S_4, Z) &\rightarrow (S_4, Z, R) \\ (S_4, \_) &\rightarrow (S_A, \_, STOP)\end{aligned}$$

S4 = Seeking blank

These are the transitions that result in halting states.

$$\begin{aligned}(S_4,1) &\rightarrow (h,1,STOP) \\ (S_4,2) &\rightarrow (h,2,STOP) \\ (S_4,\_) &\rightarrow (S_A,\_,STOP) \\ (S_0,1) &\rightarrow (h,1,STOP) \\ (S_0,2) &\rightarrow (h,2,STOP) \\ (S_1,2) &\rightarrow (h,2,STOP) \\ (S_2,\_) &\rightarrow (h,\_,STOP)\end{aligned}$$

TM Example 2 : Design a Turing machine to accept a Palindrome

ex: #1011101 \_ \_ \_ \_ \_ .....

Step 1: Stamp the first character (0/1) with  $\_$ , then seek the last character by moving till a  $\_$  is reached. If the last character is not 0/1 (as required) then halt the process immediately.

$$\begin{aligned}(S_0,0) &\rightarrow (S_1,\_,R) \\ (S_0,1) &\rightarrow (S_2,\_,R) \\ (S_1,\_) &\rightarrow (S_3,\_,L) \\ (S_3,1) &\rightarrow (h,1,STOP) \\ (S_2,\_) &\rightarrow (S_5,\_,L) \\ (S_5,0) &\rightarrow (h,0,STOP)\end{aligned}$$

Step 2: If the last character is 0/1 accordingly, then move left until a blank is reached to start the process again.

$$\begin{aligned}(S_3,0) &\rightarrow (S_4,\_,L) \\ (S_4,1) &\rightarrow (S_4,1,L) \\ (S_4,0) &\rightarrow (S_4,0,L) \\ (S_4,\_) &\rightarrow (S_0,\_,R) \\ (S_5,1) &\rightarrow (S_6,\_,L) \\ (S_6,1) &\rightarrow (S_6,1,L) \\ (S_6,0) &\rightarrow (S_6,0,L) \\ (S_6,\_) &\rightarrow (S_0,\_,R)\end{aligned}$$

Step 3 : If a blank (  $\_$  ) is reached when seeking next pair of characters to match or when seeking a matching character, then accepting state is reached.

$$\begin{aligned}
(S_3, \_) &\rightarrow (S_A, \_, STOP) \\
(S_5, \_) &\rightarrow (S_A, \_, STOP) \\
(S_0, \_) &\rightarrow (S_A, \_, STOP)
\end{aligned}$$

The sequence of events for the above given input are as follows:

```

#s010101_ _ _
    #_s20101_ _ _
        #_0s2101_ _ _
            . . . .
            #_0101s5_ _ _
                #_010s6_ _ _ _
                    #_s60101_ _ _
                        #_s00101_ _ _
                            . . . .
                            #_ _ _ _ s5 _ _ _ _ _
                                #_ _ _ _ sA _ _ _ _ _

```

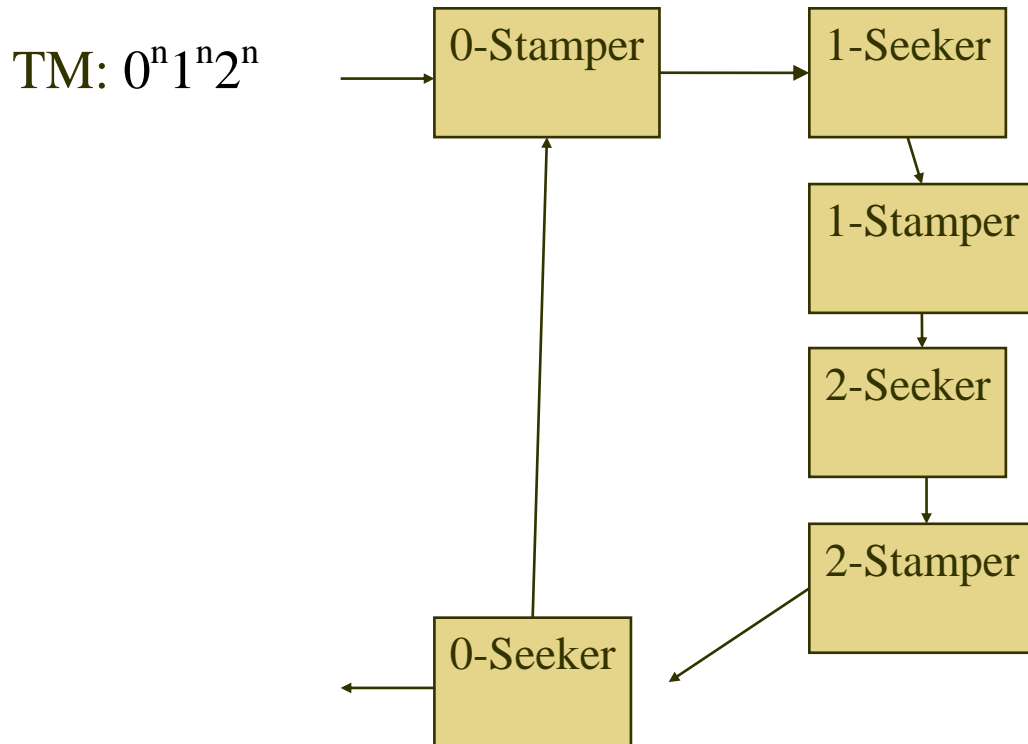
### Exercises:

1. Design a TM to recognize a string of the form  $a^n b^{2n}$ .
2. Design a TM to recognize a string of 0s and 1s such that the number of 0s is not twice as that of 1s.

### Modularization of TMs

Designing complex TM s can be done using modular approach. The main problem can be divided into sequence of modules. Inside each module, there could be several state transitions.

For example, the problem of designing Turing machine to recognize the language  $0^n 1^n 2^n$  can be divided into modules such as 0-stamper, 1-stamper, 0-seeker, 1-seeker, 2-seeker and 2-stamper. The associations between the modules are shown in the following figure:



### Universal Turing Machine

A Universal Turing Machine UTM takes an encoding of a TM and the input data as its input in its tape and behaves as that TM on the input data.

A TM spec could be as follows:

$TM = (S, S_0, H, T, d)$

Suppose,  $S = \{a, b, c, d\}$ ,  $S_0 = a$ ,  $H = \{b, d\}$ ,  $T = \{0, 1\}$

$\delta : (a, 0) \rightarrow (b, 1, R), (a, 1) \rightarrow (c, 1, R),$   
 $(c, 0) \rightarrow (d, 0, R)$  and so on

then TM spec:

$\$abcd\$a\$bd\$01\$a0b1Ra1c1Rc0d0R\ldots\ldots$

where  $\$$  is delimiter

This spec along with the actual input data would be the input to the UTM.

This can be encoded in binary by assigning numbers to each of the characters appearing in the TM spec.

The encoding can be as follows:

\$ : 0000	0 : 0101
a : 0001	1 : 0110
b : 0010	L : 0111
c : 0011	R : 1000
d : 0100	

So the TM spec given in previous slide can be encoded as:

0000.0001.0010.0011.0100.0000.0001.0000.0010.0100 .....

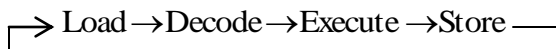
Hence TM spec can be regarded just as a number.

Sequence of actions in UTM:

Initially UTM is in the start state S0.

- Load the input which is TM spec.
- Go back and find which transition to apply.
- Make changes, where necessary.
- Then store the changes.
- Then repeat the steps with next input.

Hence, the sequence goes through the cycle:



## Extensions to Turing Machines

### Proving Equivalence

For any two machines  $M_1$  from class  $C_1$  and  $M_2$  from class  $C_2$ :

$M_2$  is said to be at least as expressive as  $M_1$   
if  $L(M_2) = L(M_1)$  or if  $M_2$  can *simulate*  $M_1$ .

$M_1$  is said to be at least as expressive as  $M_2$   
if  $L(M_1) = L(M_2)$  or if  $M_1$  can simulate  $M_2$ .

### Composite Tape TMs

Track 0	0	1	1	0	1	0	1	0	0	...
Track 1	0	0	1	1	1	1	1	1	0	...

A composite tape consists of many *tracks* which can be read or written *simultaneously*.  
A composite tape TM (CTM) contains more than one tracks in its tape.

### Equivalence of CTMs and TMs

A CTM is simply a TM with a complex alphabet..

$T = \{a, b, c, d\}$

$T' = \{00, 01, 10, 11\}$

### Turing Machines with Stay Option

Turing Machines with stay option has a third option for movement of the TM head:  
left, right or *stay*.

$STM = \langle S, T, \delta, s_0, H \rangle$

$\delta: S \times T \rightarrow S \times T \times \{L, R, S\}$

### Equivalence of STMs and TMs

STM = TM:

Just don't use the S option...

TM = STM:

For L and R moves of a given STM build a TM that moves correspondingly L or R...

TM = STM:

For S moves of the STM, do the following:

1. Move right,
2. Move back left without changing the tape

3.STM:  $\delta(s,a) \dashv (s',b,S)$   
 TM:  $\delta(s,a) \dashv (s'', b, R)$   
 $\delta(s'',*) \dashv (s',*,L)$

## 2-way Infinite Turing Machine

In a 2-way infinite TM (2TM), the tape is infinite on both sides.  
 There is no # that delimits the left end of the tape.

## Equivalence of 2TMs and TMs

2TM = TM:

Just don't use the left part of the tape...

TM = 2TM:

Simulate a 2-way infinite tape on a one-way infinite tape...

... -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 ...

0 -1 1 -2 2 -3 3 -4 4 -5 5 ...

## Multi-tape Turing Machines

A multi-tape TM (MTM) utilizes many tapes.



## Equivalence of MTMs and TMs

MTM = TM:

Use just the first tape...

TM = MTM:

Reduction of multiple tapes to a single tape.

Consider an MTM having  $m$  tapes. A single tape TM that is equivalent can be constructed by reducing  $m$  tapes to a single tape.

A    0 1 2 3 4 5 6 7 ...

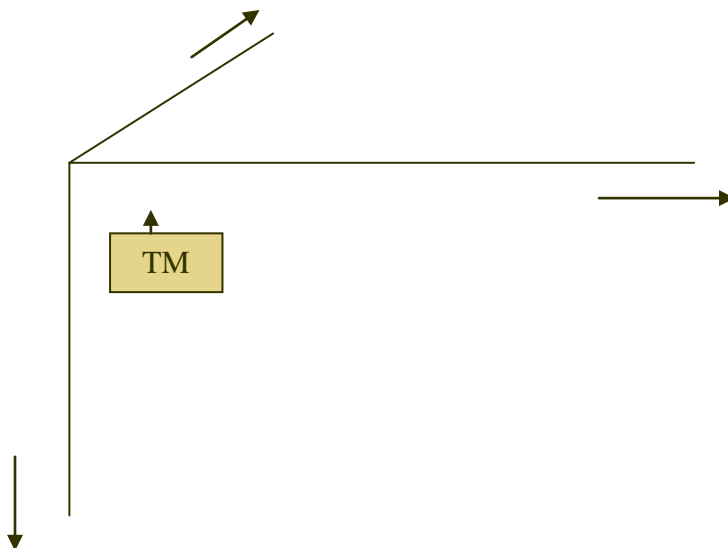
B    0 1 2 3 4 5 6 7 ...

C    0 1 2 3 4 5 6 7 ...

TM    A0 B0 C0 A1 B1 C1 A2 B2 C2 A3 B3 ..

## Multi-dimensional TMs

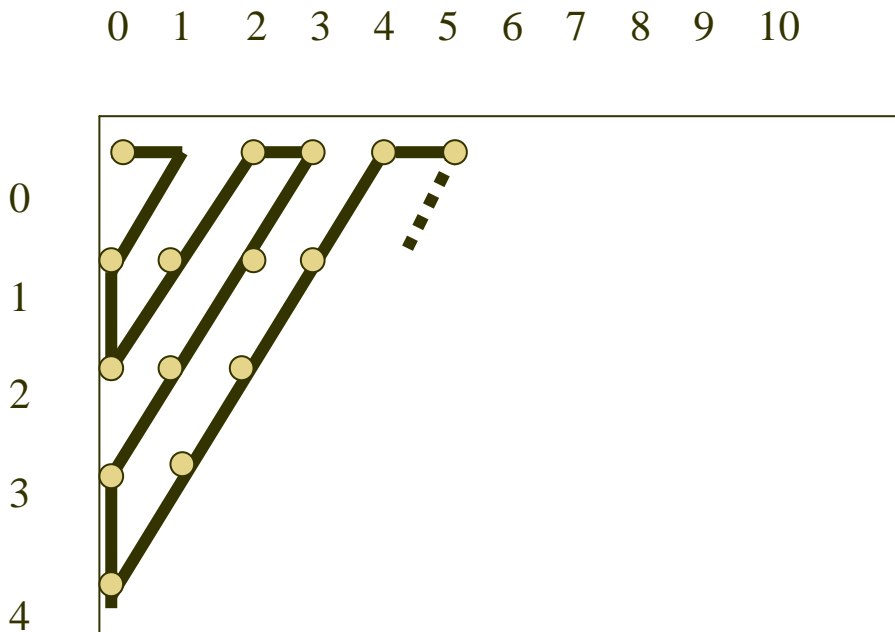
Multi-dimensional TMs (MDTMs) use a multi-dimensional space instead of a single dimensional tape.





### Equivalence of MDTMs to TMs

Reducing a multi-dimensional space to a single dimensional tape.



### Non-deterministic TM

A non-deterministic TM (NTM) is defined as:

$$\text{NTM} = \langle S, T, s_0, \delta, H \rangle$$

where  $\delta: S \times T \rightarrow 2^{S \times T \times \{L, R\}}$

Ex:  $(s_2, a) \rightarrow \{(s_3, b, L), (s_4, a, R)\}$

### Equivalence of NTMs and TMs

A “concurrent” view of an NTM:

$$(s_2, a) \rightarrow \{(s_3, b, L), (s_4, a, R)\}$$

$\Rightarrow$  at  $(s_2, a)$ , two TMs are spawned:

$$(s_2, a) \rightarrow (s_3, b, L)$$

$$(s_2, a) \rightarrow (s_4, a, R)$$

### Simulating an NTM with an MDTM

Consider an MDTM to simulate an NTM:  
 $(s_2, a) \rightarrow \{(s_3, b, L) (s_4, a, R)\}$

bccaaabccacb

$s_2$



bccbaabccacb

$s_3$

bccaaabccacb

$s_4$