University of Technology
الجامعة التكنولوجية
Computer Science Department
قسم علوم الحاسوب

Prof. Dr.Abeer Tariq Maolood
Prof. Alaa Nori Mezher

Computation Theory

Third Class/ first course /All Branches

2024-2025

cs.uotechnology.edu.iq

الاسبوع الاول

1. Regular Expression

الاسبوع الثاني

2. Finite Automata, DFA and NFA,

الاسبوع الثالث

3. Equivalence of NFA and DFA

الاسبوع الرابع

4. Introduction to moor and mealy machine.

الاسبوع الخامس

5. The equivalence of mealy and moor machine

الاسبوع السادس

6. Introduction to Crammers, Phrase Structure Grammar

الاسبوع السابع

7. Context sensitive Grammar, Context Free grammar,

الاسبوع الثامن

8. Chomsky Normal Form

الاسبوع التاسع

9. CNF different Examples

الاسبوع العاشر

10. Tree, leftmost and rightmost derivations

الاسبوع الحادي عشر

11. Regular grammar, Left linear grammar, Right linear grammar

الاسبوع الثاني عشر

12. Push down automata

الاسبوع الثالث عشر

13. Top down –bottom up derivation

الاسبوع الرابع عشر

14. Turing machine

Regular Expression

The language-defining symbols we are about to create are called regular expressions. The languages that are associated with these regular expressions are called regular languages.

Example consider the language L
where L={Λ x xx xxx …} by using star notation we may write
L=language(x*).
Since x* is any string of x's (including Λ).

Example if we have the alphabet ∑={a,b}
And L={a ab abb abbb abbbb …}
Then L=language(ab*)

Example (ab)*= Λ or ab or abab or ababab or abababab or ….

Example L1=language(xx*)
The language L1 can be defined by any of the expressions:
xx* or x^+ or xx*x* or x*xx* or x^+x* or x*x^+ or x*x*x*xx* …
Remember x* can always be Λ.

Example language(ab*a)={aa aba abba abbba abbbba …}

Example language(a*b*)={ Λ a b aa ab bb aaa aab abb bbb … } ba and aba are not in this language so a*b* ≠ (ab)*

Example the following expressions both define the language
L2={x^{odd}}: x(xx)* or (xx)*x
But the expression x*xx* does not since it includes the word (xx)x(x).

Example consider the language T defined over the alphabet ∑={a,b,c}
T={a c ab cb abb cbb abbb cbbb abbbb cbbbb …}
Then T=language((a+c)b*)
T=language(either a or c then some b's)

Example consider a finite language L that contains all the strings of a's and b's of length exactly three.
L={aaa aab aba abb baa bab bba bbb}
L=language((a+b)(a+b)(a+b))
L=language($(a+b)^3$)

Note from the alphabet ∑={a,b} , if we want to refer to the set of all possible strings of a's and b's of any length (including Λ) we could write **(a+b)***

<u>*Example*</u> we can describe all words that begins with a and end with b with the expression *a(a+b)*b* which mean *a(arbitrary string)b*

<u>*Example*</u> if we have the expression *(a+b)*a(a+b)** then the word abbaab can be considerd to be of this form in three ways: (Λ)a(bbaab) or (abb)a(ab) or (abba)a(b)

<u>*Example*</u> (a+b)*a(a+b)*a(a+b)*
= (some beginning)(the first important a)(some middle)(the second important a)(some end)

Another expressions that denote all the words with at least two a's are:
b*ab*a(a+b)*, (a+b)*ab*ab*, b*a(a+b)*ab*

Then we could write:
language((a+b)*a(a+b)*a(a+b)*)
=language(b*ab*a(a+b)*)
=language((a+b)*ab*ab*)
=language(b*a(a+b)*ab*)
=all words with at least two a's.

Note: we say that two regular expressions are equivalent if they describe the same language.

<u>*Example*</u> if we want all the words with exactly two a's, we could use the expression: *b*ab*ab** which describe such words as aab, baba, bbbabbabbbb,…

<u>*Example*</u> the language of all words that have at least one a and at least one b is: *(a+b)*a(a+b)*b(a+b)*+(a+b)*b(a+b)*a(a+b)**

Note: (a+b)*b(a+b)*a(a+b)* ≠ bb*aa* since the left includes the word aba, which the expression on the right side does not.

Note: (a+b)* = (a+b)* + (a+b)*
(a+b)* = (a+b)*(a+b)*
(a+b)* = a(a+b)* + b(a+b)* + Λ
(a+b)* = (a+b)*ab(a+b)* + b*a*

Note: usually when we employ the star operation we are defining an infinite language. We can represent a finite language by using the plus alone.

Example L={abba baaa bbbb}
L=language(abba + baaa + bbbb)

Example L={ Λ a aa bbb}
L=language(Λ + a + aa + bbb)

Example L={Λ a b ab bb abb bbb abbb bbbb …}
We can define L by using the expression $b^* + ab^*$

Definition

The set of regular expressions is defined by the following rules:
Rule1: every letter of ∑ can be made into a regular expression, Λ is a regular expression.
Rule2: if r1 and r2 are regular expressions, then so are: (r1) r1r2 r1+r2 r1*.
Rule3: nothing else is a regular expression.

Remember that $r1^{+}=r1r1^*$

Definition

If S and T are sets of strings of letters (whether they are finite or infinite sets), we define the product set of strings of letters to be:
ST={all combination of a string from S concatenated with a string from T}

Example if S={a aa aaa} T={bb bbb}
Then ST={abb abbb aabb aabbb aaabb aaabbb}
(a+aa+aaa)(bb+bbb)=abb+abbb+ aabb+aabbb+aaabb+aaabbb)

Example if P={a bb bab} Q={Λ bbbb}
Then PQ={a bb bab abbbb bbbbbb babbbbb}
$(a+bb+bab)(\Lambda+bbbb)=a+bb+bab+ab^4+b^6+bab^5$

Example if M={Λ x xx} N={Λ y yy yyy yyyy …}
Then MN={Λ y yy yyy yyyy …
x xy xyy xyyy xyyyy …
xx xxy xxyy xxyyy xxyyyy …}
Using regular expression we could write:
(Λ+x+xx)(y*)=y*+xy*+xxy*

Definition

The following rules define the language associated with any regular expression.

Rule1: the language associated with the regular expression that is just a single letter is that one-letter word alone and the language associated with Λ is just{Λ}, a one-word language.

Rule2: if r1 is regular expression associated with the language L1 and r2 is regular expression associated with the language L2 then:

i) The regular expression (r1)(r2) is associated with the language L1 times L2.

Language(r1r2)=L1L2

ii) The regular expression r1+r2 is associated with the language formed by the union of the sets L1 and L2.

Language(r1+r2)=L1+L2

iii) The language associated with the regular expression (r1)* is L1*, the kleene closure of the set L1 as a set of words.

Language(r1)*=L1*

Example L={baa abba bababa}
The regular expression for this language is: (baa+abba+bababa)

Example L={Λ x xx xxx xxxx xxxxx}
The regular expression for this language is: (Λ+x+xx+xxx+xxxx+xxxxx)
$=(\Lambda+x)^5$

Example L= language((a+b)*(aa+bb)(a+b)*)
=(arbitrary)(double letter)(arbitrary)
{Λ a b ab ba aba bab abab baba …} these words are not included in L but they included by the regular expression: (Λ+b)(ab)*(Λ+a)

Example

E=(a+b)*a(a+b)*(a+Λ)(a+b)*a(a+b)*
E=(a+b)*a(a+b)*a(a+b)*a(a+b)*+(a+b)*a(a+b)*Λ(a+b)*a(a+b)*

We have: (a+b)*Λ(a+b)*=(a+b)*
Then: E=(a+b)*a(a+b)*a(a+b)*a(a+b)*+(a+b)*a(a+b)*a(a+b)*
The language associated with E is not different from the language associated with: (a+b)*a(a+b)*a(a+b)* **Note**:
(a+b*)*=(a+b)*
(a*)*=a*
(aa+ab*)*≠ (aa+ab)*
(a*b*)*=(a+b)*

Example E=[aa+bb+(ab+ba)(aa+bb)*(ab+ba)]* Even-even={Λ aa bb aabb abab abba baab baba bbaa aaaabb aaabab

Finite Automata (FA)

A finite automata is a collection of five things:

1. A finite set of states
2. An alphabet $\sum$ of possible input letters from which are formed strings that are to be read one letter at a time.
3. A finite set of transitions that tell for each state and for each letter of the input alphabet which state to go to next.
4. The initial state; and
5. The set of final states.
6. Therefore formally a finite automa
ta is a five-tuple:

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

where:

- Q is a set of states of the finite automata,
- Σ is a set of input symbols, and
- δ specifies the transitions in the automata.

If from a state *p* there exists a transition going to state *q* on an input symbol *a*, then we write $\delta(p, a) = q$. Hence, δ is a function whose domain is a set of ordered pairs, (*p*, *a*), where *p* is a state and *a* is an input symbol, and the range is a set of states.

Therefore δ defines a map
ping from $Q \times \Sigma$ to Q,

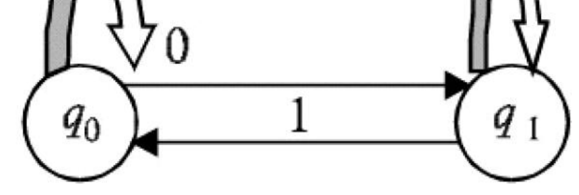
q_0 is the initial state, and *F* is a set of final sates of the automata. For example:

$$M = (\{q_0, q_1\}, \{0, 1\}, \delta, q_0, \{q_1\})$$

where

$$\delta(q_0, 0) = q_1,\ \delta(q_0, 1) = q_0$$
$$\delta(q_1, 0) = q_1,\ \delta(q_1, 1) = q_0$$

The transition diagram of this automata is:



Transition Diagram

	0	1
q_0	q_1	q_0
$^*q_1$	q_1	q_0

Transition Table

For example:

$$M = (\{q_0, q_1\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

where

$$\delta(q_0, 0) = q_1,\ \delta(q_0, 0) = q_0$$
$$\delta(q_1, {}^{0}0) = q_1,\ \delta(q_1, 1) = q_0$$

Let *x* be 010. To find out if x is accepted by the automata or not, we proceed as follows:

$\delta_1(q_0, 0) = \delta(q_0, 0) = q_1$

Therefore, $\delta_1(q_0, 01) = \delta\{\delta_1(q_0, 0), 1\} = q_0$

$\delta_1(q_0, 010) = \delta\{\delta_1(q_0, 01), 0\} = q_1$

In the finite automata discussed above, since δ defines mapping from Q × Σ to Q, there exists exactly one transition from a state on an input symbol; and therefore, this finite automata is considered a deterministic finite automata (DFA).

Therefore, we define the DFA as the finite automata:

where:

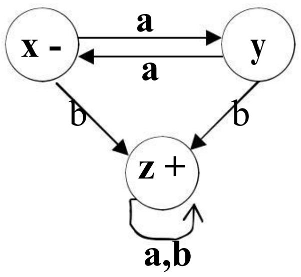
M = (*Q*, Σ, δ, *q* , *F*), such that there exists exactly one transition from a state on a input symbol.

Example if ∑={a,b}, states={x,y,z}

Rules of transition:

1. From state x and input a go to state y.
2. From state x and input b go to state z.
3. From state y and input a go to state x.
4. From state y and input b go to state z.
5. From state z and any input stay at the state z.

Let x be the start state and z be the final state.



Transition Diagram

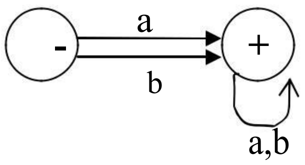
The FA above will accept all strings that have the letter b in them and no other strings. The language associated with(or accepted by) this FA is the one defined by the regular expression: *a*b(a+b)**

The set of all strings that do leave us in a final state is called the language defined by the FA. The word abb is accepted by this FA, but

The word aaa is not.

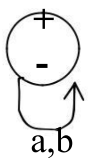
	a	b
x -	y	Z
y	x	Z
z +	z	Z

Example The following FA accept all strings from the alphabet {a,b} except Λ.



The regular expression is: (a+b)(a+b)*=$(a+b)^+$

Example The following FA accept all words from the alphabet {a,b}.

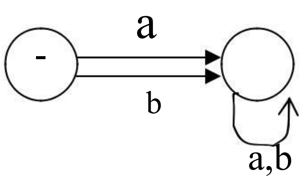


The regular expression is: (a+b)*

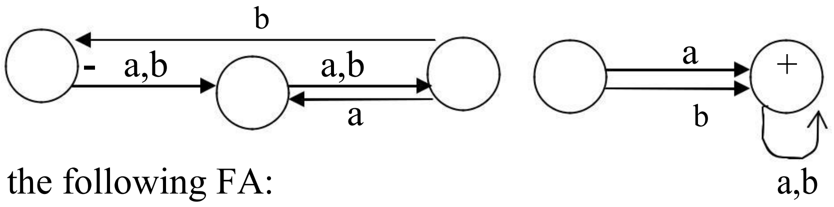
Note: every language that can be accepted by an FA can be defined by a regular expression and every language that can be defined by a regular expression can be accepted by some FA.

FA that accepts no language will be one of the two types:

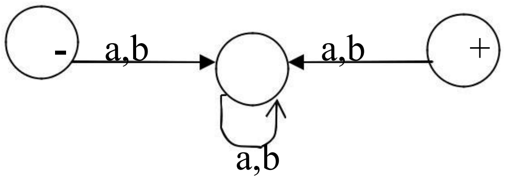
1. FA that have no final states. Like the following FA:



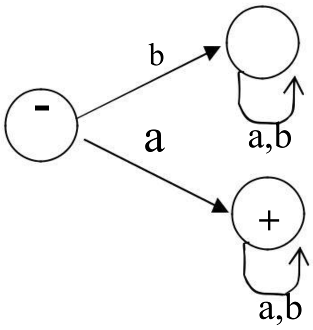
2. FA in which the final states cannot be reached. Like the following FA:



Or Like the following FA:

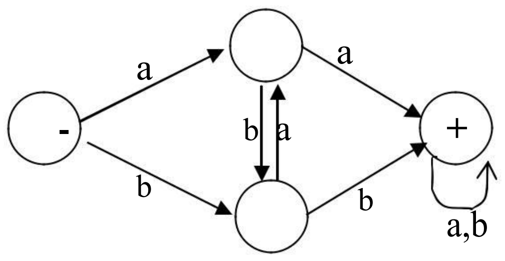


Example The following FA accept all strings from the alphabet {a,b} that start with a.



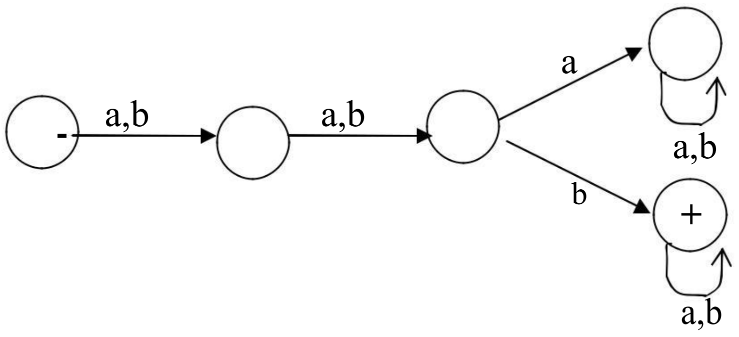
The regular expression is: a(a+b)*

<u>*Example*</u> The following FA accept all strings from the alphabet {a,b} with double letter.

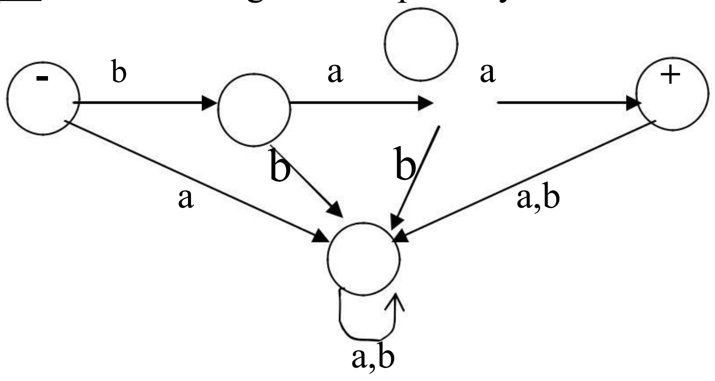


The regular expression is: (a+b)*(aa+bb) (a+b)*

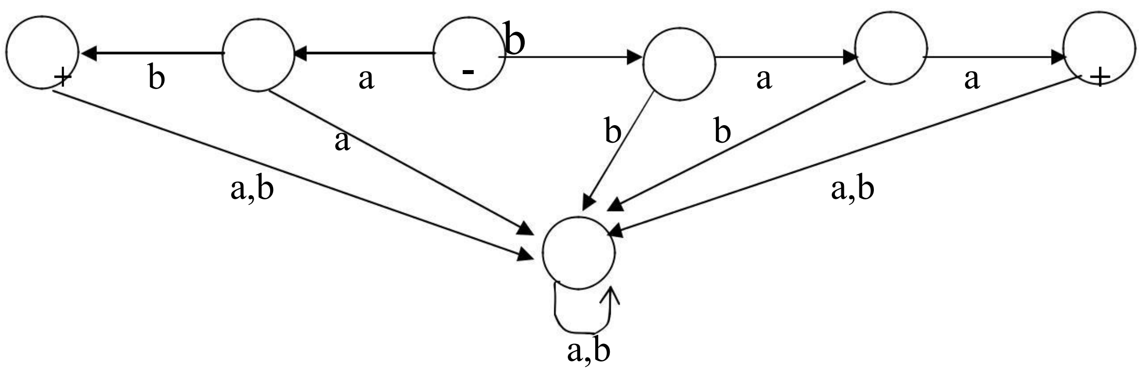
<u>*Example*</u> the following FA accepts the language defined by the regular expression: *(a+b)(a+b)b(a+b)**



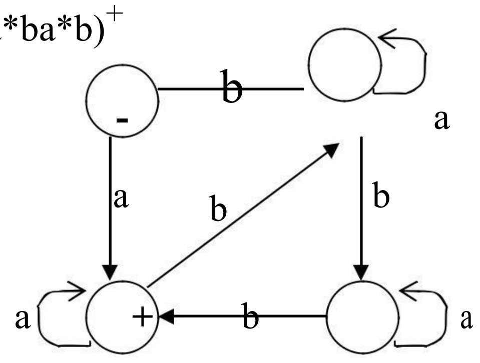
<u>*Example*</u> the following FA accepts only the word baa.



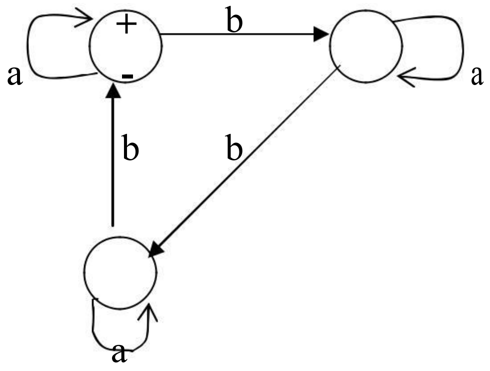
<u>*Example*</u> the following FA accepts the words baa and ab.



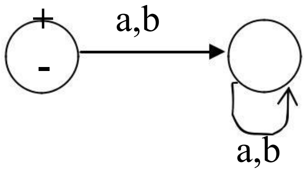
__Example__ the following FA accepts the language defined by the regular expression: $(a+ba^*ba^*b)^+$



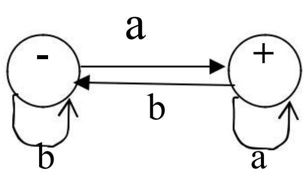
__Example__ the following FA accepts the language defined by the regular expression: $(a+ba^*ba^*b)^*$



__Example__ the following FA accepts only the word Λ.

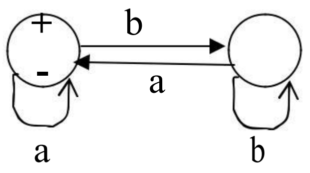


__Example__ the following FA accepts all words from the alphabet {a,b} that end with a.



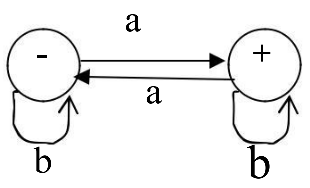
The regular expression for this language is: $(a+b)^*a$

Example the following FA accepts all words from the alphabet {a,b} that do not end in b and accept Λ.



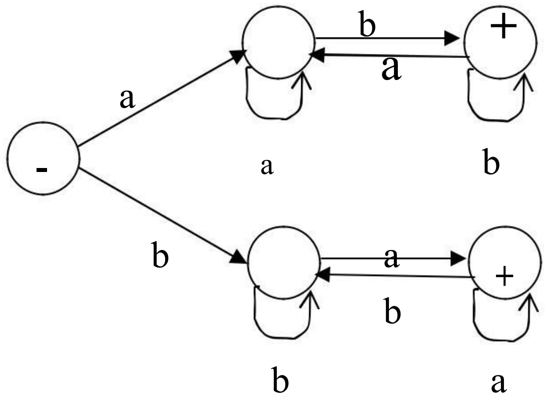
The regular expression for this language is: (a+b)*a + Λ

Example the following FA accepts all words from the alphabet {a,b} with an odd number of a's.



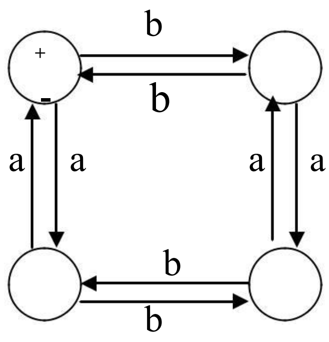
The regular expression for this language is: b*a(b*ab*ab*)*

Example the following FA accepts all words from the alphabet {a,b} that have different first and last letters.



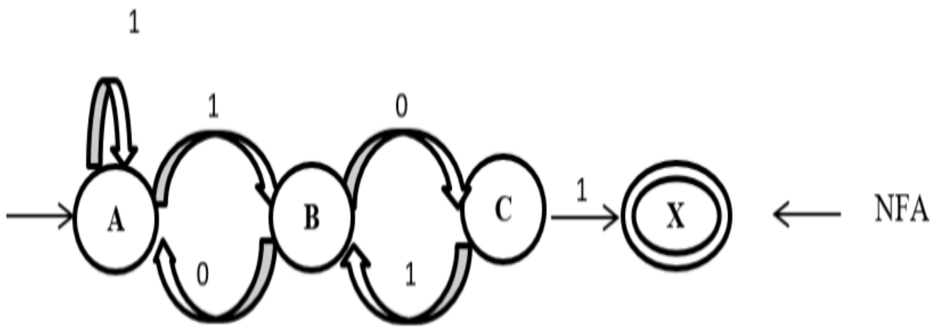
The regular expression for this language is: a(a+b)*b + b(a+b)*a

Example the following FA accepts the language defined by the regular expression (even-even): [aa+bb+(ab+ba)(aa+bb)*(ab+ba)]*



Converting From NFA to DFA

Ex (1):-convert the following NFA into an equivalents DFA



Solution:-

S ({A}, 0) = Φ

S ({A}, 1) = {A, B}   new node

S ({A, B}, 0) = {A, C}   new node

S ({A, B}, 1) = {A, B}

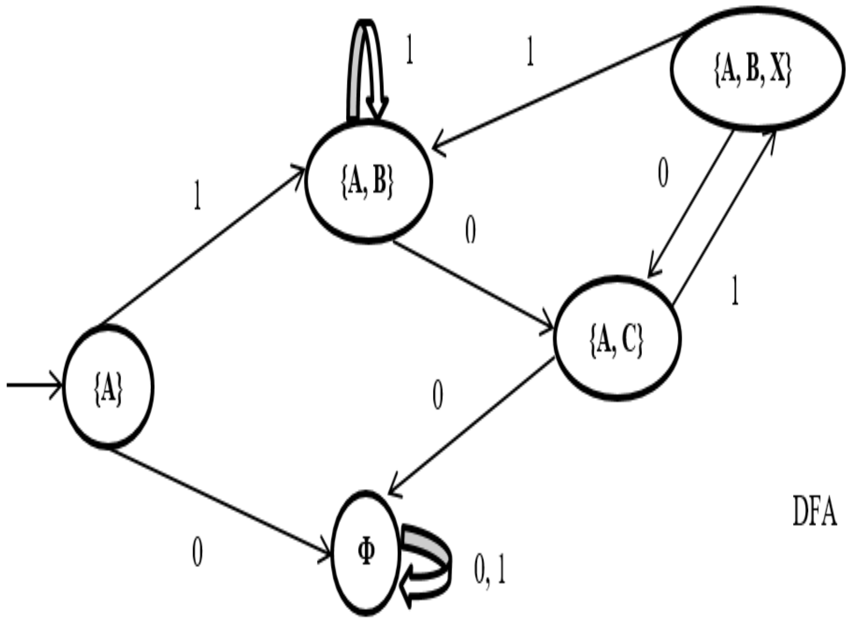
S ({A, C}, 0) = Φ

S ({A, C}, 1) = {A, B, X}   new node

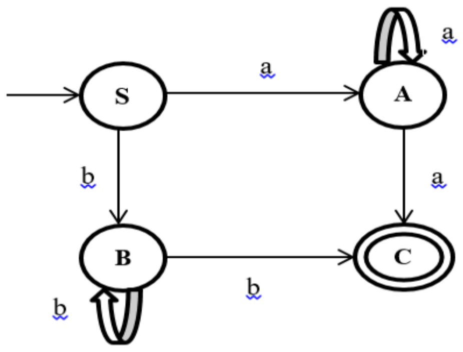
S ({A, B, X}, 0) = {A, C}

S ({A, B, X}, 1) = {A, B}

	0	**1**
{A}	**Φ**	**{A, B}**
{A, B}	**{A, C}**	**{A, B}**
{A, C}	**{Φ}**	**{A, B, X}**
{A, B, X}	**{A, C}**	**{A, B}**
{Φ}	**{Φ}**	**{Φ}**



Ex (2):-convert the following NFA into an equivalents DFA



Solution:-

S ({S}, a) = {A} new node

S ({S}, b) = {B} new node

S ({A}, a) = {A, C} new node

S ({A}, b) = Φ

S ({B}, a) = Φ

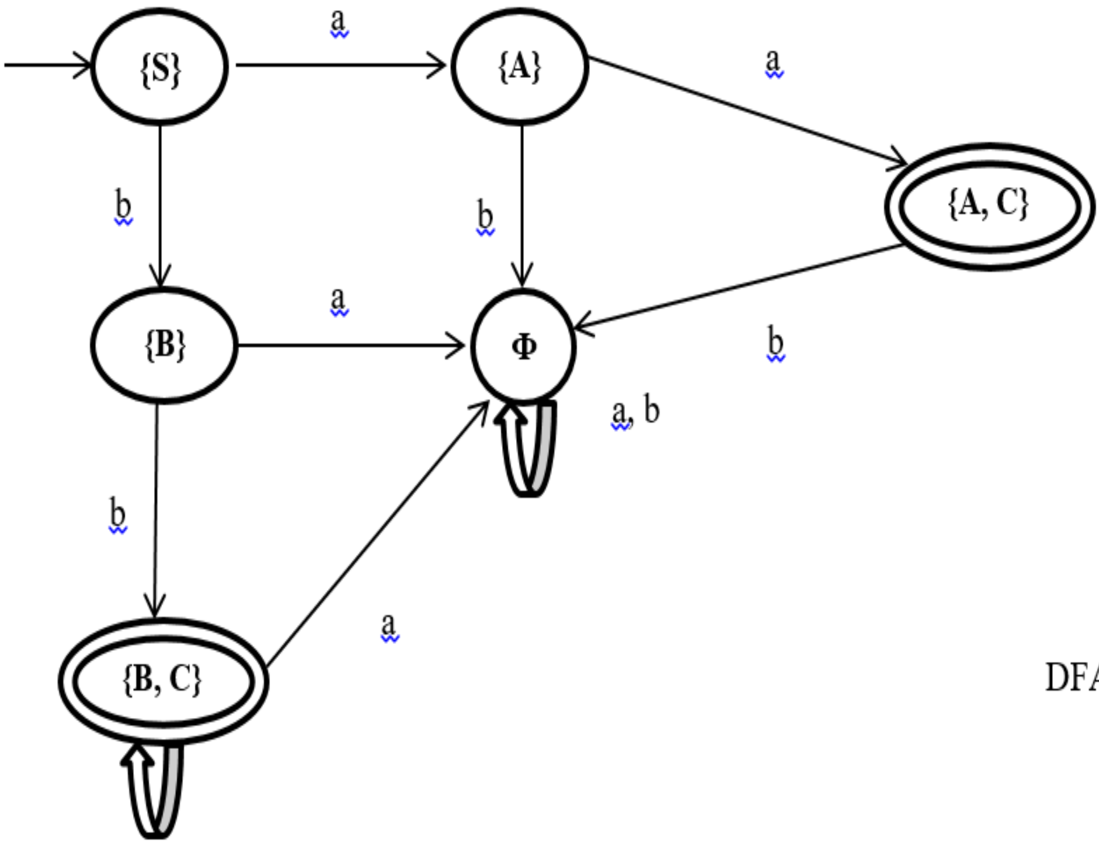
S ({B}, b) = {B, C} new node

S ({A, C}, a) = {A, C}

S ({A, C}, b) = Φ

S ({B, C}, a) = Φ

S ({B, C}, b) = {B, C}

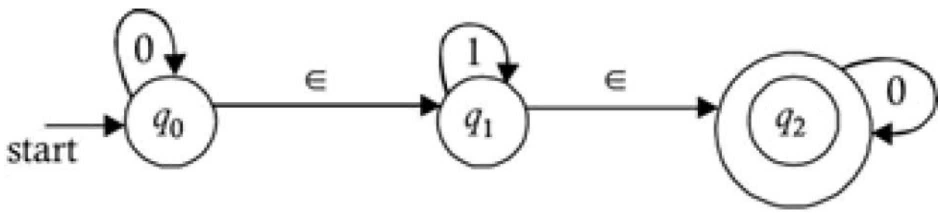


DFA

THE NFA WITH ∈-MOVES

If a finite automata is modified to permit transitions without input symbols, along with zero, one, or more transitions on the input symbols, then we get an NFA with '□-moves,' because the *transitions* made without symbols are called "□-transitions."

Consider the NFA shown in.



Finite automata with Î-moves.

This is an NFA with Î-moves because it is possible to transition from state *q*0 to *q*1 without consuming any of the input symbols. Similarly, we can also transition from state *q*1 to *q*2 without consuming any input symbols. Since it is a finite automata, an NFA with □-moves will also be denoted as a five-tuple:

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

where *Q*, □, *q*0, and F have the usual meanings, and □ defines a mapping from

$$Q \times (\Sigma \cup \in) \text{ to } 2^Q$$

(to take care of the □-transitions as well as the non □-transitions).

Since *x* is a member of Σ*, and there may exist zero, one, or more transitions from a state on an input symbol, we define a new transition function, δ , which defines a mapping from $2^Q \times \Sigma^*$ to 2^Q. If *x* is written as *wa*, where *a* is the last symbol of *x* and *w* is a string made of remaining symbols of *x* then:

$$\delta_1(\{q_0\}, x) = \delta_1(\delta_1(\{q_0\}, w), a)$$

since δ1 defines a mapping from $2^Q \times \Sigma^*$ to 2^Q.

$$\in\text{-closure}(\delta_1(\in\text{-closure}(q_0), x) = P$$

such that *P* contains at least one member of *F* and:

$$\in\text{-closure}(\delta_1(\in\text{-closure}(q_0), x) = \in\text{-closure}(\in\text{-closure}(\delta_1(\in\text{-closure}(q_0), w)), a)$$

For example, in the NFA with ∈-moves, given above, if *x* = 01, then to find out whether *x* is accepted by the automata or not, we proceed as follows:

$$\begin{aligned}
\delta_1(\in\text{-closure}(q_0), 0) &= \delta_1(\{q_0, q_1, q_2\}, 0) \\
&= \delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0) \\
&= \{q_0\} \cup \phi \cup \{q_2\} = \{q_0, q_2\} \\
\delta_1(\in\text{-closure}(q_0), 01) &= \delta_1(\in\text{-closure}(\delta_1(\in\text{-closure}(q_0), 0)), 1) \\
&= \delta_1(\in\text{-closure}(\{q_0, q_2\}), 1) \\
&= \delta_1(\{q_0, q_1, q_2\}, 1) \\
&= \delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1) \\
&= \phi \cup \{q_1\} \cup \phi \\
&= \{q_1\}
\end{aligned}$$

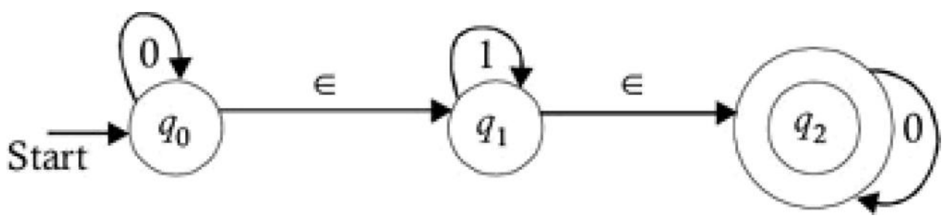
Therefore:

∈-closure(δ1 (∈-closure (q0), 01) = ∈-closure({q1}) = {q1, q2} Since *q2* is a final state, *x* = 01 is accepted by the automata.

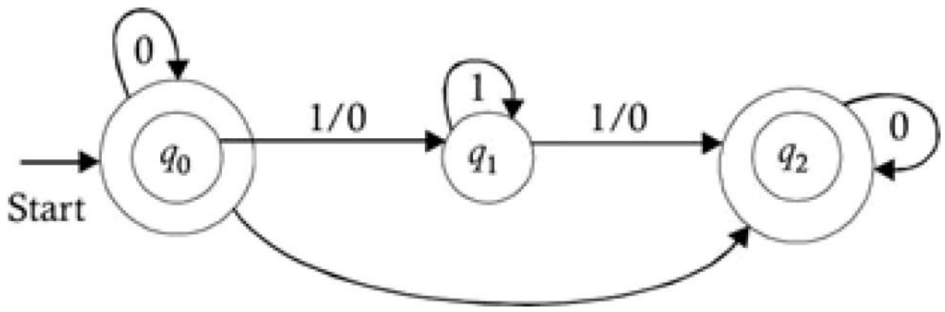
Equivalence of NFA with ∈-Moves to NFA Without ∈-Moves

For every NFA with ∈-moves, there exists an equivalent NFA without ∈-moves that accepts the same language. To obtain an equivalent NFA without1 ∈-moves, given an NFA with ∈-moves, what is required is an elimination of ∈-transitions from a given automata. But simply eliminating the ∈-transitions from a given NFA with ∈-moves will change the language accepted by the automata. Hence, for every ∈-transition to be eliminated, we have to add some non-∈-transitions as substitutes in order to maintain the language's acceptance by the automata. Therefore, transforming an NFA with ∈-moves to and NFA without ∈-moves involves finding the non-∈-transitions that must be added to the automata for every ∈-transition to be eliminated.

Consider the NFA with □-moves shown in



Therefore, by adding these non-□-transitions, and by making the initial state one of the final states, we get the automata shown in.



Therefore, when transforming an NFA with □-moves into an NFA without ∈-moves, only the transitions are required to be changed; the states are not required to be changed. But if a given NFA with *q*0 and ∈-moves accepts □ (i.e., if the ∈-closure (*q*0) contains a member of F), then *q*0 is also required to be marked as one of the final states if it is not already a member of F. Hence:

If *M* = (*Q*, □, □, *q*0, *F*) is an NFA with □-moves, then its equivalent NFA without □-moves will be *M*1 = (*Q*, □, □1, *q*0, *F*1)

where □1 (*q*, *a*) = ∈-closure(□ (∈-closure(*q*), *a*))

and

*F*1 = *F* È (*q*0) if Î-closure (*q*0) contains a member of *F*

*F*1 = *F* otherwise

For example, consider the following NFA with Î-moves:

M=({q0,q1,q2},{0,1}, δ, q0, {q2})

Where

δ	0	1	λ
q0	{q0}	Φ	{q1}
q1	Φ	{q1}	{q2}
q2	Φ	{q2}	Φ

Its equivalent NFA without λ- moves will be:

M=({q0,q1,q2},{0,1}, δ, q0, {q0,q2})

δ 1	0	1
q0	{q0,q1,q2}	{q1,q2}
q1	Φ	{q1,q2}
q2	Φ	{q2}

FINITE AUTOMATA WITH OUTPUT
Moore machine and *Mealy machine*.

We shall investigate two different models for FA's with output capabilities; these are *Moore machine* and *Mealy machine*.

A **Moore machine** is a collection of five things:

1. A finite set of states q0,q1,q2,… where q0 is designed as the start state.
2. An alphabet of letters for forming the input string ∑= { a, b, c, …}.
3. An alphabet of possible output characters Γ = { x, y, z, …}.
4. A transition table that shows for each state and each input letter what state is reached next.
5. An output table that shows what character from Γ is printed by each state that is entered.

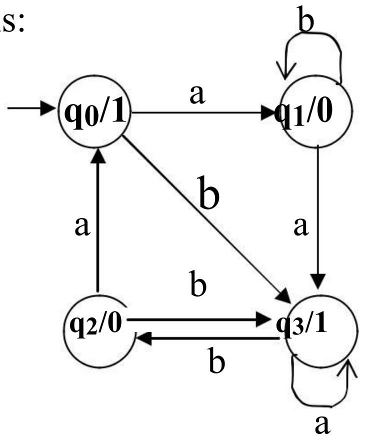
A Moore machine does not define a language of accepted words, since every input string creates an output string and there is no such thing as a final state. The processing is terminated when the last input letter is read and the last output character is printed.

Example

Input alphabet: ∑ = {a, b}
Output alphabet: Γ = {0, 1}
Names of states: q0, q1, q2, q3. (q0 = start state)

Old state	**Transition table** New state After input a	after input b	**Output table** (the character printed in the old state)
-q0	q1	q3	1
q1	q3	q1	0
q2	q0	q3	0
q3	q3	q2	1

The Moore machine is:

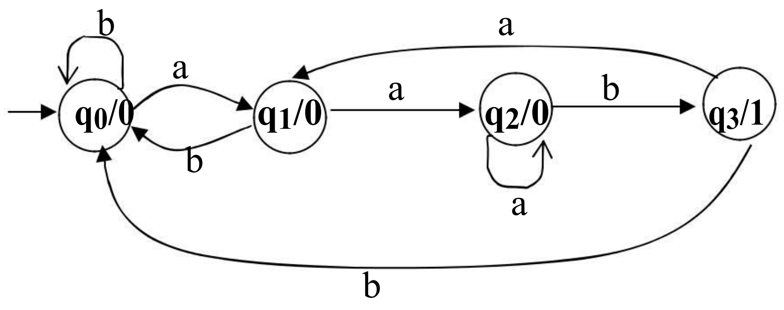


Note: the two symbols inside the circle are separated by a slash "/", on the left side is the name of the state and on the right is the output from that state.

If the input string is abab to the Moore machine then the output will be 10010.

Example

The following Moore machine will "count" how many times the substring aab occurs in a long input string.



The number of substrings aab in the input string will be exactly the number of 1's in the output string.

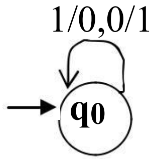
Input string		a	a	a	b	a	b	b	a	a	b	b
State	q_0	q_1	q_2	q_2	q_3	q_1	q_0	q_0	q_1	q_2	q_3	q_0
Output	0	0	0	0	1	0	0	0	0	0	1	0

A **Mealy machine** is a collection of four things:

1. A finite set of states q0,q1,q2,… where q0 is designed as the start state.
2. An alphabet of letters for forming the input string ∑= { a, b, c, …}.
3. An alphabet of possible output characters Γ = { x, y, z, …}.
4. A pictorial representation with states represented by small circles and directed edges indicating transitions between states. Each edge is labeled with a compound symbol of the form i/o where i is an input letter and o is an output character. Every state must have exactly one outgoing edge for each possible input letter. The edge we travel is determined by the input letter i; while traveling on the edge we must print the output character o.

<u>*Example*</u>

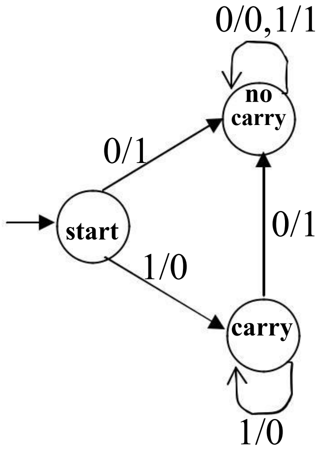
The following Mealy machine prints out the 1's complement of an input bit string.



If the input is 001010 the output is 110101. This is a case where the input alphabet and output alphabet are both {0,1}.

<u>*Example*</u>

The following Mealy machine called the increment machine.



If the input is 1011 the output is 1100.

<u>Definition</u>

Given the Mealy machine Me and the Moore machine Mo, which prints the automatic start -state character x, we will say that these two machines are equivalent if for every input string the output string from Mo is exactly x concatenated with the output from Me.

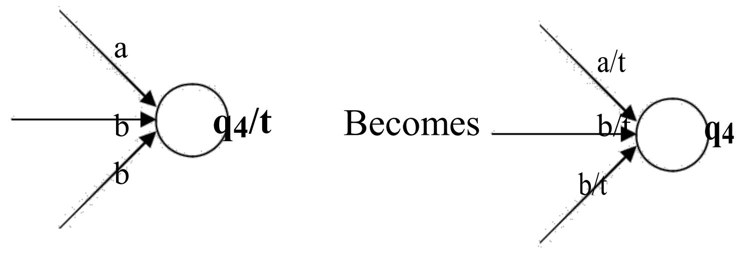
Note: we prove that for every Moore machine there is an equivalent Mealy machine and for every Mealy machine there is an equivalent Moore machine. We can then say that the two types of machine are completely equivalent.

Theorem

If Mo is a Moore machine, then there is a Mealy machine Me that is equivalent to it.

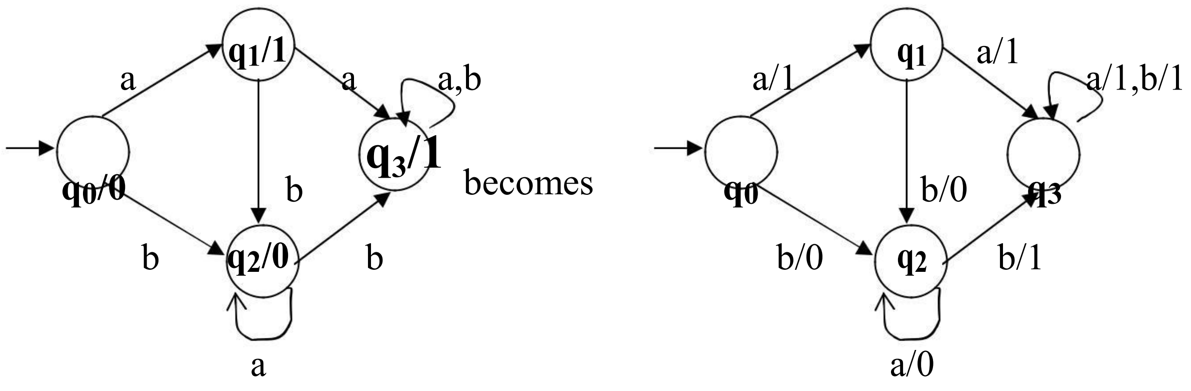
Proof

The proof will be by constructive algorithm.



Example

Below, a Moore machine is converted into a Mealy machine:

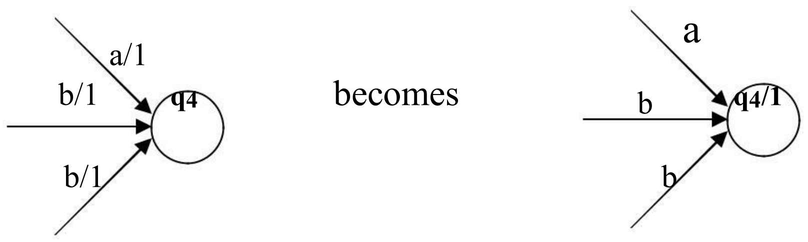


Theorem

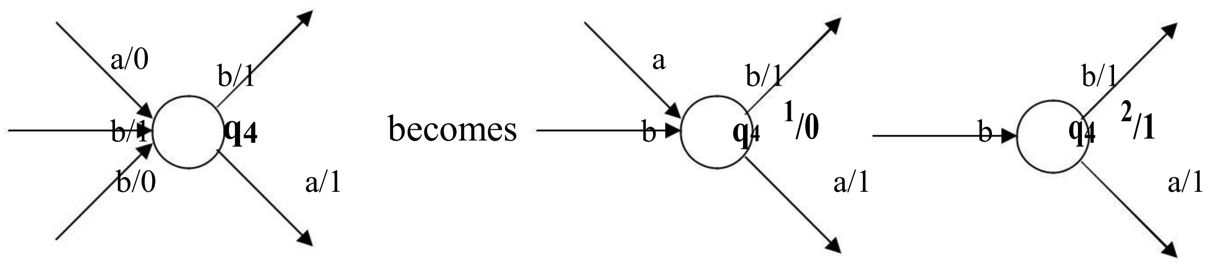
For every Mealy machine Me there is a Moore machine Mo that is equivalent to it.

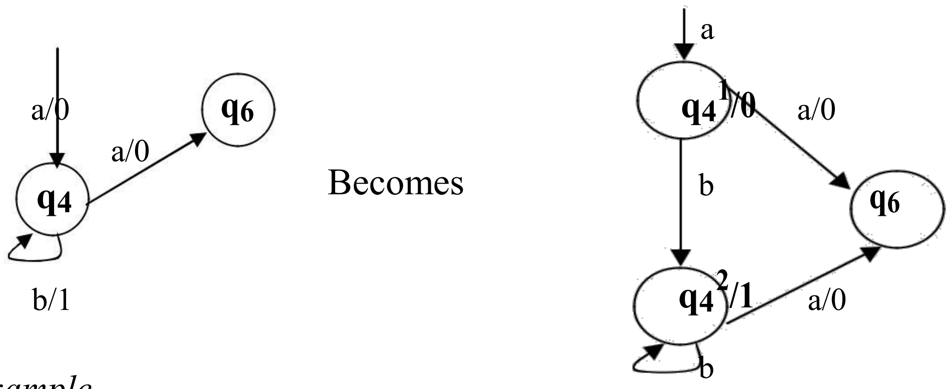
Proof

The proof will be by constructive algorithm.



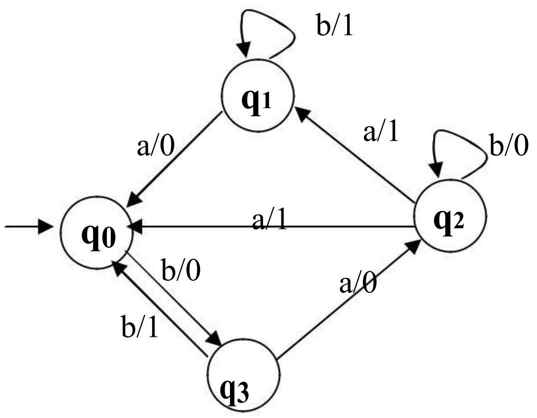
If there is more than one possibility for printing as we enter the state, then we need a copy of the state for each character we might have to print. (we may need as many copies as there are character in Γ).



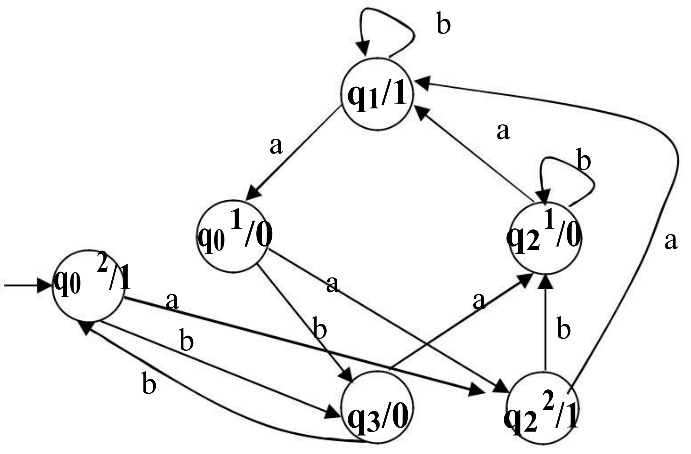


Example

Convert the following Mealy machine to Moore machine:



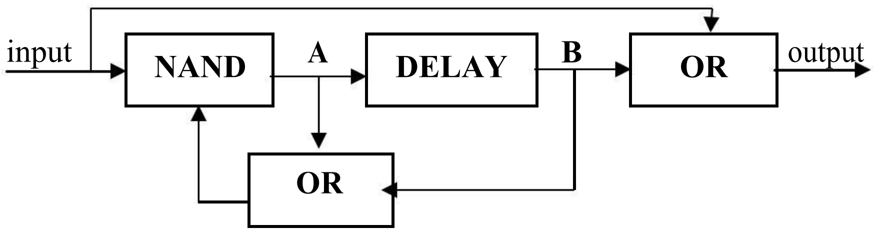
Mealy machine



Moore machine

Example

Draw the Mealy machine for the following sequential circuit:



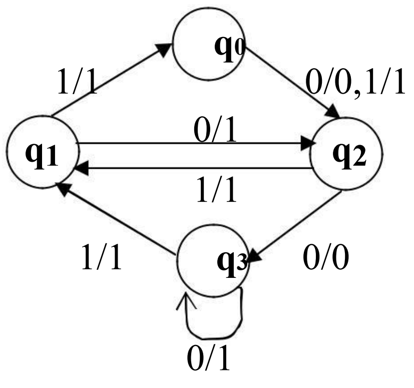
First we identify four states:

q0 is A= 0 B= 0
q1 is A= 0 B= 1
q2 is A= 1 B= 0
q3 is A= 1 B= 1

the operation of this circuit is such that after an input of 0 or 1 the state changes according to the following rules: new B= old A

new A = (input) NAND (old A OR old B)
output = (input) OR (old B)
Suppose we are in q0 and we receive the input 0.
new B = old A = 0
new A = 0 NAND (0 OR 0)
= 0 NAND 0
= 1
output = 0 OR 0 = 0
the new state is q2 (since new A=1, new B=0)
if we are in state q0 and we receive the input 1:
new B= old A = 0
new A = 1 NAND (0 OR 0) =1
output = 1 OR 0 =1
the new state is q2.
We repeat this process for every state and for each input to produce the following table:

Old state	After input 0		After input 1	
	New state	Output	New state	Output
q0	q2	0	q2	1
q1	q2	1	q0	1
q2	q3	0	q1	1
q3	q3	1	q1	1



Mealy machine

A phrase-Structure Grammar

A *phrase-Structure Grammar*, called **PSG**, is a collection of three things:

1. An alphabet ∑ of letters called **terminals.**
2. A set of symbols called **nonterminals** that includes the start symbol S.
3. A finite set of productions of the form:

$$\text{String } 1 \rightarrow \text{String } 2$$

Where string 1 can be any string of terminals and nonterminals that contains at least one nonterminals and where string 2 is any string of terminals and nonterminals whatsoever.

Definition:
The language generated by the PSG is the set of all strings of terminals that can be derived starting at S.

Example: the following is a phrase-structure grammar over Σ={a,b} with nonterminals X and S:

$S \longrightarrow XS$
$X \longrightarrow aX \mid a$
$aaaX \longrightarrow ba$

In this language we can have the following derivation:

```
S ⟶ XS XXS
    XXXS
    XXX
    aXXX
    aaXXX
    aaaXXX
    baXX
    baaXX
    baaaX
    bba
```

Example:

$S \longrightarrow aSBA$
$S \longrightarrow abA$
$AB \longrightarrow BA$
$bB \longrightarrow bb$
$bA \longrightarrow ba$
$aA \longrightarrow aa$

A context-sensitive grammar (CSG)

A context-sensitive grammar (CSG) is a formal grammar in which the left-hand sides and right-hand sides of any production rules may be surrounded by a context of terminal and nonterminal symbols.

Definition

A formal grammar $G = (N, \Sigma, P, S)$
Where N the Non - Terminal
Σ the terminal
P is context-sensitive if all rules in P are of the form $\alpha A\beta \rightarrow \alpha\gamma\beta$
where $A \in N$ (i.e., A is a single nonterminal),
$\alpha,\beta \in (N \cup \Sigma)^*$ (α and β are strings of nonterminals and terminals)
and $\gamma \in (N \cup \Sigma)^+$ (γ is a nonempty string of nonterminals and terminals).
A rule of the form $S \rightarrow \lambda$ provided S does not appear on the right side of any rule where λ represents the empty string is permitted.
The addition of the empty string allows the statement that the context sensitive languages are a proper superset of the context free languages, rather than having to make the weaker statement that all context free grammars with no →λ productions are also context sensitive grammars.

<u>Example:</u> This grammar generates the context sensitive language:

1. $S \rightarrow aSBC$
2. $S \rightarrow aBC$
3. $CB \rightarrow HB$
4. $HB \rightarrow HC$
5. $HC \rightarrow BC$
6. $aB \rightarrow ab$
7. $bB \rightarrow bb$
8. $bC \rightarrow bc$
9. $cC \rightarrow cc$

$\Rightarrow_1 aSBC$
$\Rightarrow_1 a\boldsymbol{aSBC}BC$
$\Rightarrow_2 aa\boldsymbol{aBC}BCBC$
$\Rightarrow_3 aaaB\boldsymbol{HB}CBC$
$\Rightarrow_4 aaaB\boldsymbol{HC}CBC$
$\Rightarrow_5 aaaB\boldsymbol{BC}CBC$
$\Rightarrow_3 aaaBBC\boldsymbol{HB}C$
$\Rightarrow_4 aaaBBC\boldsymbol{HC}C$
$\Rightarrow_5 aaaBBC\boldsymbol{BC}C$
$\Rightarrow_3 aaaBB\boldsymbol{HB}CC$
$\Rightarrow_4 aaaBB\boldsymbol{HC}CC$
$\Rightarrow_5 aaaBB\boldsymbol{BC}CC$
$\Rightarrow_6 aa\boldsymbol{ab}BBCCC$
$\Rightarrow_7 aaa\boldsymbol{bb}BCCC$
$\Rightarrow_7 aaab\boldsymbol{bb}CCC$
$\Rightarrow_8 aaabb\boldsymbol{bc}CC$
$\Rightarrow_9 aaabbb\boldsymbol{cc}C$
$\Rightarrow_9 aaabbbc\boldsymbol{cc}$

CONTEXT FREE GRAMMAR

A *context free grammar*, called **CFG**, is a collection of <u>three</u> things:

1. An alphabet $\sum$ of letters called **terminals** from which we are going to make strings that will be the words of a language.
2. A set of symbols called **nonterminals**, one of which is the symbol S, standing for "start here".
3. A finite set of production of the form:

One nonterminal $\rightarrow$ finite string of terminals and/ or nonterminals Where the strings of terminals and nonterminals can consist of only terminals or of any nonterminals, or any mixture of terminals and nonterminals or even the empty string. We require that at least one production has the nonterminal S as its left side.

<u>Definition</u>

The language generated by the CFG is the set of all strings of terminals that can be produced from the start symbol S using the production as substitutions. A language generated by the CFG is called a **context free language (CFL)**.

<u>Example</u>

Let the only terminal be a.

Let the only nonterminal be S.

Let the production be:

$$S \rightarrow aS$$
$$S \rightarrow \Lambda$$

The language generated by this CFG is exactly a*.

In this language we can have the following derivation:

$S \rightarrow aS \rightarrow aaS \rightarrow aaaS \rightarrow aaaaS \rightarrow aaaaaS \rightarrow aaaaa\Lambda = aaaaa$

Example

Let the only terminal be a.

Let the only nonterminal be S.

Let the production be:

$$S \rightarrow SS$$
$$S \rightarrow a$$
$$S \rightarrow \Lambda$$

The language generated by this CFG is also just the language a*.

In this language we can have the following derivation:

$S \rightarrow SS \rightarrow SSS \rightarrow SaS \rightarrow SaSS \rightarrow \Lambda aSS \rightarrow \Lambda aaS \rightarrow \Lambda aa\Lambda = aa$

Example

Let the terminals be a, b. And the only nonterminal be S.

Let the production be:

$$S \rightarrow aS$$
$$S \rightarrow bS$$
$$S \rightarrow a$$
$$S \rightarrow b$$

The language generated by this CFG is $(a+b)^+$.

In this language we can have the following derivation:

$S \rightarrow bS \rightarrow baS \rightarrow baaS \rightarrow baab$

Example

Let the terminals be a, b. And the only nonterminal be S.

Let the production be:

$$S \rightarrow aS$$
$$S \rightarrow bS$$
$$S \rightarrow \Lambda$$

The language generated by this CFG is $(a+b)^*$.

In this language we can have the following derivation:

$S \rightarrow bS \rightarrow baS \rightarrow baaS \rightarrow baa\Lambda = baa$

Example

Let the terminals be a, b,Λ. And the nonterminals be S,X,Y.

Let the production be:

$$\begin{aligned} S &\to X \\ S &\to Y \\ X &\to \Lambda \\ Y &\to aY \\ Y &\to bY \\ Y &\to a \\ Y &\to b \end{aligned}$$

The language generated by this CFG is $(a+b)^*$.

Example

Let the terminals be a, b,Λ. And the nonterminals be S,X,Y.

Let the production be:

$$\begin{aligned} S &\to XY \\ X &\to \Lambda \\ Y &\to aY \\ Y &\to bY \\ Y &\to a \\ Y &\to b \end{aligned}$$

The language generated by this CFG is $(a+b)^+$.

Example

Let the terminals be a, b.

Let the nonterminals be S,X.

Let the production be:

$$\begin{aligned} S &\to XaaX \\ X &\to aX \\ X &\to bX \\ X &\to \Lambda \end{aligned}$$

The language generated by this CFG is (a+b)* aa(a+b)*.

To generate baabaab we can proceed as follows:

$S \to XaaX \to bXaaX \to baXaaX \to baaXaaX \to baabXaaX \to baab\Lambda aaX = baabaaX$

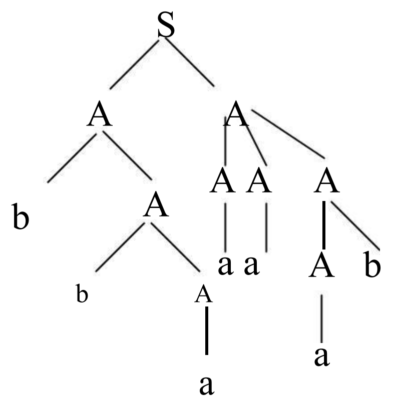
$\to baabaabX \to baabaab\Lambda = baabaa$

Trees

Example

S → AA
A → AAA | bA | Ab | a

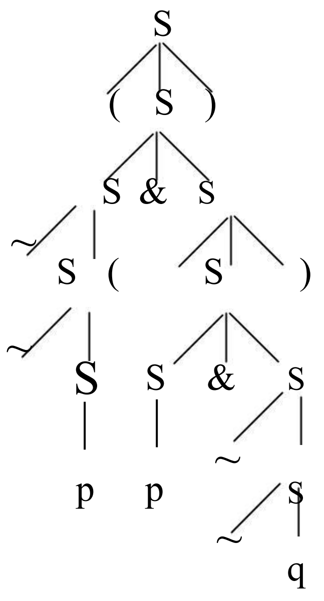
If we want to produce the word bbaaaab, the tree will be:



This diagram is called **syntax tree** or **parse tree** or **generation tree** or **production tree** or **derivation tree**.

Example

S → (S) | S&S | ~S | p| q
The derivation tree for the word (~ ~ p & (p & ~ ~ q)) will be:

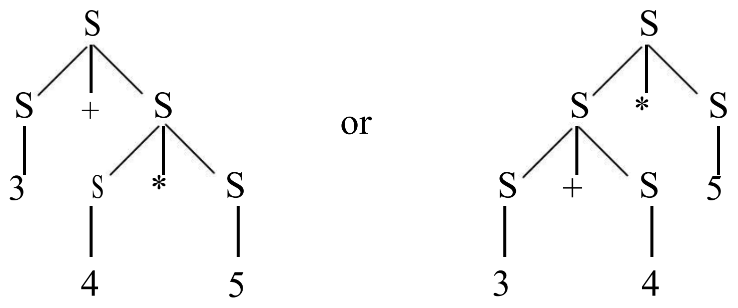


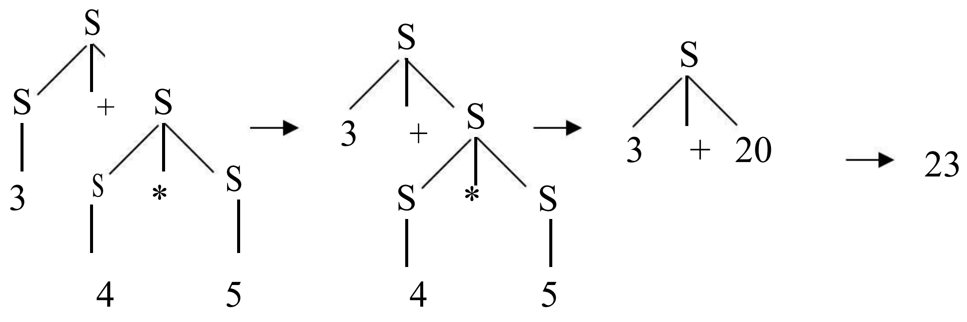
Example

S → S + S | S* S | number

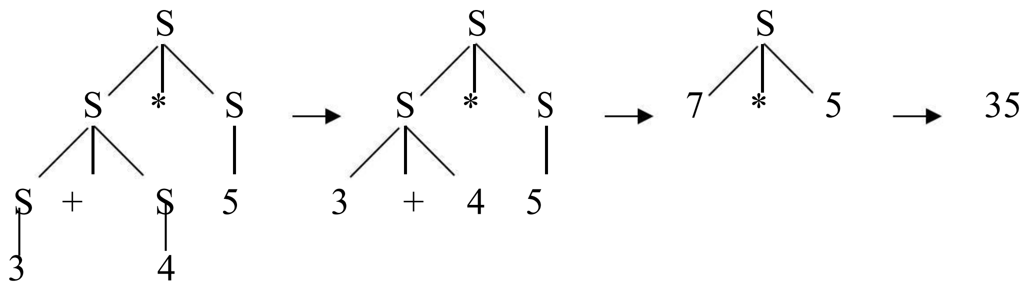
Does the expression 3+4*5 mean (3+4)*5 which is 35 or does it mean 3+(4*5) which is 23?

We can distinguish between these two possible meaning for the expression 3+4*5 by looking at the two possible derivation trees that might have produced it.





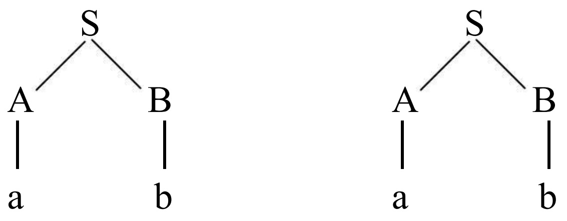
Or



Example

S → AB
A → a
B → b

S → AB → aB → ab or S → AB → Ab → ab

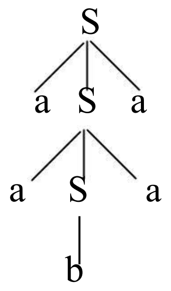


There is no ambiguity of interpretation.

Definition
A CFG is called **ambiguous** if for at least one word in the language that it generates there are two possible derivations of the word that correspond to different syntax trees.

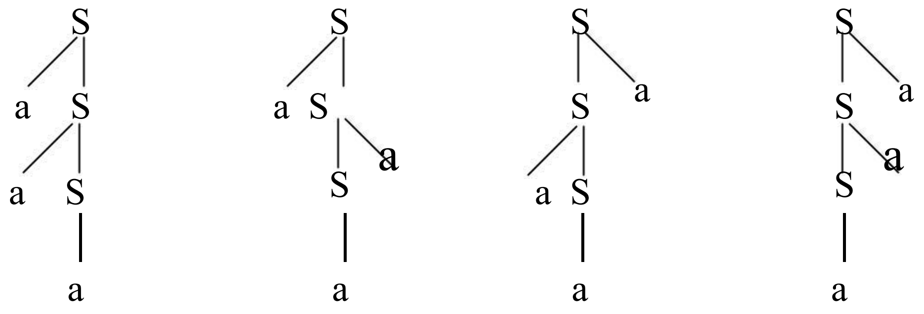
Example
The CFG for palindrome
S → aSa | bSb |a |b | Λ

S → aSa → aaSaa → aabaa

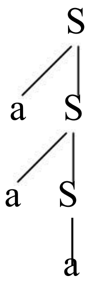


The CFG is unambiguous.

Example
S → aS |Sa | a
In this case the word a^3 can be generated by four different trees:



The CFG is therefore ambiguous.
The same language can be defined by the CFG:
S → aS | a
For which the word a^3 has only one production tree:



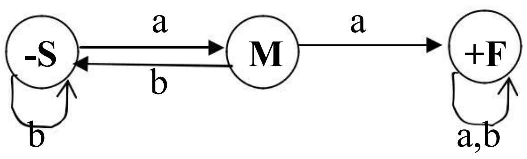
This CFG is not ambiguous.

REGULAR GRAMMARS

Note : all regular languages can be generated by CFG's, and so can some non-regular languages but not all possible languages.

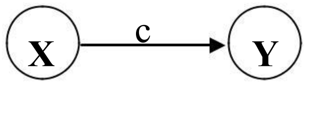
Example

Consider the FA below, which accepts the language of all words with a double a:



All the necessary to convert it to CFG is that:

1. every edge between states be a production:

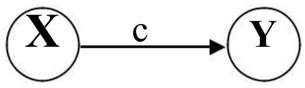


becomes X→ cY

and

2. every production correspond to an edge between states:

X→ cY comes from



or to the possible termination at a final state:

X→ Λ

only when X is a final state.

So the production rules of our example will be:

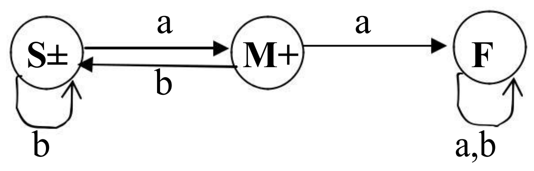
S → aM | bS
M → aF | bS
F →aF | bF | Λ

Definition

For a given CFG a **semiword** is a string of terminals (maybe none) concatenated with exactly one nonterminal (on the right), for example:

(terminal) (terminal) . . . (terminal) (Nonterminal)

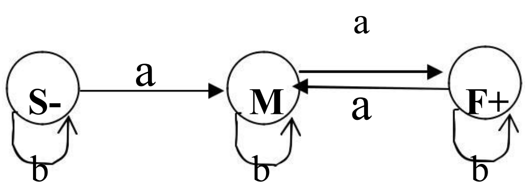
Example
Consider the following FA with two final states:



So the production rules of our example will be:
S → aM | bS | Λ
M → aF | bS | Λ
F →aF | bF

Theorem
All regular languages can be generated by CFG's. This can be stated as:
All regular languages are CFL's.

Example
The language of all words with an even number of a's (with at least some a's) is regular since it can be accepted by this FA:



We have the following set of productions:
S → aM | bS
M → aF | bM
F →aM | bF | Λ

linear grammar

A grammar is linear if it is context-free and all of its productions' right hand sides have at most one nonterminal.
A linear language is a language generated by some linear grammar.

Example
A simple linear grammar is *G* with *N* = {S}, Σ = {a, b}, *P* with start symbol *S* and rules
S → aSb
S → Λ
It generates the language $\{a^i b^i \mid i \geq 0\}$

Relationship with regular grammars:

Two special types of linear grammars are the following:

1. the **left-linear or left regular grammars**, in which all nonterminals in right hand sides are at the left ends;
2. the **right-linear or right regular grammars**, in which all nonterminals in right hand sides are at the right ends.

These two special types of linear grammars are known as the regular grammars; both can describe exactly the regular languages.
Another special type of linear grammar is the linear grammars in which all nonterminals in right hand sides are at the left or right ends, but not necessarily all at the same end.

By inserting new nonterminals, every linear grammar can be brought into this form without affecting the language generated. For instance, the rules of *G* above can be replaced with
S → aA
A → Sb
S → Λ

CHOMSKY NORMAL FORM (CNF)

Theorem

If L is a context-free language generated by CFG that includes Λ-productions, then there is a different CFG that has no Λ-production that generates either the whole language L(if L does not include the word Λ) or else generates the language of all the words in L that are not Λ.

Definition

In a given CFG, we call a nonterminal N **nullable** if:

- There is a production: N→ Λ

Or

- There is a derivation that start at N and leads to Λ: N→…→ Λ

Example

Consider the CFG:

S → a| Xb| aYa

X → Y | Λ

Y → b | X

X and Y are nullable.

The new CFG is:

S → a| Xb| aYa| b| aa

X → Y

Y → b | X

Example

Consider the CFG:

S → Xa

X → aX| bX | Λ

X is the only nullable nonterminal.

The new CFG is:

$S \rightarrow Xa|\ a$

$X \rightarrow aX|\ bX\ |a|\ b$

Example

Consider the CFG:

$S \rightarrow XY$

$X \rightarrow Zb$

$Y \rightarrow bW$

$Z \rightarrow AB$

$W \rightarrow Z$

$A \rightarrow aA|\ bA|\ \Lambda$

$B \rightarrow Ba|\ Bb|\ \Lambda$

A, B, W and Z are nullable.

The new CFG is:

$S \rightarrow XY$

$X \rightarrow Zb|\ b$

$Y \rightarrow bW|\ b$

$Z \rightarrow AB|\ A|\ B$

$W \rightarrow Z$

$A \rightarrow aA|\ bA|\ a|\ b$

$B \rightarrow Ba|\ Bb|\ a|\ b$

Definition

A production of the form: One Nonterminal → One Nonterminal is called a **unit** production.

Theorem

If there is a CFG for the language L that has no Λ-production, then there is also a CFG for L with no Λ-production and no unit production.

Example

Consider the CFG:

$$S \rightarrow A \mid bb$$
$$A \rightarrow B \mid b$$
$$B \rightarrow S \mid a$$

$S \rightarrow A$	gives	$S \rightarrow b$
$S \rightarrow A \rightarrow B$	gives	$S \rightarrow a$
$A \rightarrow B$	gives	$A \rightarrow a$
$A \rightarrow B \rightarrow S$	gives	$A \rightarrow bb$
$B \rightarrow S$	gives	$B \rightarrow bb$
$B \rightarrow S \rightarrow A$	gives	$B \rightarrow b$

The new CFG for this language is:

$$S \rightarrow bb \mid b \mid a$$
$$A \rightarrow b \mid a \mid bb$$
$$B \rightarrow a \mid bb \mid b$$

Theorem

If L is a language generated by some CFG then there is another CFG that generates all the non- Λ words of L, all of these productions are of one of two basic forms:

Nonterminal → string of only Nonterminals

Or

Nonterminal → One Terminal

Example

Consider the CFG:

$S \rightarrow X_1 | X_2aX_2 | aSb |$

$b\ X_1 \rightarrow X_2X_2 | b$

$X_2 \rightarrow aX_2 | aaX_1$

Becomes:

$S \rightarrow X_1$

$S \rightarrow X_2AX_2$

$S \rightarrow ASB\ S$

$\rightarrow B$

$X_1 \rightarrow X_2X_2$

$X_1 \rightarrow B$

$X_2 \rightarrow AX_2$

$X_2 \rightarrow AAX_1$

$A \rightarrow a$

$B \rightarrow b$

Example

Consider the CFG:

$S \rightarrow Na$

$N \rightarrow a | b$

Becomes:

$S \rightarrow NA$
$N \rightarrow a | b$
$A \rightarrow a$

Theorem

For any CFL the non- Λ words of L can be generated by a grammar in which all productions are of one of two forms:

Nonterminal → string of exactly two Nonterminals

Or

Nonterminal → One Terminal

Definition

If a CFG has only productions of the form:

Nonterminal → string of two Nonterminals

Or of the form:

Nonterminal → One Terminal

It is said to be in **Chomsky Normal Form (CNF)**.

Example

Convert the following CFG into CNF: $S \rightarrow aSa \mid bSb \mid a \mid b \mid aa \mid bb$

$S \rightarrow ASA$

$S \rightarrow BSB$

$S \rightarrow AA$

$S \rightarrow BB$

$S \rightarrow a$

$S \rightarrow b$

$A \rightarrow a$

$B \rightarrow b$

The CNF:

$S \rightarrow AR_1$

$R_1 \rightarrow SA$

$S \rightarrow BR_2$

$R_2 \rightarrow SB$

$S \rightarrow AA$

$S \rightarrow BB$

$S \rightarrow a$

$S \rightarrow b$

$A \rightarrow a$

$B \rightarrow b$

Example

Convert the following CFG into CNF:

$S \rightarrow bA \mid aB$

$A \rightarrow bAA \mid aS \mid a$

$B \rightarrow aBB \mid bS \mid b$

The CNF:

$S \rightarrow YA \mid XB$

$A \rightarrow YR_1 \mid XS \mid a$

$B \rightarrow XR_2 \mid YS \mid b$

$X \rightarrow a$

$Y \rightarrow b$

$R_1 \rightarrow AA$

$R_2 \rightarrow BB$

Example

Convert the following CFG into CNF:

$S \rightarrow AAAAS$

$S \rightarrow AAAA$

$A \rightarrow a$

The CNF:

$S \rightarrow AR_1$

$R_1 \rightarrow AR2$

$R_2 \rightarrow AR_3$

$R_3 \rightarrow AS$

$S \rightarrow AR_4$

$R_4 \rightarrow AR_5$

$R_5 \rightarrow AA$

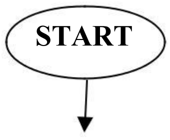
$A \rightarrow a$

PUSHDOWN AUTOMATA (PDA)

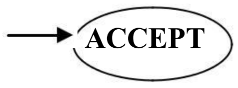
<u>Definition</u>

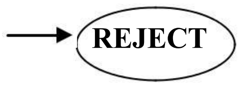
A **PDA** is a collection of <u>eight</u> things:

1. An alphabet ∑ of input letters.
2. An input TAPE (infinite in one direction). Initially the string of input letters is placed on the TAPE starting in cell i. The rest of the TAPE is blanks.
3. An alphabet Γof STACK characters.
4. A pushdown STACK (infinite in one direction). Initially the STACK is empty (contains all blanks)
5. One START state that has only out_adges, no in-edges.

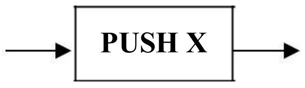


6. HALT states of two kinds: some ACCEPT and some REJECT they have in-edges and no out-edges.

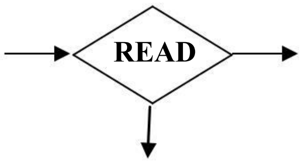




7. Finitely many non branching PUSH states that introduce characters onto the top of the STACK. they are of the form:

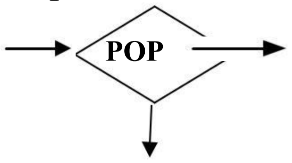


   Where X is any letter in Γ.
8. Finitely many branching states of two kinds:
   i. States that read the next unused letter from the TAPE.



   Which may have out-edges labeled with letters from ∑ and the blank character Δ, with no restrictions on duplication of labels and no insistence that there be a label for each letter of ∑, or Δ.

   ii. States that read the top character of STACK.



   Which may have out-edges labeled with letters from Γand the blank character Δ, again with no restrictions.

Note: we require that the states be connected so as to become a connected directed graph.

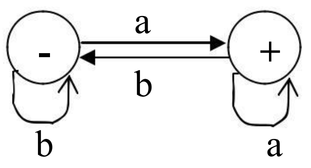
Theorem

For every regular language L there is some PDA that accepts it.

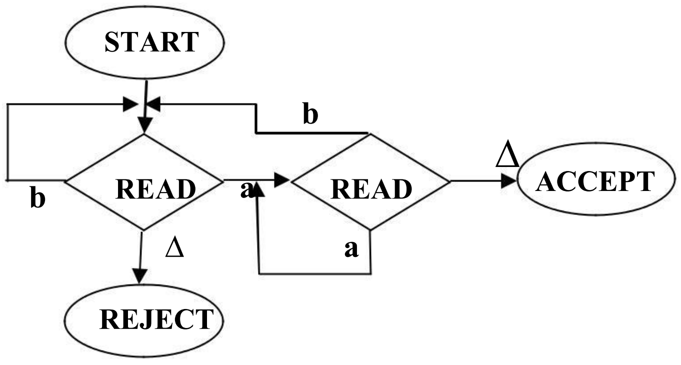
Proof

Since L is regular, so it is accepted by some FA, then we can convert FA to PDA (as in the following example).

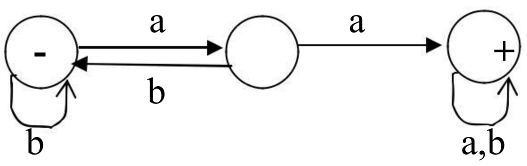
Example



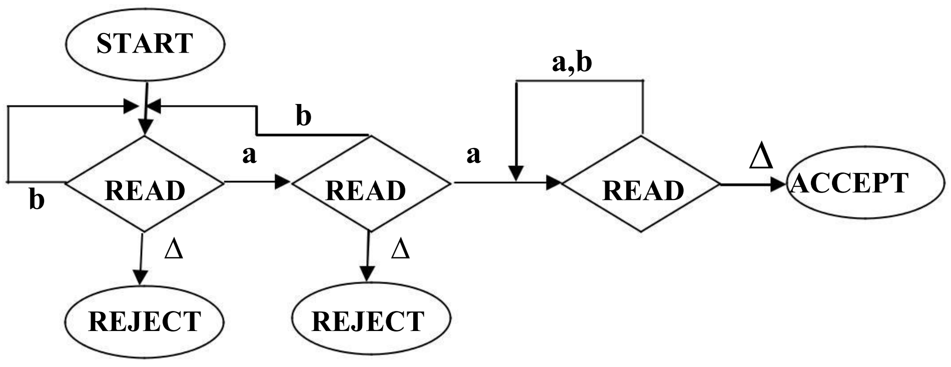
Becomes:



Example



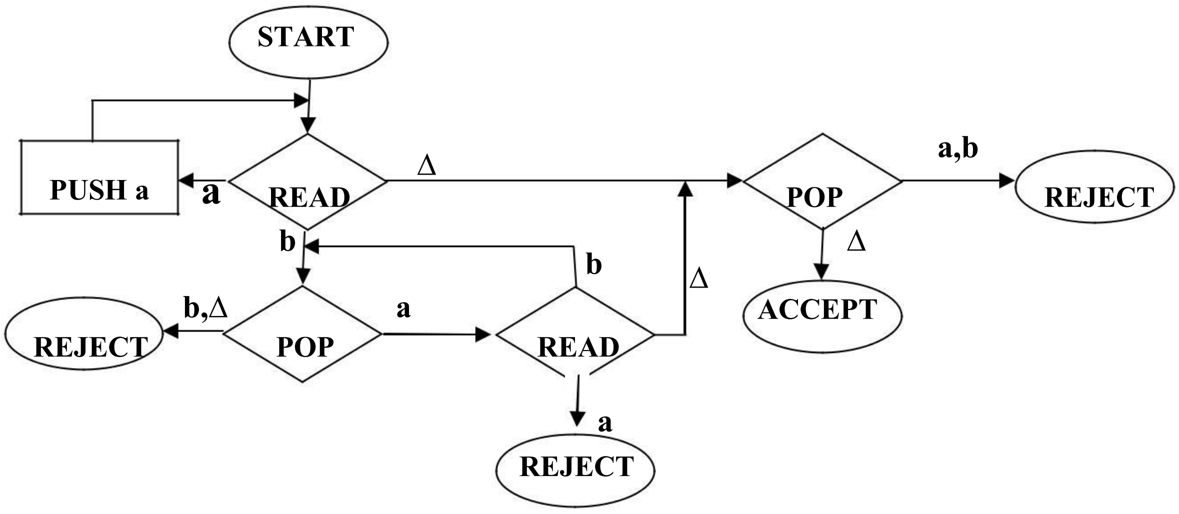
Becomes:



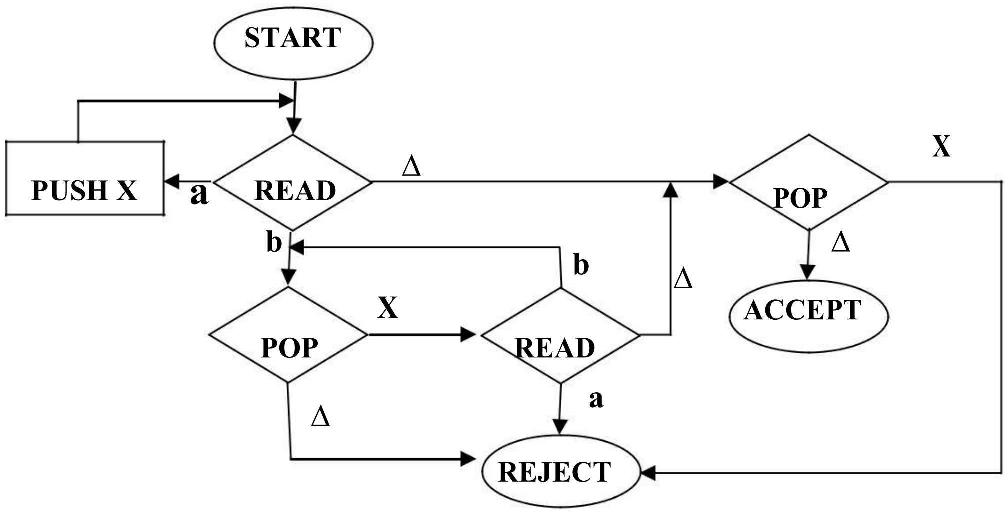
Note: we can find PDA accepts some non regular languages(as in the following example).

Example

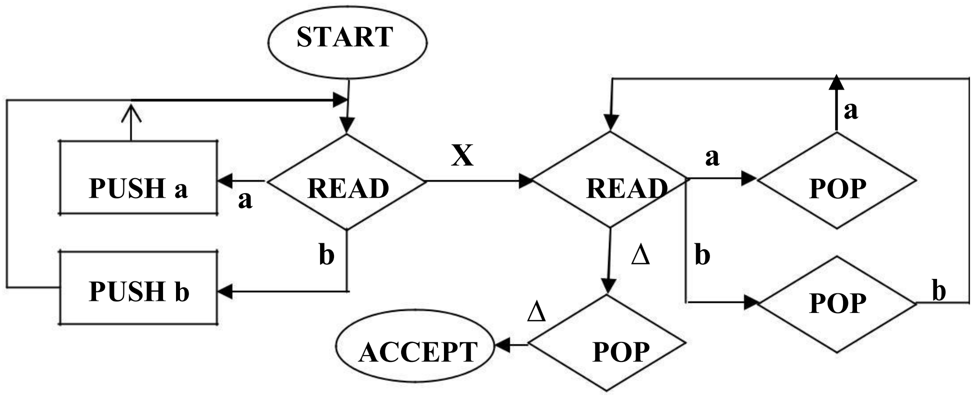
The language accepted by this PDA is exactly: $\{a^nb^n, n=0,1,2,\ldots\}$



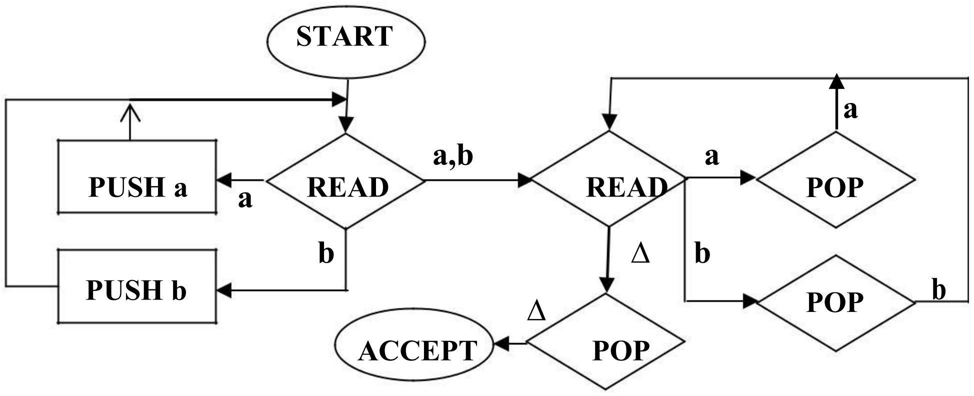
Or



Example
Consider the palindrome X, language of all words of the form:
sXreverese(s), where s is any string in (a+b)*, such as {X aXa bXb aaXaa abXba aabXbaa …}

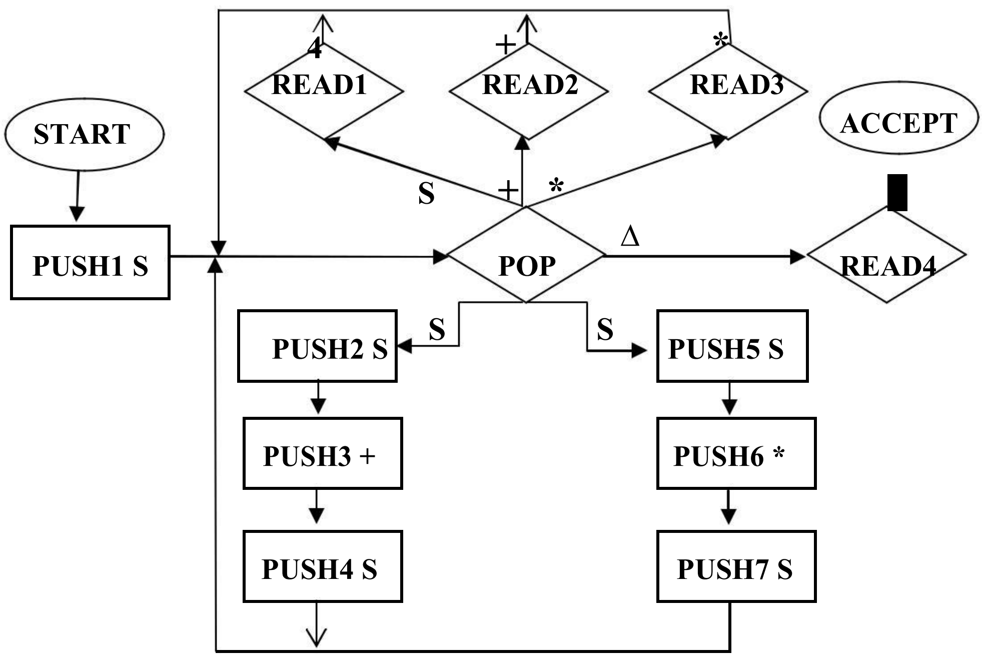


Example
odd palindrome ={a b aaa aba bab bbb …}



Nondeterministic PDA

Example
Consider the language generated by CFG: **S→S+S|S*S|4**



Now we trace the acceptance of the string: **4+4*4**

State	**Stack**	**Tape**
start	Δ	4+4*4
push1 S	S	4+4*4
pop	Δ	4+4*4
push2 S	S	4+4*4
push3 +	+ S	4+4*4
push4 S	S+ S	4+4*4
pop	+ S	4+4*4
read1	+ S	+4*4
pop	S	+4*4
read2	S	4*4
pop	Δ	4*4
push5 S	S	4*4
push6 *	* S	4*4
push7 S	S * S	4*4
pop	*S	4*4
read1	*S	*4
pop	S	*4
read3	S	4
pop	Δ	4
read1	Δ	Δ
pop	Δ	Δ
read4	Δ	Δ
accept	Δ	Δ

<u>H.W</u>

Find a PDA that accepts the language: $\{a^m b^n a^n$, m=1,2,3,…,n=1,2,3,…}

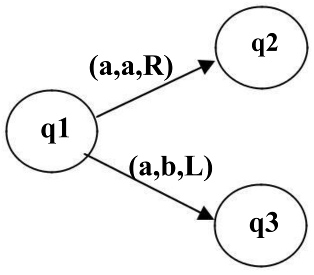
TURING MACHINE

Definition

A Turing machine (**TM**) is a collection of six things:

1. An alphabet ∑ of input letters.
2. A TAPE divided into a sequence of numbered cells each containing one character or a blank.
3. A TAPE HEAD that can in one step read the contains of a cell on the TAPE, replace it with some other character, and reposition itself to the next cell to the right or to the left of the one it has just read.
4. An alphabet Γof character that can be printed on the TAPE by the TAPE HEAD.
5. A finite set of states including exactly one START state from which we begin execution, and some (may be none) HALT states that cause execution to terminate when we enter them. The other states have no functions, only names: q1, q2, … or 1, 2, 3, …
6. A program, which is a set of rules that tell us on the basis of the letter the TAPE HEAD has just read, how to change states, what to print and where to move the TAPE HEAD. We depict the program as a collection of directed edge connecting the states. Each edge is labeled with a triplet of information: (letter, letter, direction). The first letter (either Δ or from ∑ or Γ) is the character that the TAPE HEAD reads from the cell to which it is pointing, the second letter (also Δ or from Γ) is what the TAPE HEAD prints in the cell before it leaves, the third component, the direction, tells the TAPE HEAD whether to move one cell to the right(R) or to the left (L).

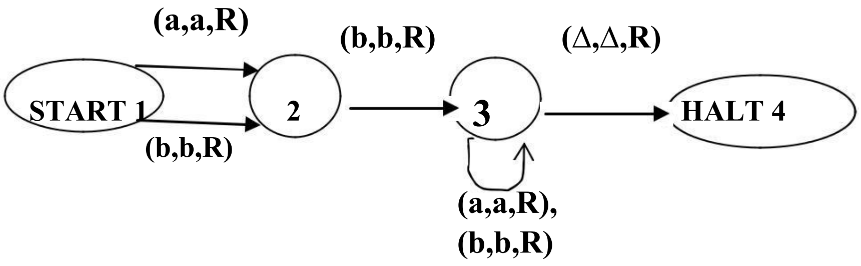
Note: TM is deterministic. This means that there is no state q that has two or more edges leaving it labeled with the same first letter. For example, the following TM is not allowed:



Example

Find TM that can accepts the language defined by the regular expression:

(a+b)b(a+b)*



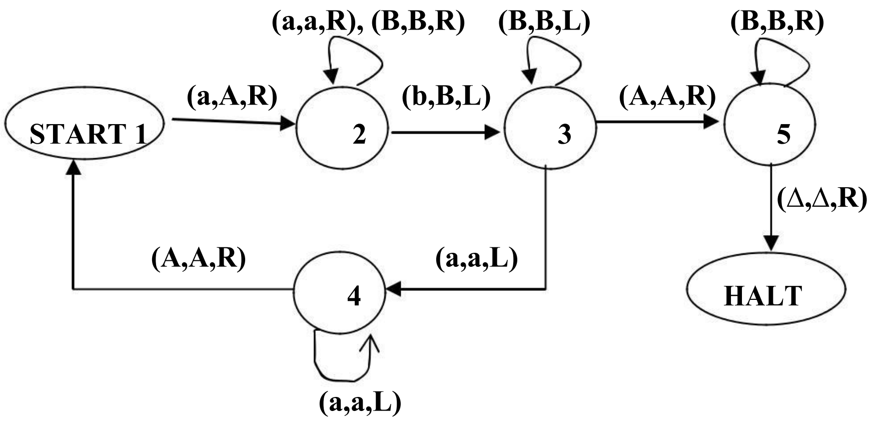
Now we trace the acceptance of the string: **aba**

$1 \rightarrow 2 \rightarrow 3 \rightarrow 3 \rightarrow 4$

$\underline{a}ba$ $a\underline{b}a$ $ab\underline{a}\Delta$ $aba\underline{\Delta}$ $aba\Delta\underline{\Delta}$

Example

Find TM that can accepts the language $\{a^nb^n\}$



Now we trace the acceptance of the string: **aaabbb**

$\underline{a}$aabbb A$\underline{a}$abbb Aa$\underline{a}$bbb Aaa$\underline{b}$bb Aa$\underline{a}$Bbb A$\underline{a}$aBbb $\underline{A}$aaBbb A$\underline{a}$aBbb

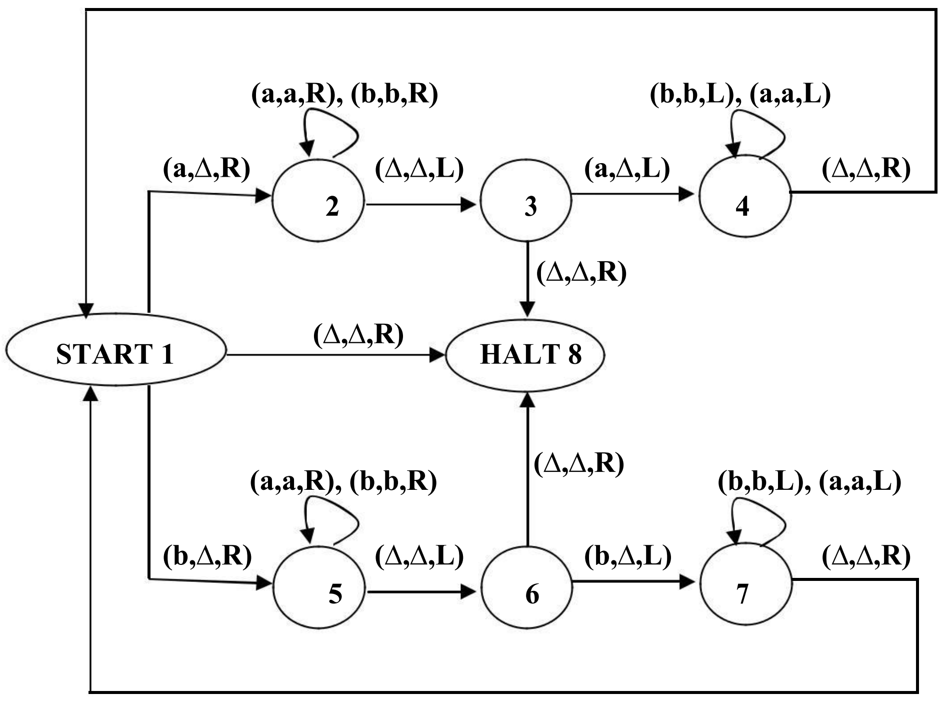
Aa$\underline{a}$Bbb Aaa$\underline{B}$bb AaaB$\underline{b}$b AAa$\underline{B}$Bb AA$\underline{a}$BBb A$\underline{A}$aBBb AA$\underline{a}$BBb

AAA$\underline{B}$Bb AAAB$\underline{B}$b AAABB$\underline{b}$ AAAB$\underline{B}$B AAA$\underline{B}$BB AA$\underline{A}$BBB

AAA$\underline{B}$BB AAAB$\underline{B}$B AAABB$\underline{B}$ AAABBB$\underline{\Delta}$ HALT

Example

Find TM that can accepts the language palindrome.



Let us trace the running of this TM on the input string: ababa

1 → 2 → 2 → 2 → 2 → 2 → 3 →

$\underline{a}$baba Δ$\underline{b}$aba Δb$\underline{a}$ba Δba$\underline{b}$a Δbab$\underline{a}$ Δbaba$\underline{\Delta}$ Δbab$\underline{a}$

4 → 4 → 4 → 4 → 1 → 5 → 5 →

Δba$\underline{b}$Δ Δba$\underline{b}$Δ Δ$\underline{b}$abΔ $\underline{\;}$ΔbabΔ Δ$\underline{b}$abΔ ΔΔ$\underline{a}$bΔ ΔΔa$\underline{b}$Δ

5 → 6 → 7 → 7 → 1 → 2 → 3 →8

ΔΔab$\underline{\Delta}$ ΔΔab$\underline{\Delta}$ ΔΔ$\underline{a}$ΔΔ ΔΔ$\underline{a}$ΔΔ ΔΔ$\underline{a}$ΔΔ ΔΔΔΔ$\underline{\Delta}$ ΔΔ$\underline{\Delta}$ΔΔ HALT

H.W

Find TM for even-even.