AUTOMATA THEORY AND COMPILER DESIGN

UNIT -I

Theory of Automata

Theory of automata is a theoretical branch of computer science and mathematical. It is the study of abstract machines and the computation problems that can be solved using these machines. The abstract machine is called the automata. The main motivation behind developing the automata theory was to develop methods to describe and analyse the dynamic behaviour of discrete systems.

This automaton consists of states and transitions. The **State** is represented by **circles**, and the **Transitions** is represented by **arrows**.

Automata is the kind of machine which takes some string as input and this input goes through a finite number of states and may enter in the final state.

There are the basic terminologies that are important and frequently used in automata:

Symbols:
Symbols are an entity or individual objects, which can be any letter, alphabet or any picture.

Example:
1, a, b, #

Alphabets:
Alphabets are a finite set of symbols. It is denoted by ∑.

Examples:
1. ∑ = {a, b}
2.
3. ∑ = {A, B, C, D}
4.
5. ∑ = {0, 1, 2}
6.
7. ∑ = {0, 1,......, 5]
8.
9. ∑ = {#, β, Δ}

String:
It is a finite collection of symbols from the alphabet. The string is denoted by w.

Example 1:
If ∑ = {a, b}, various string that can be generated from ∑ are {ab, aa, aaa, bb, bbb, ba, aba. ... }.

- A string with zero occurrences of symbols is known as an empty string. It is represented by ε.
- The number of symbols in a string w is called the length of a string. It is denoted by |w|.

Example 2:

1. w = 010
2.
3. Number of Sting |w| = 3

Language:

A language is a collection of appropriate string. A language which is formed over Σ can be **Finite** or **Infinite**.

Example: 1

L1 = {Set of string of length 2}

= {aa, bb, ba, bb} ***Finite Language***

Example: 2

L2 = {Set of all strings starts with 'a'}

= {a, aa, aaa, abb, abbb, ababb} ***Infinite Language***

Finite Automata

- Finite automata are used to recognize patterns.
- It takes the string of symbol as input and changes its state accordingly. When the desired symbol is found, then the transition occurs.
- At the time of transition, the automata can either move to the next state or stay in the same state.
- Finite automata have two states, **Accept state** or **Reject state**. When the input string is processed successfully, and the automata reached its final state, then it will accept.

Formal Definition of FA

A finite automaton is a collection of 5-tuple (Q, ∑, δ, q0, F), where:

1. Q: finite set of states
2. ∑: finite set of the input symbol
3. q0: initial state
4. F: **final** state
5. δ: Transition function

Finite Automata Model(structure)

Finite automata can be represented by input tape and finite control.

Input tape: It is a linear tape having some number of cells. Each input symbol is placed in each cell.

Finite control: The finite control decides the next state on receiving particular input from input tape. The tape reader reads the cells one by one from left to right, and at a time only one input symbol is read.

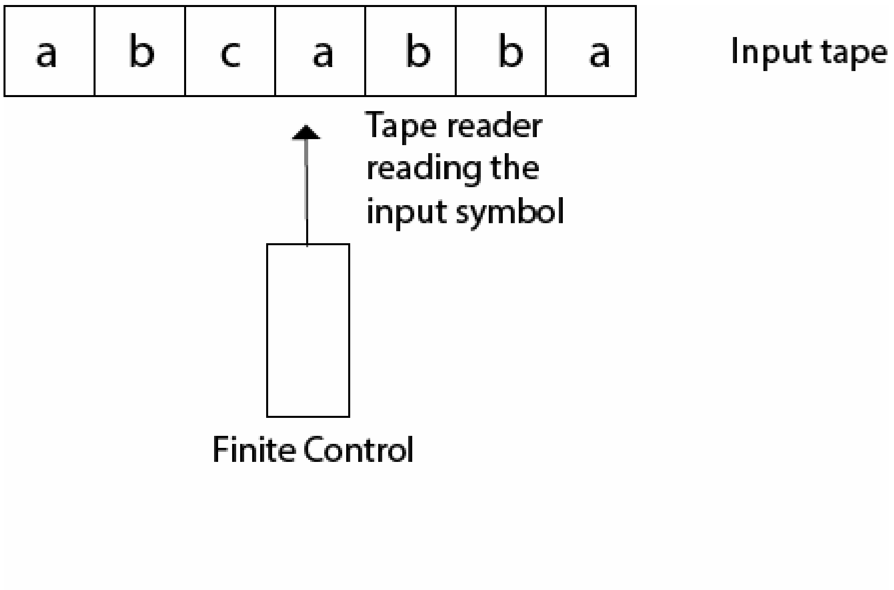
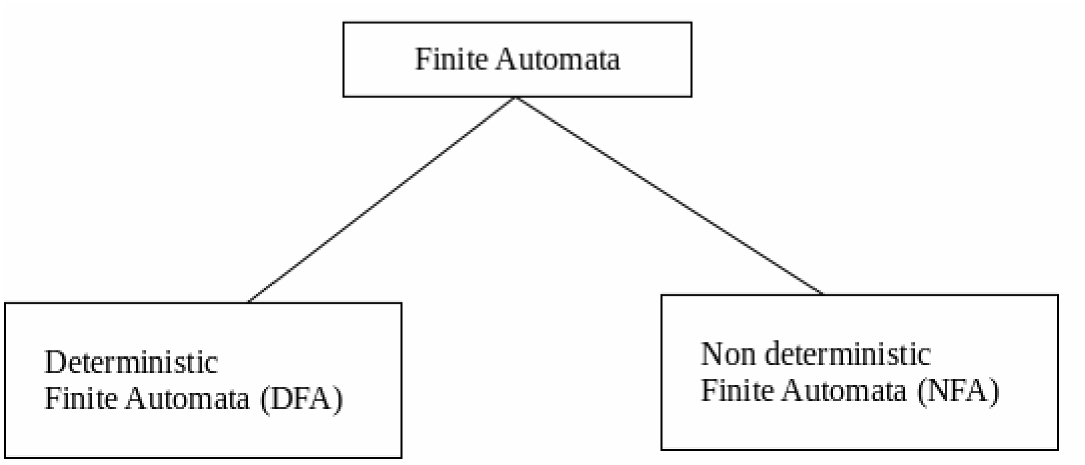


Fig :- Finite automata model

Types of Automata:
There are two types of finite automata:

1. DFA(deterministic finite automata)
2. NFA(non-deterministic finite automata)


Finite Automata
Deterministic Finite Automata (DFA)
Non deterministic Finite Automata (NFA)

1. DFA

DFA refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. In the DFA, the machine goes to one state only for a particular input character. DFA does not accept the null move.

2. NFA

NFA stands for non-deterministic finite automata. It is used to transmit any number of states for a particular input. It can accept the null move.

Some important points about DFA and NFA:

1. Every DFA is NFA, but NFA is not DFA.
2. There can be multiple final states in both NFA and DFA.
3. DFA is used in Lexical Analysis in Compiler.
4. NFA is more of a theoretical concept.

Transition Diagram

A transition diagram or state transition diagram is a directed graph which can be constructed as follows:

- There is a node for each state in Q, which is represented by the circle.
- There is a directed edge from node q to node p labeled a if δ(q, a) = p.
- In the start state, there is an arrow with no source.
- Accepting states or final states are indicating by a double circle.

Some Notations that are used in the transition diagram:

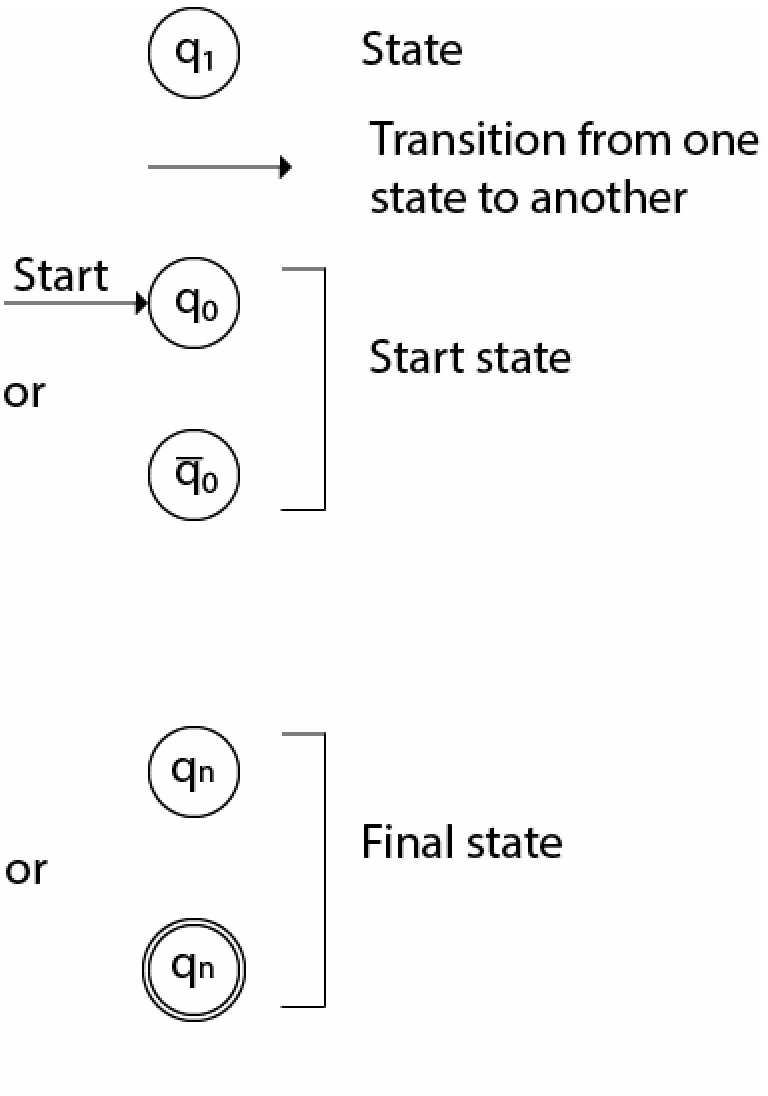


Fig:- Notations

There is a description of how a DFA operates:

1. In DFA, the input to the automata can be any string. Now, put a pointer to the start state q and read the input string w from left to right and move the pointer according to the transition function, δ. We can read one symbol at a time. If the next symbol of string w is a and the pointer is on state p, move the pointer to δ(p, a). When the end of the input string w is encountered, then the pointer is on some state F.

2. The string w is said to be accepted by the DFA if r F that means the input string w is processed successfully and the automata reached its final state. The string is said to be rejected by DFA if r F.

Example 1:
DFA with ∑ = {0, 1} accepts all strings starting with 1.

Solution:

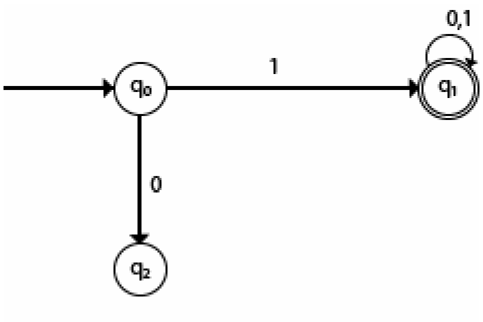
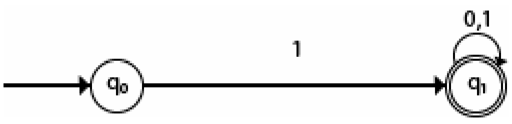


Fig: Transition diagram

The finite automata can be represented using a transition graph. In the above diagram, the machine initially is in start state q0 then on receiving input 1 the machine changes its state to q1. From q0 on receiving 0, the machine changes its state to q2, which is the dead state. From q1 on receiving input 0, 1 the machine changes its state to q1, which is the final state. The possible input strings that can be generated are 10, 11, 110, 101, 111......., that means all string starts with 1.

Example 2:
NFA with ∑ = {0, 1} accepts all strings starting with 1.

Solution:



he NFA can be represented using a transition graph. In the above diagram, the machine initially is in start state q0 then on receiving input 1 the machine changes its state to q1. From q1 on receiving input 0, 1 the machine changes its state to q1. The possible input string that can be generated is 10, 11, 110, 101, 111. , that means all string starts with 1.

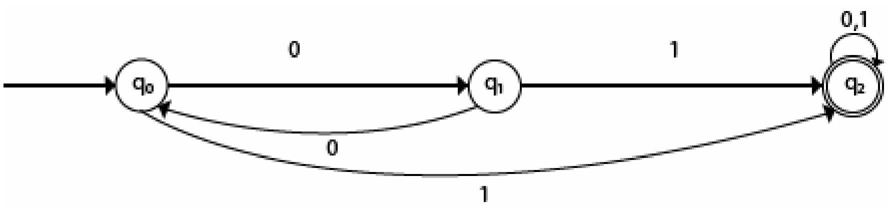
Transition Table

The transition table is basically a tabular representation of the transition function. It takes two arguments (a state and a symbol) and returns a state (the "next state").

A transition table is represented by the following things:

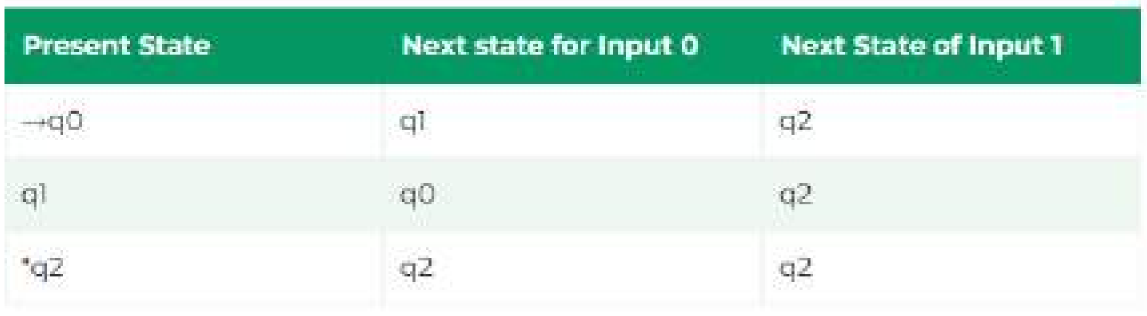
- Columns correspond to input symbols.
- Rows correspond to states.
- Entries correspond to the next state.
- The start state is denoted by an arrow with no source.
- The accept state is denoted by a star.

Example 1:



Solution:

Transition table of given DFA is as follows:

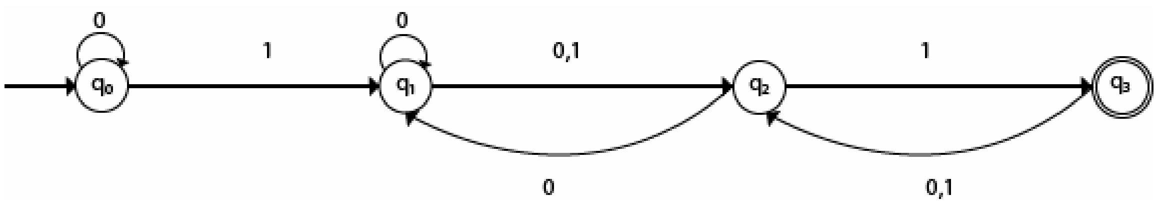


Present State	Next state for Input 0	Next State of Input 1
→q0	q1	q2
q1	q0	q2
*q2	q2	q2

Explanation:

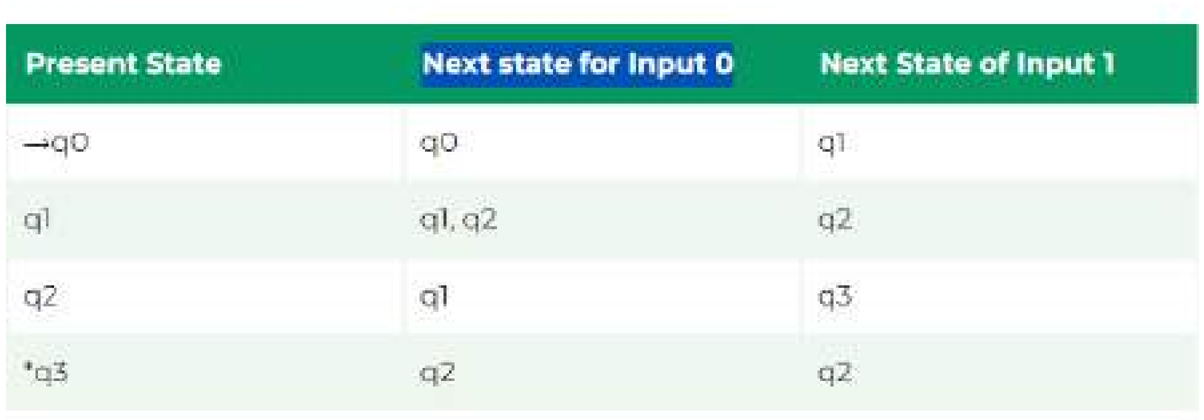
- In the above table, the first column indicates all the current states. Under column 0 and 1, the next states are shown.
- The first row of the transition table can be read as, when the current state is q0, on input 0 the next state will be q1 and on input 1 the next state will be q2.
- In the second row, when the current state is q1, on input 0, the next state will be q0, and on 1 input the next state will be q2.
- In the third row, when the current state is q2 on input 0, the next state will be q2, and on 1 input the next state will be q2.
- The arrow marked to q0 indicates that it is a start state and circle marked to q2 indicates that it is a final state.

Example 2:



Solution:

Transition table of given DFA is as follows:



Present State	Next state for Input 0	Next State of Input 1
→q0	q0	q1
q1	q1, q2	q2
q2	q1	q3
*q3	q2	q2

Explanation:

- The first row of the transition table can be read as, when the current state is q0, on input 0 the next state will be q0 and on input 1 the next state will be q1.
- In the second row, when the current state is q1, on input 0 the next state will be either q1 or q2, and on 1 input the next state will be q2.
- In the third row, when the current state is q2 on input 0, the next state will be q1, and on 1 input the next state will be q3.
- In the fourth row, when the current state is q3 on input 0, the next state will be q2, and on 1 input the next state will be q2.

DFA (Deterministic finite automata)

- DFA refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. The finite automata are called deterministic finite automata if the machine is read an input string one symbol at a time.
- In DFA, there is only one path for specific input from the current state to the next state.
- DFA does not accept the null move, i.e., the DFA cannot change state without any input character.
- DFA can contain multiple final states. It is used in Lexical Analysis in Compiler.

In the following diagram, we can see that from state q0 for input a, there is only one path which is going to q1. Similarly, from q0, there is only one path for input b going to q2.

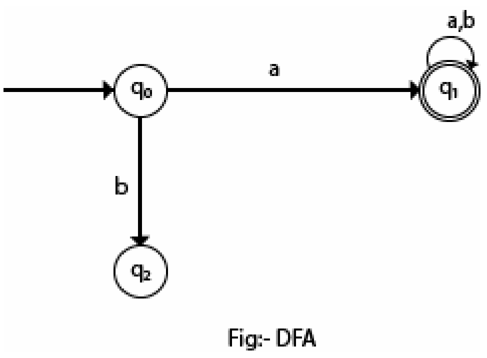


Fig:- DFA

Formal Definition of DFA

A DFA is a collection of 5-tuples same as we described in the definition of FA.

1. Q: finite set of states
2. ∑: finite set of the input symbol
3. q0: initial state
4. F: **final** state
5. δ: Transition function

Transition function can be defined as:

1. δ: Q x ∑→Q

Graphical Representation of DFA

A DFA can be represented by digraphs called state diagram. In which:

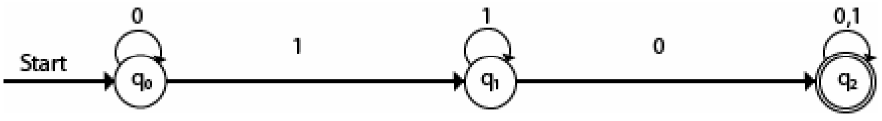
1. The state is represented by vertices.
2. The arc labeled with an input character show the transitions.
3. The initial state is marked with an arrow.
4. The final state is denoted by a double circle.

Example 1:

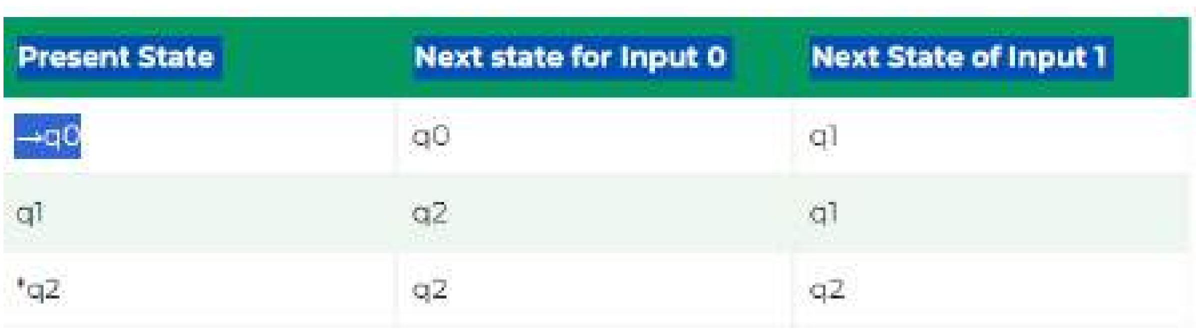
1. Q = {q0, q1, q2}
2. ∑ = {0, 1}
3. q0 = {q0}
4. F = {q2}

Solution:

Transition Diagram:



Transition Table:

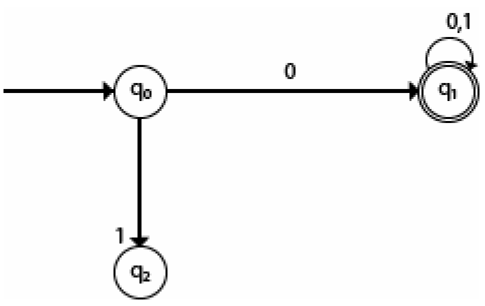


Present State	Next state for Input 0	Next State of Input 1
→q0	q0	q1
q1	q2	q1
*q2	q2	q2

Example 2:

DFA with ∑ = {0, 1} accepts all starting with 0.

Solution:



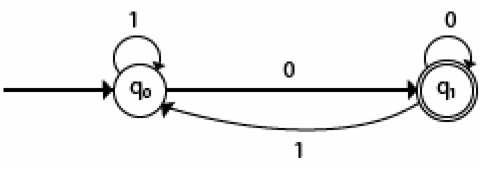
Explanation:

- In the above diagram, we can see that on given 0 as input to DFA in state q0 the DFA changes state to q1 and always go to final state q1 on starting input 0. It can accept 00, 01, 000, 001. etc. It can't accept any string which starts with 1, because it will never go to final state on a string starting with 1.

Example 3:

DFA with ∑ = {0, 1} accepts all ending with 0.

Solution:



Explanation:

In the above diagram, we can see that on given 0 as input to DFA in state q0, the DFA changes state to q1. It can accept any string which ends with 0 like 00, 10, 110, 100....etc. It can't accept any string which ends with 1, because it will never go to the final state q1 on 1 input, so the string ending with 1, will not be accepted or will be rejected.

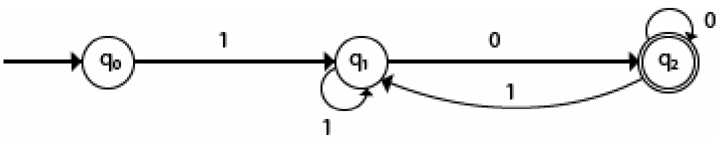
Examples of DFA

Example 1:

Design a FA with ∑ = {0, 1} accepts those string which starts with 1 and ends with 0.

Solution:

The FA will have a start state q0 from which only the edge with input 1 will go to the next state.



In state q1, if we read 1, we will be in state q1, but if we read 0 at state q1, we will reach to state q2 which is the final state. In state q2, if we read either 0 or 1, we will go to q2 state or q1 state respectively. Note that if the input ends with 0, it will be in the final state.

Example 2:

Design a FA with ∑ = {0, 1} accepts the only input 101.

Solution:

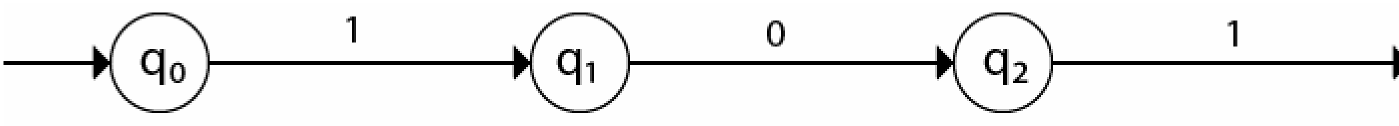


Fig: FA

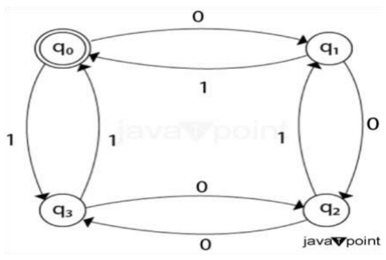
In the given solution, we can see that only input 101 will be accepted. Hence, for input 101, there is no other path shown for other input.

Example 3:

Design FA with ∑ = {0, 1} accepts even number of 0's and even number of 1's.

Solution:

This FA will consider four different stages for input 0 and input 1. The stages could be:



Here q0 is a start state and the final state also. Note carefully that a symmetry of 0's and 1's is maintained. We can associate meanings to each state as:

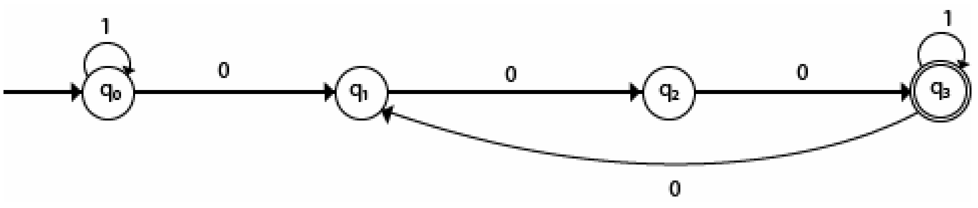
q0: state of even number of 0's and even number of 1's.
q1: state of odd number of 0's and even number of 1's.
q2: state of odd number of 0's and odd number of 1's.
q3: state of even number of 0's and odd number of 1's.

Example 4:

Design FA with ∑ = {0, 1} accepts the set of all strings with three consecutive 0's.

Solution:

The strings that will be generated for this particular languages are 000, 0001, 1000, 10001, ... in which 0 always appears in a clump of 3. The transition graph is as follows:



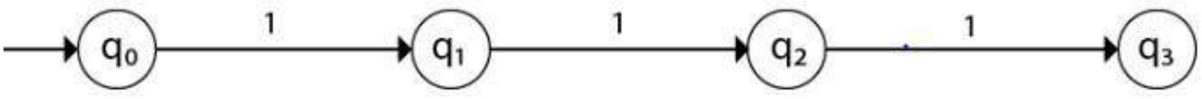
Note that the sequence of triple zeros is maintained to reach the final state.

Example 5:

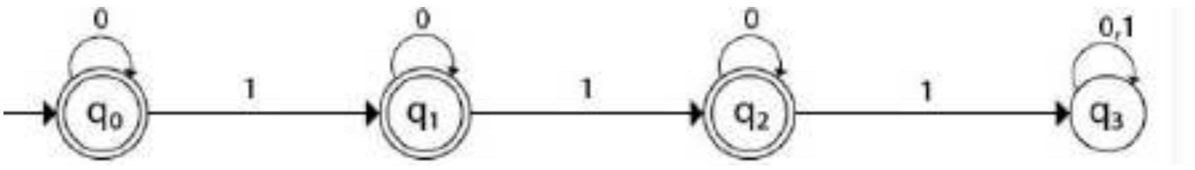
Design a DFA L(M) = {w | w ε {0, 1}*} and W is a string that does not contain consecutive 1's.

Solution:

When three consecutive 1's occur the DFA will be:



Here two consecutive 1's or single 1 is acceptable, hence



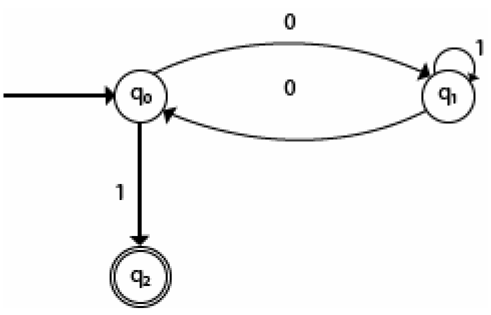
The stages q0, q1, q2 are the final states. The DFA will generate the strings that do not contain consecutive 1's like 10, 110, 101,..... etc.

Example 6:

Design a FA with ∑ = {0, 1} accepts the strings with an even number of 0's followed by single 1.

Solution:

The DFA can be shown by a transition diagram as:



NFA (Non-Deterministic finite automata)

- NFA stands for non-deterministic finite automata. It is easy to construct an NFA than DFA for a given regular language.
- The finite automata are called NFA when there exist many paths for specific input from the current state to the next state.
- Every NFA is not DFA, but each NFA can be translated into DFA.
- NFA is defined in the same way as DFA but with the following two exceptions, it contains multiple next states, and it contains ε transition.

In the following image, we can see that from state q0 for input a, there are two next states q1 and q2, similarly, from q0 for input b, the next states are q0 and q1. Thus it is not fixed or determined that with a particular input where to go next. Hence this FA is called non-deterministic finite automata.

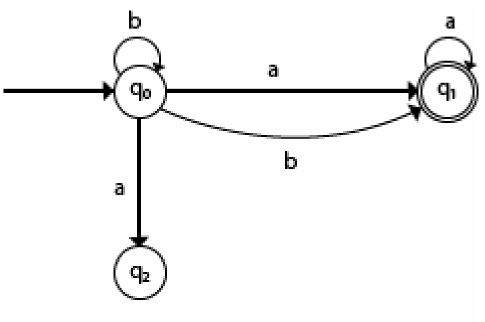


Fig:- NDFA

Formal definition of NFA:

NFA also has five states same as DFA, but with different transition function, as shown follows:

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

where,

1. Q: finite set of states
2. ∑: finite set of the input symbol
3. q0: initial state
4. F: **final** state

5. δ: Transition function

Graphical Representation of an NFA

An NFA can be represented by digraphs called state diagram. In which:

1. The state is represented by vertices.
2. The arc labeled with an input character show the transitions.
3. The initial state is marked with an arrow.
4. The final state is denoted by the double circle.

Example 1:

1. Q = {q0, q1, q2}
2. ∑ = {0, 1}
3. q0 = {q0}
4. F = {q2}

Solution:

Transition diagram:

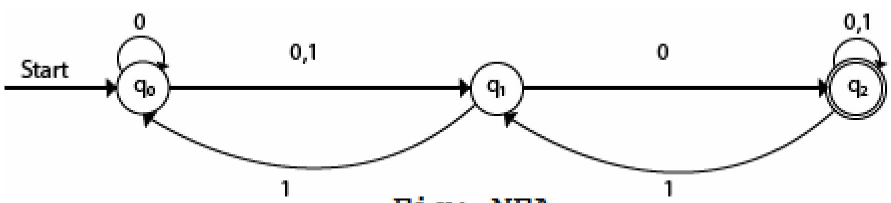


Fig: NFA

Transition Table:

Present State	Next state for Input 0	Next State of Input 1
→q0	q0, q1	q1
q1	q2	q0
*q2	q2	q1, q2

In the above diagram, we can see that when the current state is q0, on input 0, the next state will be q0 or q1, and on 1 input the next state will be q1. When the current state is q1, on input 0 the next state will be q2 and on 1 input, the next state will be q0. When the current state is q2, on 0 input the next state is q2, and on 1 input the next state will be q1 or q2.

Example 2:

NFA with ∑ = {0, 1} accepts all strings with 01.

Solution:

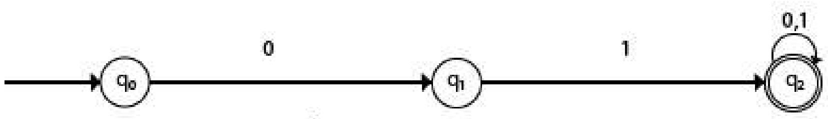


Fig: NFA

Transition Table:

Present State	Next state for Input 0	Next State of Input 1
→q0	q1	ε
q1	ε	q2
*q2	q2	q2

Example 3:

NFA with ∑ = {0, 1} and accept all string of length atleast 2.

Solution:

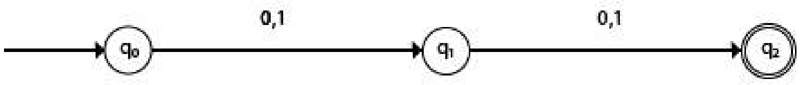
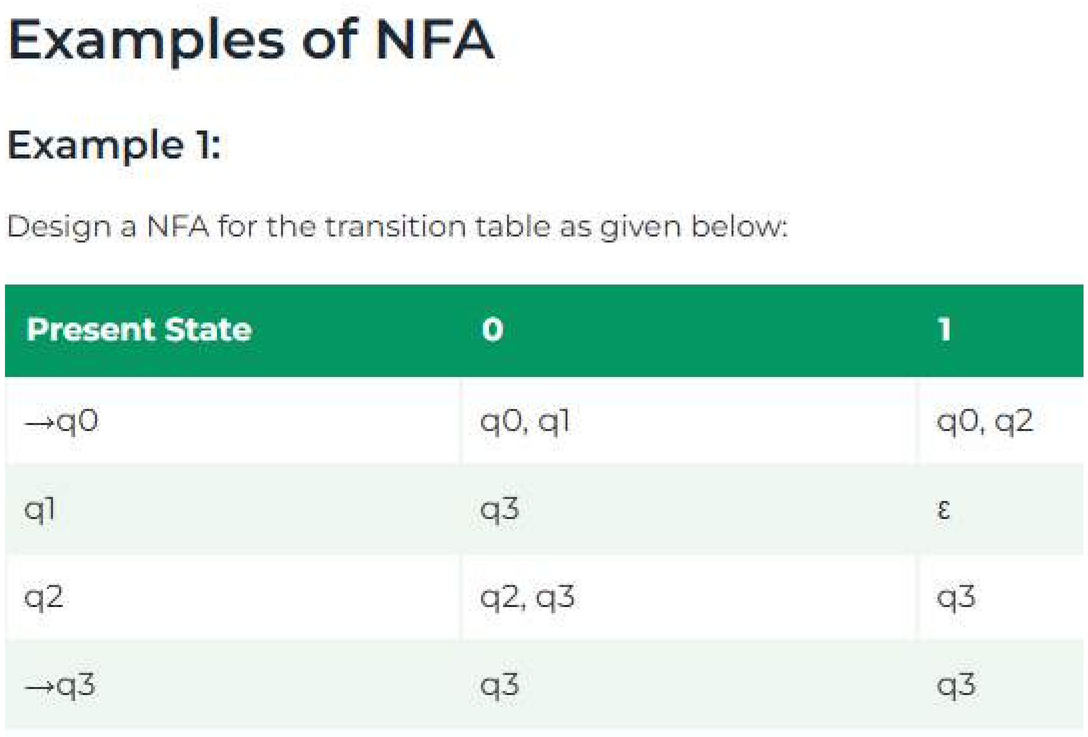


Fig: NFA

Transition Table:

Present State	Next state for Input 0	Next State of Input 1
→q0	q1	q1
q1	q2	q2
*q2	ε	ε



Examples of NFA

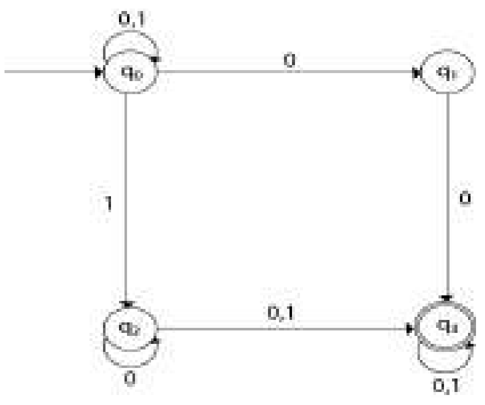
Example 1:

Design a NFA for the transition table as given below:

Present State	0	1
→q0	q0, q1	q0, q2
q1	q3	ε
q2	q2, q3	q3
→q3	q3	q3

Solution:

The transition diagram can be drawn by using the mapping function as given in the table.

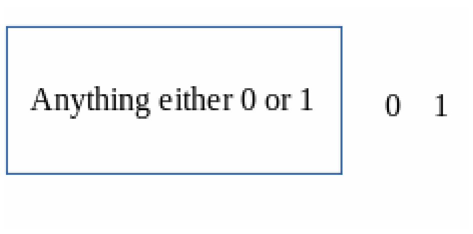


1. δ(q0, 0) = {q0, q1}
2. δ(q0, 1) = {q0, q2}
3. Then, δ(q1, 0) = {q3}
4. Then, δ(q2, 0) = {q2, q3}
5. δ(q2, 1) = {q3}
6. Then, δ(q3, 0) = {q3}
7. δ(q3, 1) = {q3}

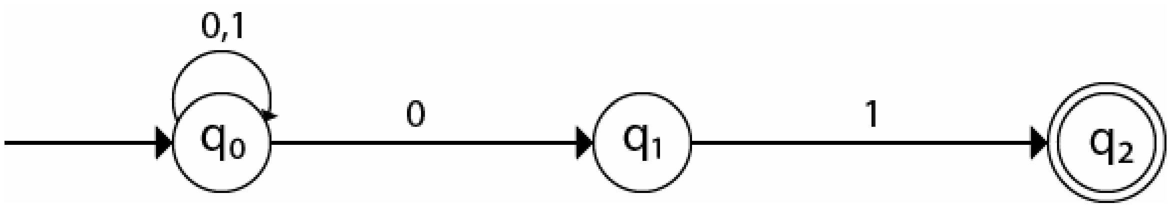
Example 2:

Design an NFA with ∑ = {0, 1} accepts all string ending with 01.

Solution:



Hence, NFA would be:



Example 3:

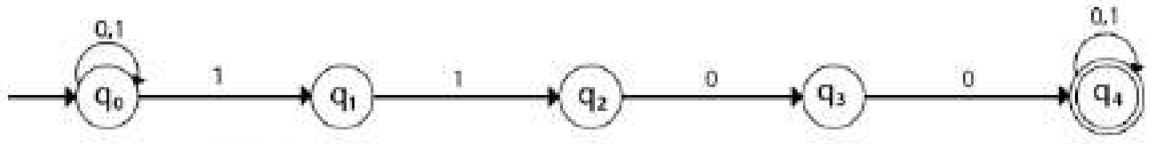
Design an NFA with ∑ = {0, 1} in which double '1' is followed by double '0'.

Solution:

The FA with double 1 is as follows:

Now before double 1, there can be any string of 0 and 1. Similarly, after double 0, there can be any string of 0 and 1.

Hence the NFA becomes:



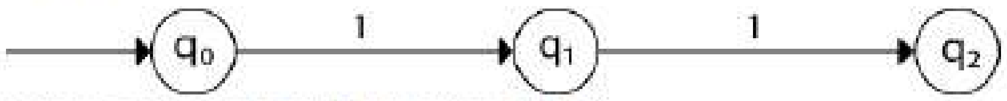
Now considering the string 01100011

q0 → q1 → q2 → q3 → q4 → q4 → q4 → q4

Example 4:

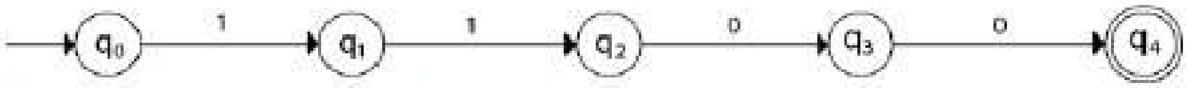
Design an NFA in which all the string contain a substring 1110.

Solution:



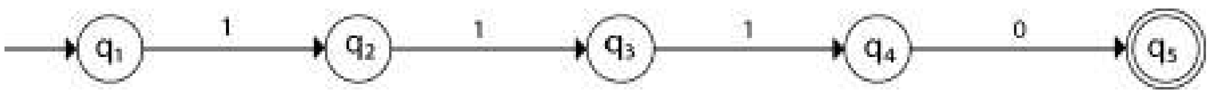
It should be immediately followed by double 0.

Then,

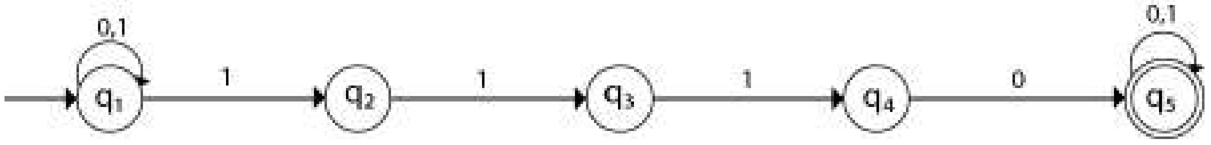


Now before double 1, there can be any string of 0 and 1. Similarly, after double 0, there can be any string of 0 and 1.

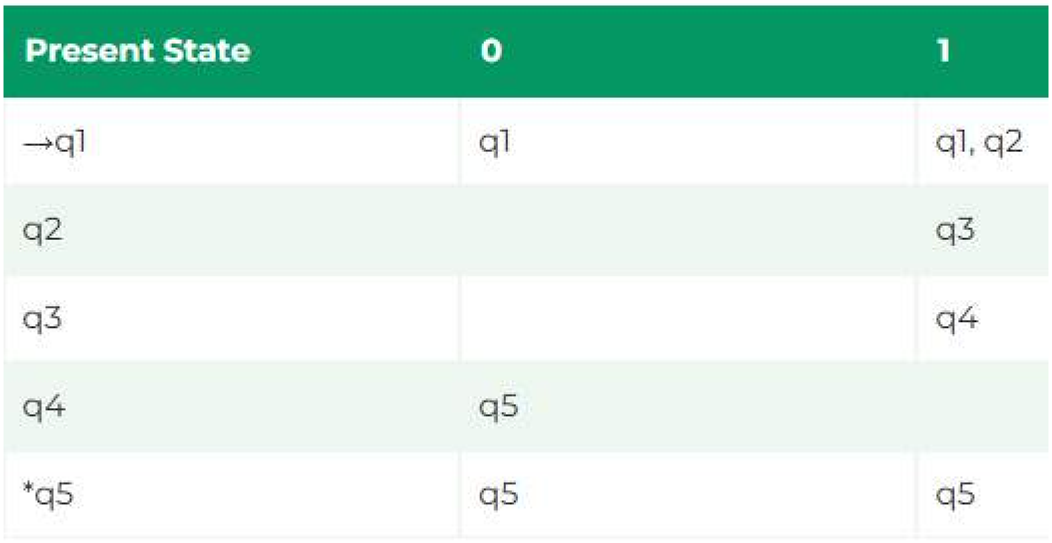
The language consists of all the string containing substring 1010. The partial transition diagram can be:



Now as 1010 could be the substring. Hence we will add the inputs 0's and 1's so that the substring 1010 of the language can be maintained. Hence the NFA becomes:



Transition table for the above transition diagram can be given below:



Present State	0	1
→q1	q1	q1, q2
q2		q3
q3		q4
q4	q5	
*q5	q5	q5

Consider a string 111010,

1. δ(q1, 111010) = δ(q1, 1100)
2. = δ(q1, 100)
3. = δ(q2, 00)

Got stuck! As there is no path from q2 for input symbol 0. We can process string 111010 in another way.

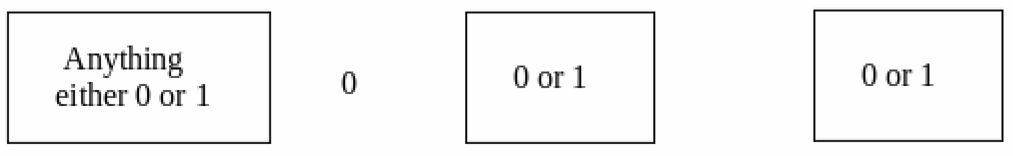
1. δ(q1, 111010) = δ(q2, 1100)
2. = δ(q3, 100)
3. = δ(q4, 00)
4. = δ(q5, 0)
5. = δ(q5, ε)

As state q5 is the accept state. We get the complete scanned, and we reached to the final state.

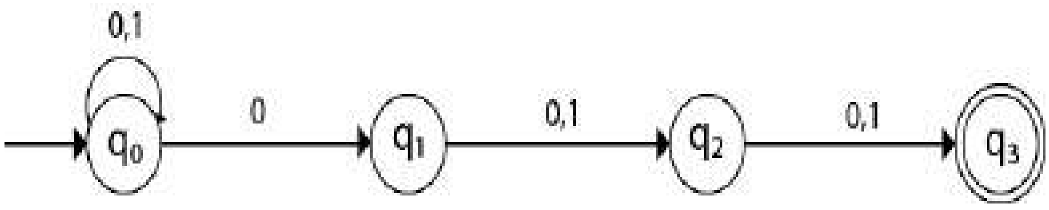
Example 5:

Design an NFA with ∑ = {0, 1} accepts all string in which the third symbol from the right end is always 0.

Solution:



Thus we get the third symbol from the right end as '0' always. The NFA can be:



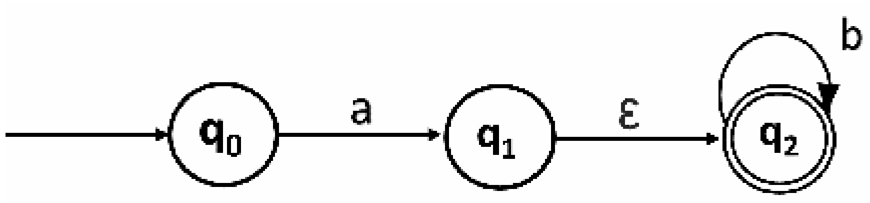
The above image is an NFA because in state q0 with input 0, we can either go to state q0 or q1.

Eliminating ε Transitions(Convert NFA with ε to NFA without ε.)

NFA with ε can be converted to NFA without ε, and this NFA without ε can be converted to DFA. To do this, we will use a method, which can remove all the ε transition from given NFA. The method will be:

1. Find out all the ε transitions from each state from Q. That will be called as ε-closure{q1} where qi   Q.
2. Then δ' transitions can be obtained. The δ' transitions mean a ε-closure on δ moves.
3. Repeat Step-2 for each input symbol and each state of given NFA.
4. Using the resultant states, the transition table for equivalent NFA without ε can be built.

Example:
Convert the following NFA with ε to NFA without ε.



Solutions: We will first obtain ε-closures of q0, q1 and q2 as follows:

1. ε-closure(q0) = {q0}
2. ε-closure(q1) = {q1, q2}
3. ε-closure(q2) = {q2}

Now the δ' transition on each input symbol is obtained as:

1. δ'(q0, a) = ε-closure(δ(δ^(q0, ε),a))
2. = ε-closure(δ(ε-closure(q0),a))
3. = ε-closure(δ(q0, a))
4. = ε-closure(q1)
5. = {q1, q2}
6.
7. δ'(q0, b) = ε-closure(δ(δ^(q0, ε),b))

8. = ε-closure(δ(ε-closure(q0),b))
9. = ε-closure(δ(q0, b))
10. = Φ

Now the δ' transition on q1 is obtained as:

1. δ'(q1, a) = ε-closure(δ(δ^(q1, ε),a))
2. = ε-closure(δ(ε-closure(q1),a))
3. = ε-closure(δ(q1, q2), a)
4. = ε-closure(δ(q1, a)   δ(q2, a))
5. = ε-closure(Φ   Φ)
6. = Φ
7.
8. δ'(q1, b) = ε-closure(δ(δ^(q1, ε),b))
9. = ε-closure(δ(ε-closure(q1),b))
10. = ε-closure(δ(q1, q2), b)
11. = ε-closure(δ(q1, b)   δ(q2, b))
12. = ε-closure(Φ   q2)
13. = {q2}

The δ' transition on q2 is obtained as:

1. δ'(q2, a) = ε-closure(δ(δ^(q2, ε),a))
2. = ε-closure(δ(ε-closure(q2),a))
3. = ε-closure(δ(q2, a))
4. = ε-closure(Φ)
5. = Φ
6.
7. δ'(q2, b) = ε-closure(δ(δ^(q2, ε),b))
8. = ε-closure(δ(ε-closure(q2),b))
9. = ε-closure(δ(q2, b))
10. = ε-closure(q2)
11. = {q2}

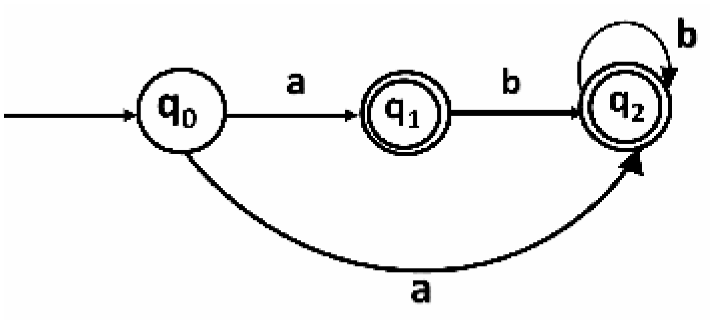
Now we will summarize all the computed δ' transitions:

1. δ'(q0, a) = {q0, q1}
2. δ'(q0, b) = Φ
3. δ'(q1, a) = Φ
4. δ'(q1, b) = {q2}
5. δ'(q2, a) = Φ
6. δ'(q2, b) = {q2}

The transition table can be:

States	a	b
→q0	{q1, q2}	Φ
*q1	Φ	{q2}
*q2	Φ	{q2}

State q1 and q2 become the final state as ε-closure of q1 and q2 contain the final state q2. The NFA can be shown by the following transition diagram:



Conversion from NFA to DFA

In this section, we will discuss the method of converting NFA to its equivalent DFA. In NFA, when a specific input is given to the current state, the machine goes to multiple states. It can have zero, one or more than one move on a given input symbol. On the other hand, in DFA, when a specific input is given to the current state, the machine goes to only one state. DFA has only one move on a given input symbol.

Let, M = (Q, ∑, δ, q0, F) is an NFA which accepts the language L(M). There should be equivalent DFA denoted by M' = (Q', ∑', q0', δ', F') such that L(M) = L(M').

Steps for converting NFA to DFA:

Step 1: Initially Q' = ϕ

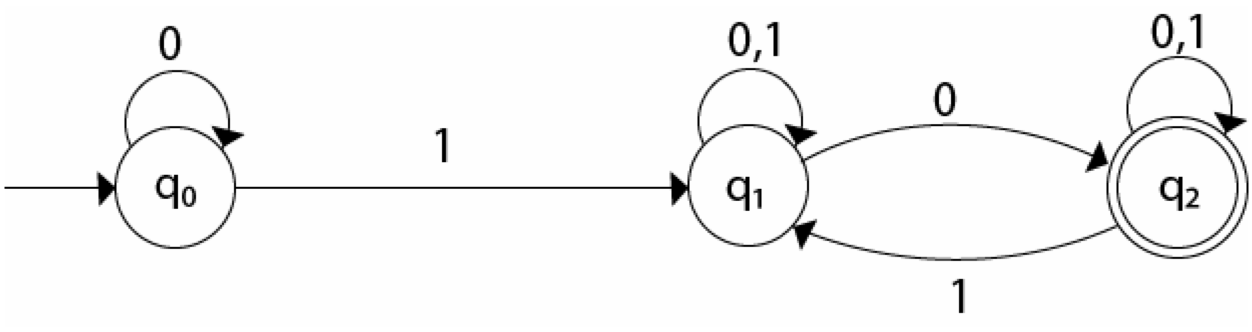
Step 2: Add q0 of NFA to Q'. Then find the transitions from this start state.

Step 3: In Q', find the possible set of states for each input symbol. If this set of states is not in Q', then add it to Q'.

Step 4: In DFA, the final state will be all the states which contain F(final states of NFA)

Example 1:

Convert the given NFA to DFA.



Solution: For the given transition diagram we will first construct the transition table.

State	0	1
→q0	q0	q1
q1	{q1, q2}	q1
*q2	q2	{q1, q2}

Now we will obtain δ' transition for state q0.

1. δ'([q0], 0) = [q0]
2. δ'([q0], 1) = [q1]

The δ' transition for state q1 is obtained as:

1. '([q1], 0) = [q1, q2]   (**new** state generated)
2. δ'([q1], 1) = [q1]

The δ' transition for state q2 is obtained as:

1. δ'([q2], 0) = [q2]
2. δ'([q2], 1) = [q1, q2]

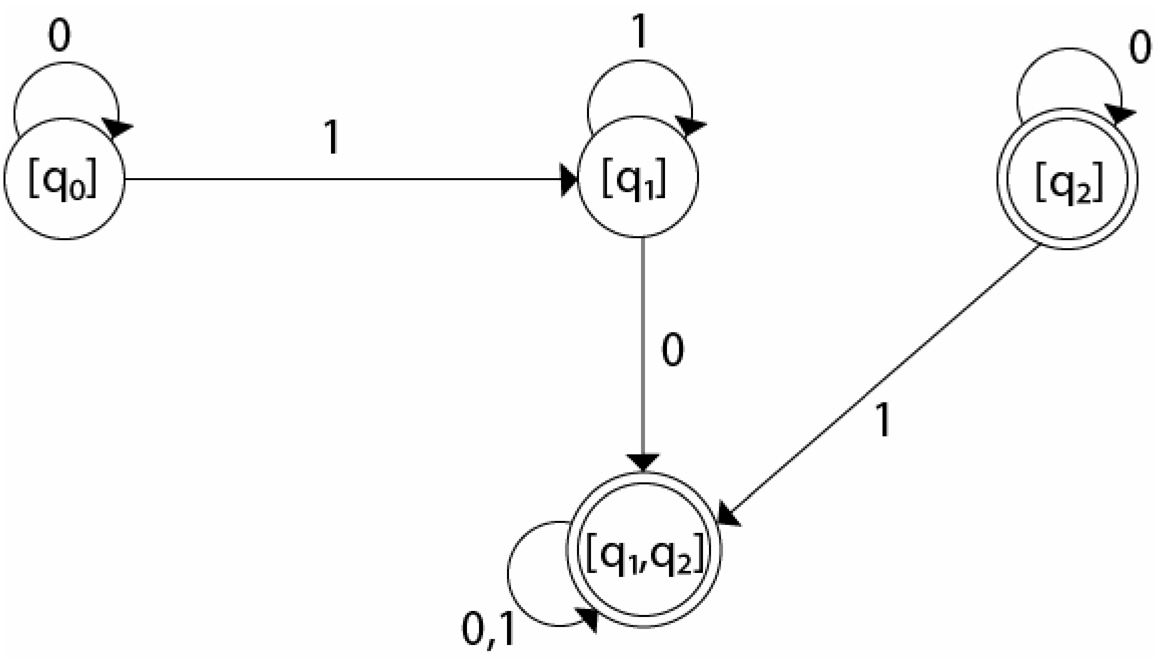
Now we will obtain δ' transition on [q1, q2].

1. δ'([q1, q2], 0) = δ(q1, 0)   δ(q2, 0)
2. = {q1, q2}   {q2}
3. = [q1, q2]
4. δ'([q1, q2], 1) = δ(q1, 1)   δ(q2, 1)
5. = {q1}   {q1, q2}
6. = {q1, q2}
7. = [q1, q2]

The state [q1, q2] is the final state as well because it contains a final state q2. The transition table for the constructed DFA will be:

State	0	1
→[q0]	[q0]	[q1]
[q1]	[q1, q2]	[q1]
*[q2]	[q2]	[q1, q2]
*[q1, q2]	[q1, q2]	[q1, q2]

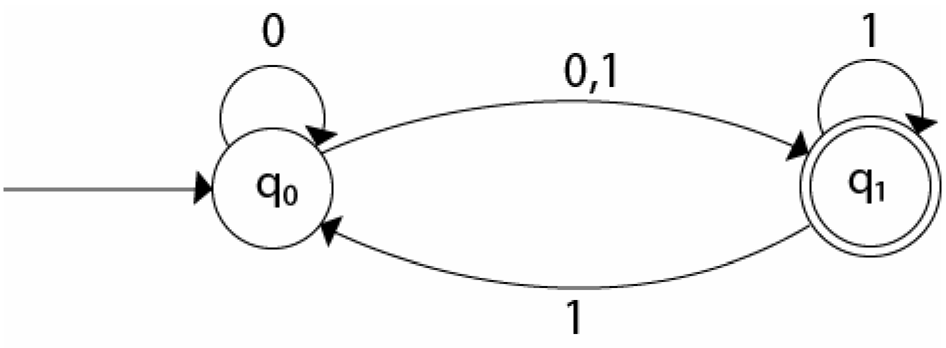
The Transition diagram will be:



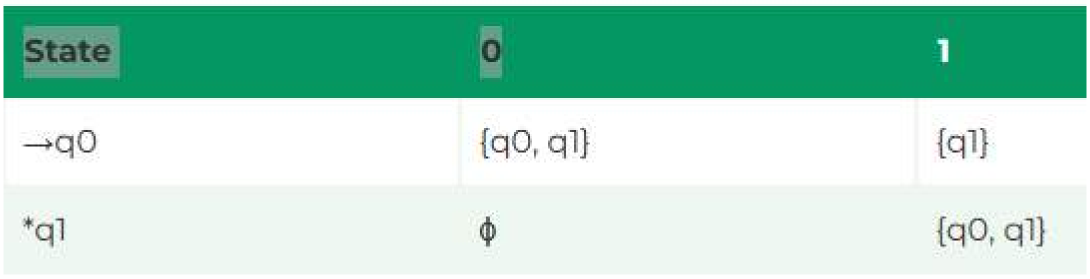
The state q2 can be eliminated because q2 is an unreachable state.

Example 2:

Convert the given NFA to DFA.



Solution: For the given transition diagram we will first construct the transition table.



State	0	1
→q0	{q0, q1}	{q1}
*q1	ϕ	{q0, q1}

Now we will obtain δ' transition for state q0.

1. δ'([q0], 0) = {q0, q1}
2. = [q0, q1]   (**new** state generated)
3. δ'([q0], 1) = {q1} = [q1]

The δ' transition for state q1 is obtained as:

1. δ'([q1], 0) = ϕ
2. δ'([q1], 1) = [q0, q1]

Now we will obtain δ' transition on [q0, q1].

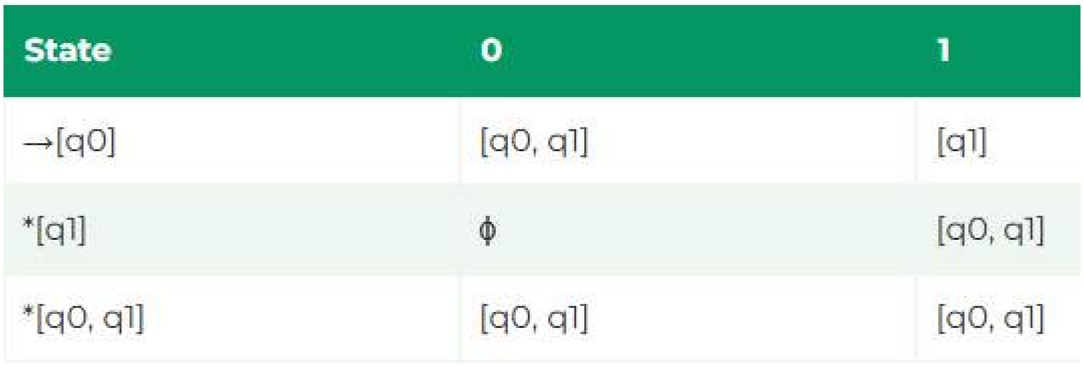
1. δ'([q0, q1], 0) = δ(q0, 0)   δ(q1, 0)
2. = {q0, q1}   ϕ
3. = {q0, q1}
4. = [q0, q1]

Similarly,

1. δ'([q0, q1], 1) = δ(q0, 1)   δ(q1, 1)
2. = {q1}   {q0, q1}
3. = {q0, q1}
4. = [q0, q1]

As in the given NFA, q1 is a final state, then in DFA wherever, q1 exists that state becomes a final state. Hence in the DFA, final states are [q1] and [q0, q1]. Therefore set of final states F = {[q1], [q0, q1]}.

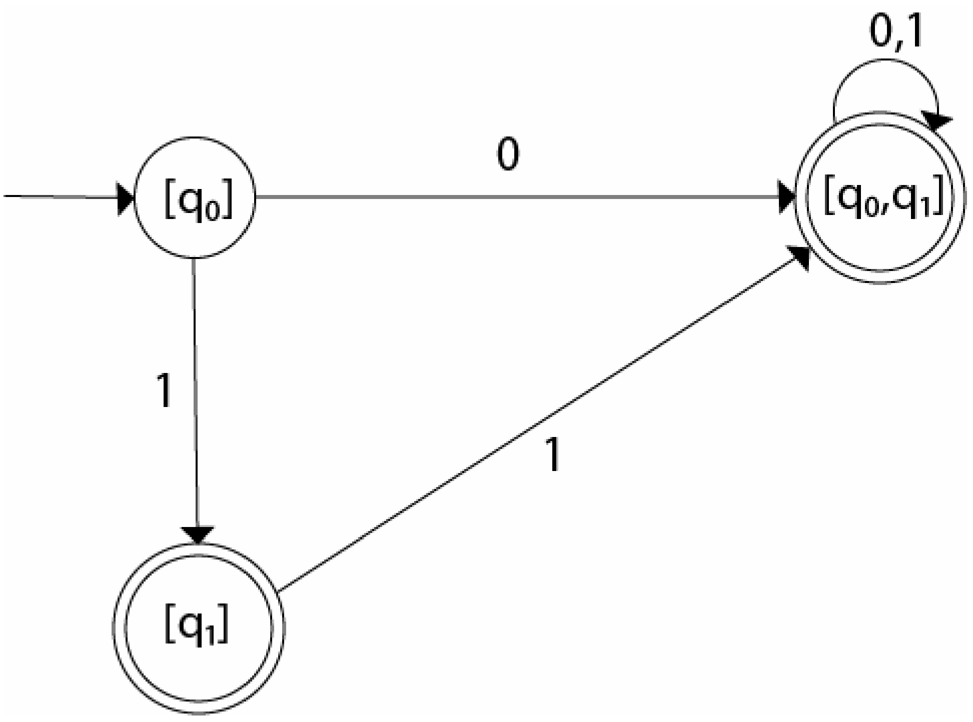
The transition table for the constructed DFA will be:



State	0	1
→[q0]	[q0, q1]	[q1]
*[q1]	ϕ	[q0, q1]
*[q0, q1]	[q0, q1]	[q0, q1]

The Transition diagram will be:

The Transition diagram will be:



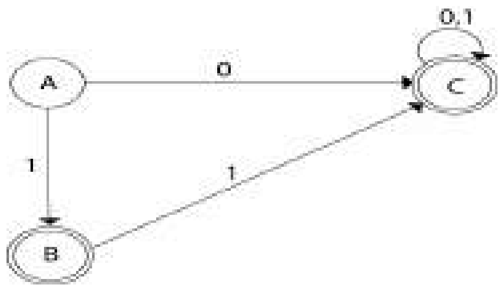
$$\begin{tikzcd}
\arrow[r] & {[q_0]} \arrow[rr, "0"] \arrow[d, "1"'] & & {[q_0,q_1]} \arrow[loop above, "0,1"] \\
& {[q_1]} \arrow[urr, "1"'] & &
\end{tikzcd}$$

Even we can change the name of the states of DFA.

Suppose

1. A = [q0]
2. B = [q1]
3. C = [q0, q1]

With these new names the DFA will be as follows:



Conversion from NFA with ε to DFA

Non-deterministic finite automata(NFA) is a finite automata where for some cases when a specific input is given to the current state, the machine goes to multiple states or more than 1 states. It can contain ε move. It can be represented as M = { Q, ∑, δ, q0, F}.

Where

1. Q: finite set of states
2. ∑: finite set of the input symbol
3. q0: initial state
4. F: **final** state
5. δ: Transition function

NFA with move: If any FA contains ε transaction or move, the finite automata is called NFA with move.

ε-closure: ε-closure for a given state A means a set of states which can be reached from the state A with only ε(null) move including the state A itself.

Steps for converting NFA with ε to DFA:

Step 1: We will take the ε-closure for the starting state of NFA as a starting state of DFA.

Step 2: Find the states for each input symbol that can be traversed from the present. That means the union of transition value and their closures for each state of NFA present in the current state of DFA.

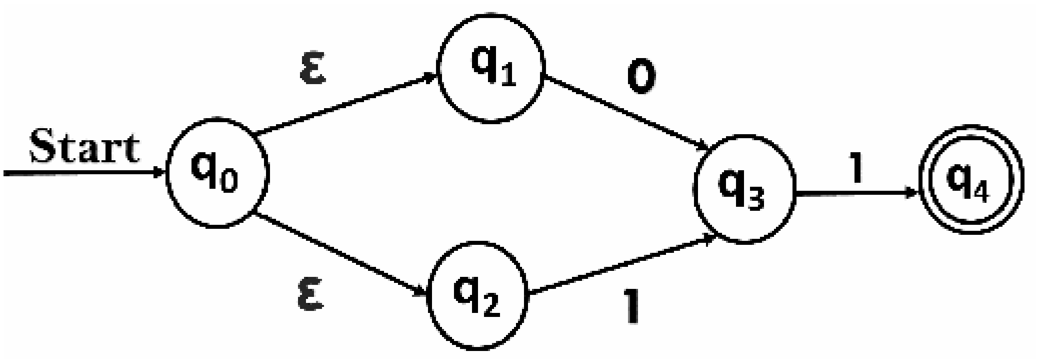
Step 3: If we found a new state, take it as current state and repeat step 2.

Step 4: Repeat Step 2 and Step 3 until there is no new state present in the transition table of DFA.

Step 5: Mark the states of DFA as a final state which contains the final state of NFA.

Example 1:

Convert the NFA with ε into its equivalent DFA.



Solution:

Let us obtain ε-closure of each state.

1. ε-closure {q0} = {q0, q1, q2}
2. ε-closure {q1} = {q1}
3. ε-closure {q2} = {q2}
4. ε-closure {q3} = {q3}
5. ε-closure {q4} = {q4}

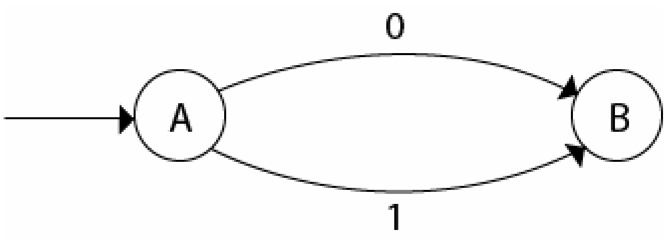
Now, let ε-closure {q0} = {q0, q1, q2} be state A.

Hence

δ'(A, 0) = ε-closure {δ((q0, q1, q2), 0) }
= ε-closure {δ(q0, 0) δ(q1, 0) δ(q2, 0) }
= ε-closure {q3}
= {q3} ***call it as state B****.*

δ'(A, 1) = ε-closure {δ((q0, q1, q2), 1) }
= ε-closure {δ((q0, 1) δ(q1, 1) δ(q2, 1) }
= ε-closure {q3}
= {q3} = B.

The partial DFA will be



Now,

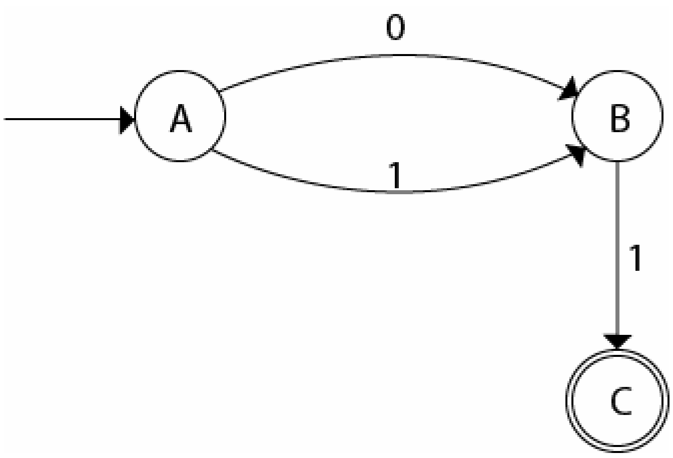
δ'(B, 0) = ε-closure {δ(q3, 0) }
= ϕ
δ'(B, 1) = ε-closure {δ(q3, 1) }
= ε-closure {q4}

= {q4} ***i.e. state C***

For state C:

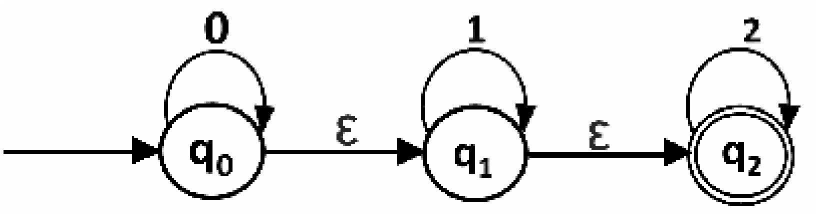
1. δ'(C, 0) = ε-closure {δ(q4, 0) }
2. = ϕ
3. δ'(C, 1) = ε-closure {δ(q4, 1) }
4. = ϕ

The DFA will be,



Example 2:

Convert the given NFA into its equivalent DFA.



Solution: Let us obtain the ε-closure of each state.

1. ε-closure(q0) = {q0, q1, q2}
2. ε-closure(q1) = {q1, q2}
3. ε-closure(q2) = {q2}

Now we will obtain δ' transition. Let ε-closure(q0) = {q0, q1, q2} call it as **state A**.

δ'(A, 0) = ε-closure{δ((q0, q1, q2), 0)}
= ε-closure{δ(q0, 0)   δ(q1, 0)   δ(q2, 0)}
= ε-closure{q0}
= {q0, q1, q2}

δ'(A, 1) = ε-closure{δ((q0, q1, q2), 1)}
= ε-closure{δ(q0, 1)   δ(q1, 1)   δ(q2, 1)}
= ε-closure{q1}
= {q1, q2} ***call it as state B***

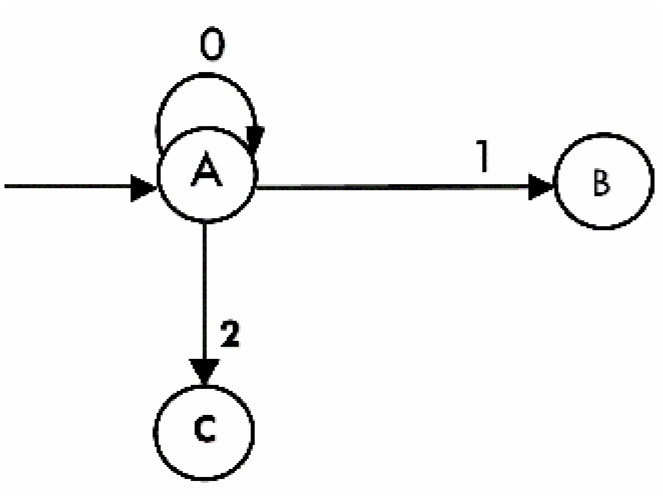
δ'(A, 2) = ε-closure{δ((q0, q1, q2), 2)}

= ε-closure{δ(q0, 2)   δ(q1, 2)   δ(q2, 2)}
= ε-closure{q2}
= {q2} ***call it state C***

Thus we have obtained

1. δ'(A, 0) = A
2. δ'(A, 1) = B
3. δ'(A, 2) = C

The partial DFA will be:



Now we will find the transitions on states B and C for each input.

Hence

δ'(B, 0) = ε-closure{δ((q1, q2), 0)}
= ε-closure{δ(q1, 0)   δ(q2, 0)}
= ε-closure{ϕ}
= ϕ

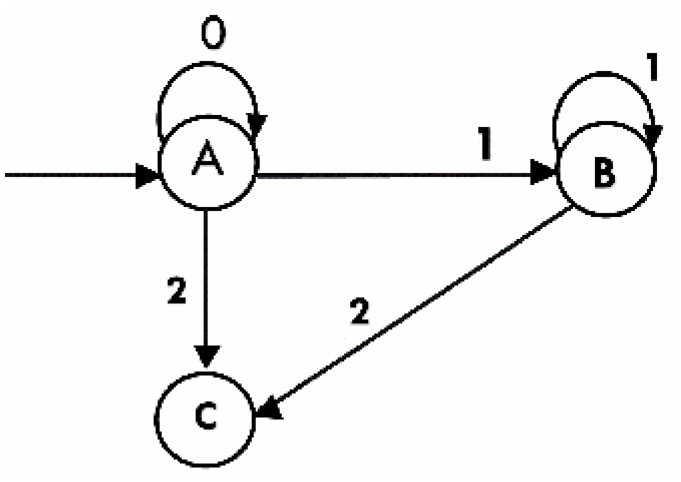
δ'(B, 1) = ε-closure{δ((q1, q2), 1)}
= ε-closure{δ(q1, 1)   δ(q2, 1)}
= ε-closure{q1}
= {q1, q2} ***i.e. state B itself***

δ'(B, 2) = ε-closure{δ((q1, q2), 2)}
= ε-closure{δ(q1, 2)   δ(q2, 2)}
= ε-closure{q2}
= {q2} ***i.e. state C itself***

Thus we have obtained

1. δ'(B, 0) = ϕ
2. δ'(B, 1) = B
3. δ'(B, 2) = C

The partial transition diagram will be



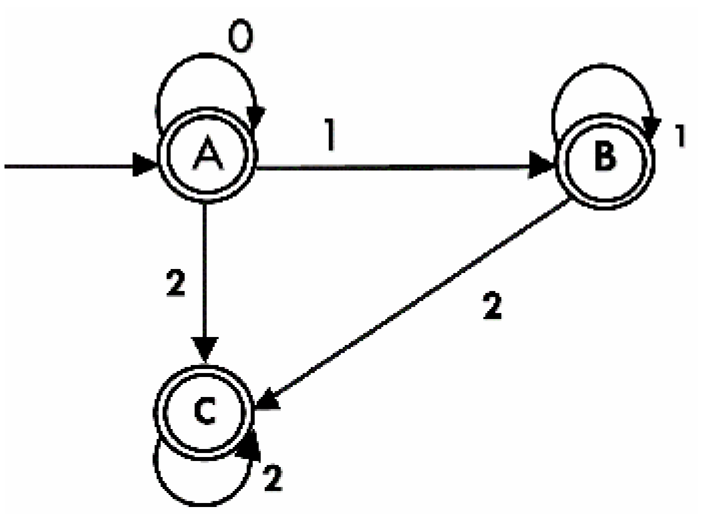
Now we will obtain transitions for C:

δ'(C, 0) = ε-closure{δ(q2, 0)}
= ε-closure{ϕ}
= ϕ

δ'(C, 1) = ε-closure{δ(q2, 1)}
= ε-closure{ϕ}
= ϕ

δ'(C, 2) = ε-closure{δ(q2, 2)}
= {q2}

Hence the DFA is



As A = {q0, q1, q2} in which final state q2 lies hence A is final state. B = {q1, q2} in which the state q2 lies hence B is also final state. C = {q2}, the state q2 lies hence C is also a final state.

UNIT 2

Regular expression

- Regular expression is a sequence of pattern that defines a string. It is used to denote regular languages.
- It is also used to match character combinations in strings. String searching algorithm used this pattern to find the operations on string.
- In regular expression, x* means zero or more occurrence of x. It can generate {e, x, xx, xxx, xxxx,. }
- In regular expression, x+ means one or more occurrence of x. It can generate {x, xx, xxx, xxxx,. }

Operations on Regular Language

The various operations on regular language are:

Union: If L and M are two regular languages then their union L U M is also a union.

1. L U M = {s | s is in L or s is in M}

Intersection: If L and M are two regular languages then their intersection is also an intersection.

1. L ∩ M = {st | s is in L and t is in M}

Kleene closure: If L is a regular language then its kleene closure L1* will also be a regular language.

1. L* = Zero or more occurrence of language L.

Example

Write the regular expression for the language: L = {abn

w:n ≥ 3, w ∈ (a,b)+}

Solution:

The string of language L starts with "a" followed by atleast three b's. Itcontains atleast one "a" or one "b" that is string are like abbba, abbbbbba, abbbbbbbb, abbbb. a

So regular expression is:

r= ab3b* (a+b)+

Here + is a positive closure i.e. (a+b)+ = (a+b)* - ∈

Optimization of DFA

To optimize the DFA you have to follow the various steps. These are as follows:

Step 1: Remove all the states that are unreachable from the initial state via any set of the transition of DFA.

Step 2: Draw the transition table for all pair of states.

Step 3: Now split the transition table into two tables T1 and T2. T1 contains all final states and T2 contains non-final states.

Step 4: Find the similar rows from T1 such that:

1. δ (q, a) = p
2. δ (r, a) = p

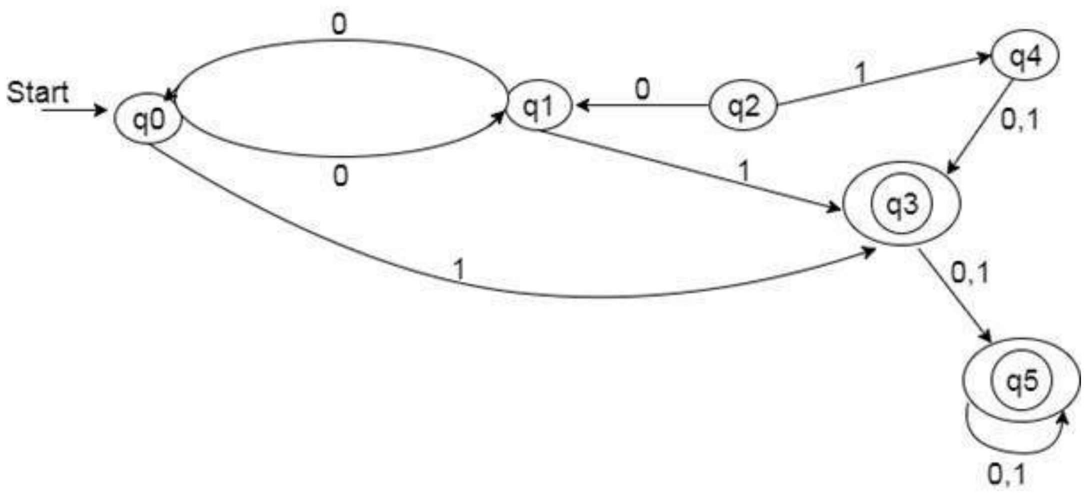
That means, find the two states which have same value of a and b and remove one of them.

Step 5: Repeat step 3 until there is no similar rows are available in the transition table T1.

Step 6: Repeat step 3 and step 4 for table T2 also.

Step 7: Now combine the reduced T1 and T2 tables. The combined transition table is the transition table of minimized DFA.

Example



Solution:

Step 1: In the given DFA, q2 and q4 are the unreachable states so remove them.

Step 2: Draw the transition table for rest of the states.

State	0	1
→q0	q1	q3
q1	q0	q3
*q3	q5	q5
*q5	q5	q5

Step 3:

Now divide rows of transition table into two sets as:

1. One set contains those rows, which start from non-final sates:

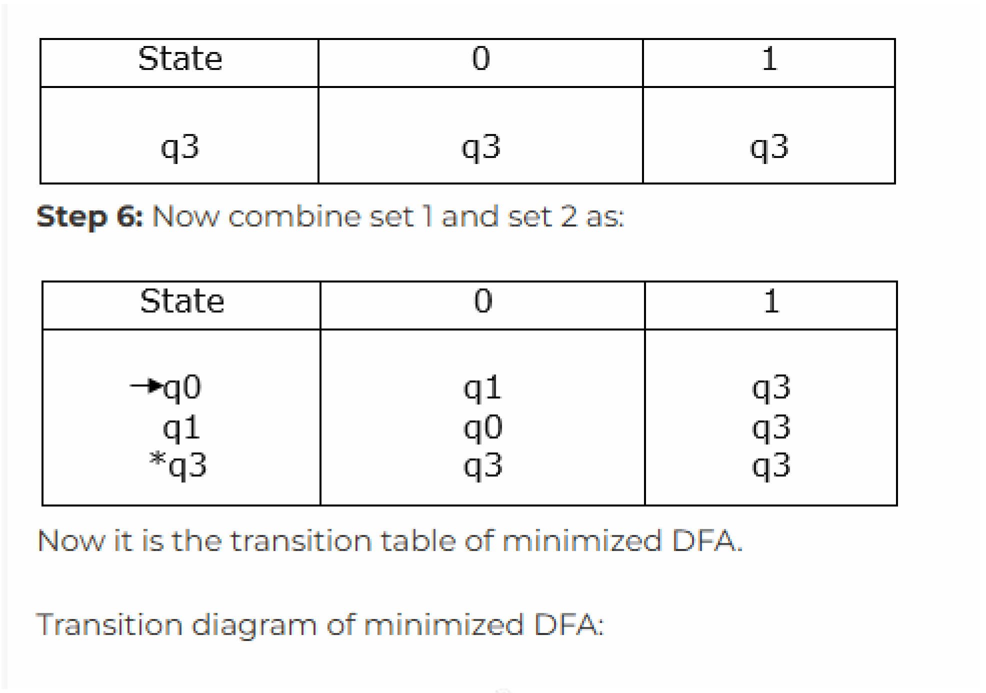
State	0	1
q0	q1	q3
q1	q0	q3

2. Other set contains those rows, which starts from final states.

State	0	1
q3	q5	q5
q5	q5	q5

Step 4: Set 1 has no similar rows so set 1 will be the same.

Step 5: In set 2, row 1 and row 2 are similar since q3 and q5 transit to same state on 0 and 1. So skip q5 and then replace q5 by q3 in the rest.



State	0	1
q3	q3	q3

Step 6: Now combine set 1 and set 2 as:

State	0	1
→q0	q1	q3
q1	q0	q3
*q3	q3	q3

Now it is the transition table of minimized DFA.

Transition diagram of minimized DFA:

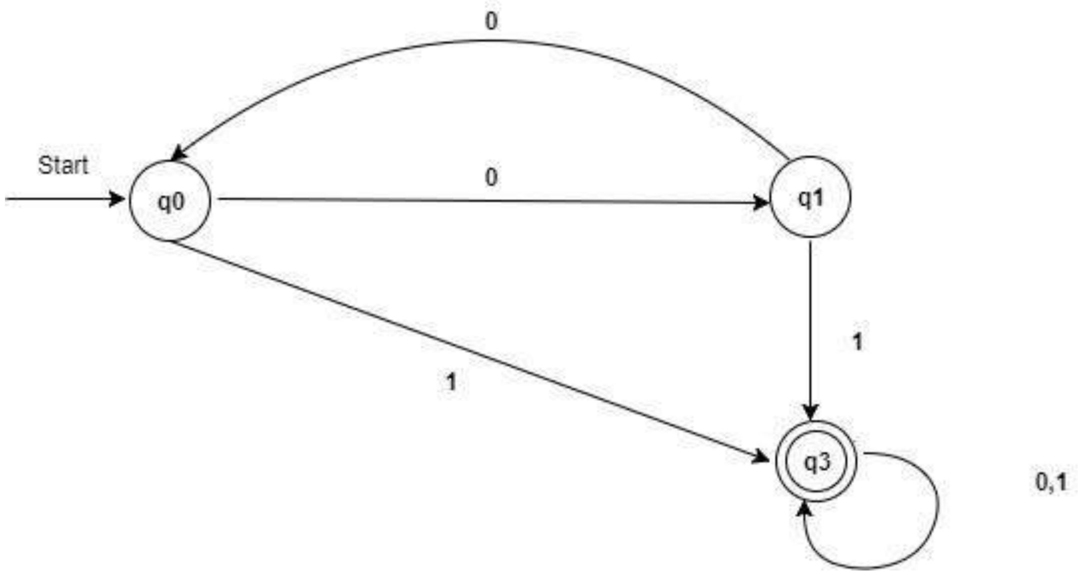


Fig: Minimized DFA

Applications of regular expression

1. Extracting emails from a Text Document

2. Regular Expressions for Web Scraping (Data Collection)

3. Working with Date-Time features

4. Using Regex for Text Pre-processing (NLP)

lgebraic Properties of Regular Expressions

Kleene closure is an unary operator and Union(+) and concatenation operator(.) are binary operators.

1. Closure

If r1 and r2 are regular expressions(RE), then

- r1* is a RE
- r1+r2 is a RE
- r1.r2 is a RE

2. Closure Laws

- (r*)* = r*, closing an expression that is already closed does not change the language.
- ∅ * = ∈ , a string formed by concatenating any number of copies of an empty string is empty itself.
- r + = r.r* = r*r, as r* = ∈ + r + rr+ rrr …. and r.r* = r+ rr + rrr ……
- r* = r*+ ∈

3. Associativity

If r1, r2, r3 are RE, then

i.) r1+ (r2+r3) = (r1+r2) +r3

- **For example** : r1 = a , r2 = b , r3 = c, then
- The resultant regular expression in LHS becomes a+(b+ c) and the regular set for the corresponding RE is {a, b, c}.
- for the RE in RHS becomes (a+ b) + c and the regular set for this RE is {a, b, c}, which is same in both cases. Therefore, the associativity property holds for union operator.

ii.) r1.(r2.r3) = (r1.r2).r3

- **For example –** r1 = a , r2 = b , r3 = c
- Then the string accepted by RE a.(b.c) is only abc.

- The string accepted by RE in RHS is (a.b).c is only abc ,which is same in both cases. Therefore, the **associativity property holds for concatenation operator.**

Associativity property does not hold for Kleene closure(*) because it is unary operator.

4. Identity

In the case of union operators,

r + ∅ = ∅ + r = r,

Therefore, ∅ is the identity element for a union operator. In the case of concatenation operator:

r.x = r , for x= ∈ , r.∈ = r

Therefore, ∈ is the identity element for concatenation operator(.).

5. Annihilator

- If r+ x = r ⇒ r ∪ x= x , there is no annihilator for +
- In the case of a concatenation operator, r.x = x, when x = ∅ , then r.∅ = ∅ , therefore ∅ is the annihilator for the (.)operator. For example {a, aa, ab}.{ } = { }

6. Commutative Property

If r1, r2 are RE, then

- r1+r2 = r2+r1. For example, for r1 =a and r2 =b, then RE a+ b and b+ a are equal.
- r1.r2 ≠ r2.r1. For example, for r1 = a and r2 = b, then RE a.b is not equal to b.a.

7. Distributed Property

If r1, r2, r3 are regular expressions, then

- (r1+r2).r3 = r1.r3 + r2.r3 i.e. Right distribution
- r1.(r2+ r3) = r1.r2 + r1.r3 i.e. left distribution
- (r1.r2) +r3 ≠ (r1+r3)(r2+r3)

8. **Idempotent Law**

- r1 + r1 = r1 ⇒ r1 ∪ r1 = r1 , therefore the union operator satisfies idempotent property.
- r.r ≠ r ⇒concatenation operator does not satisfy idempotent property.

9. **Identities for Regular Expression**

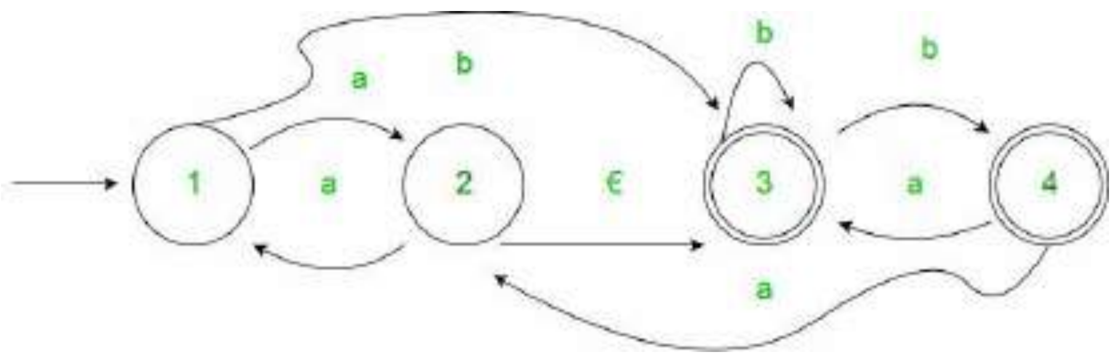
There are many identities for the regular expression. Let p, q and r are regular expressions.

- ∅ + r = r
- ∅ .r= r.∅ = ∅
- ∈ .r = r.∈ =r
- ∈* = ∈ and ∅* = ∈
- r + r = r
- r*.r* = r*
- r.r* = r*.r = r + .
- (r*)* = r*
- ∈ +r.r* = r* = ∈ + r.r*
- (p.q)*.p = p.(q.p)*
- (p + q)* = (p*.q*)* = (p* + q*)*
- (p+ q).r= p.r+ q.r and r.(p+q) = r.p + r.q

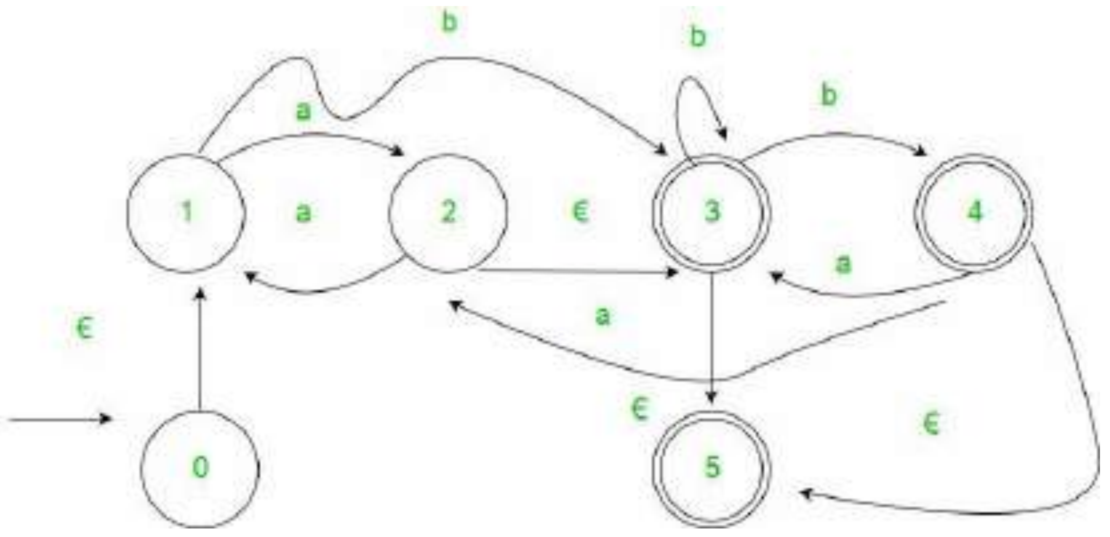
1. **State Elimination Method:**

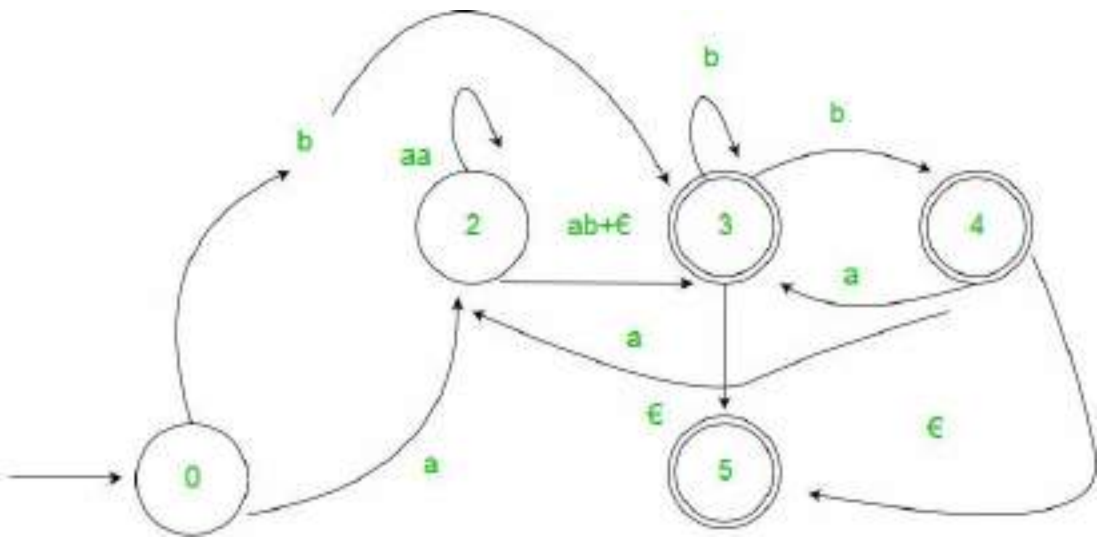
- **Step 1 –** If the start state is an accepting state or has transitions in, add a new non-accepting start state and add an €-transition between the new start state and the former start state.
- **Step 2 –** If there is more than one accepting state or if the single accepting state has transitions out, add a new accepting state, make all other states non-accepting, and add an €-transition from each former accepting state to the new accepting state.
- **Step 3 –** For each non-start non-accepting state in turn, eliminate the state and update transitions accordingly.

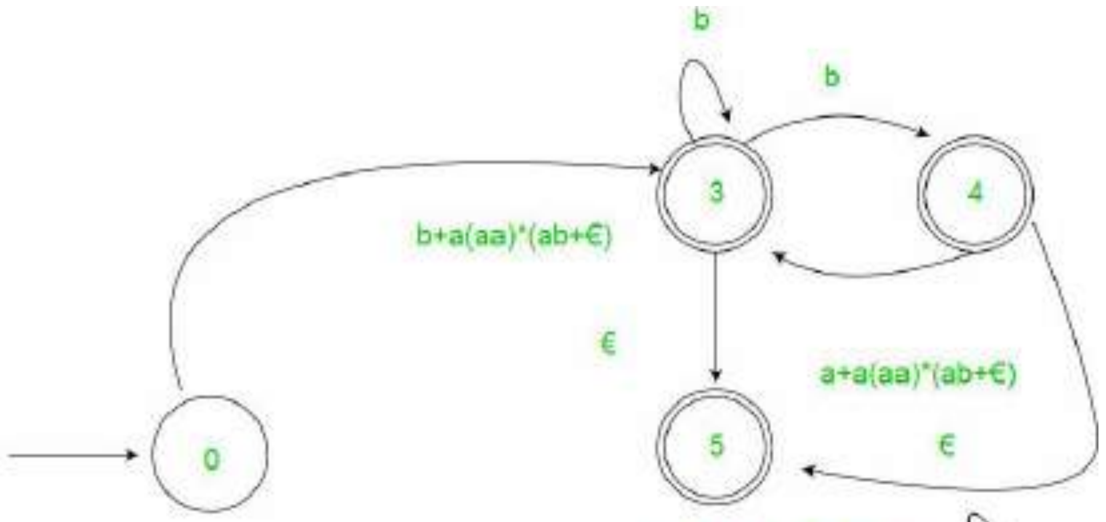
Example :-

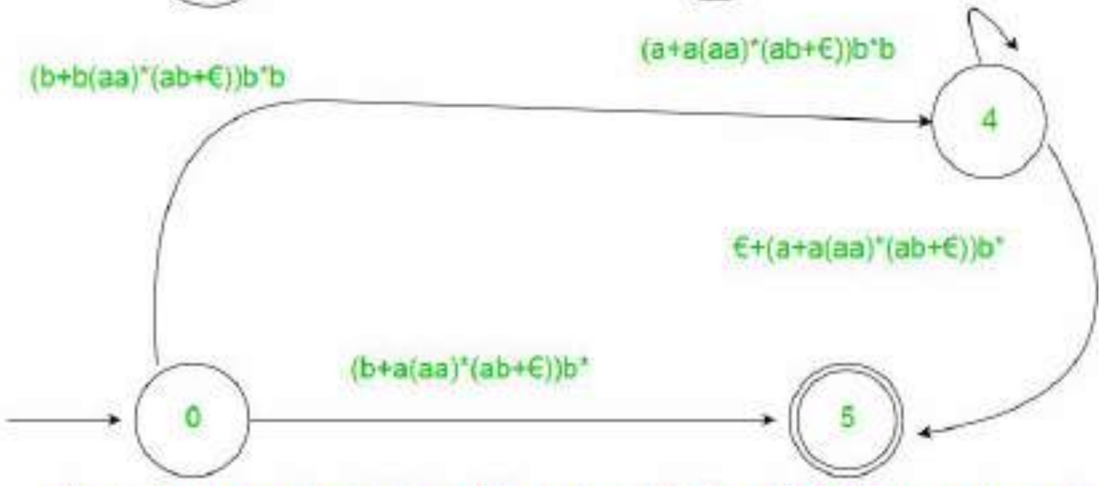


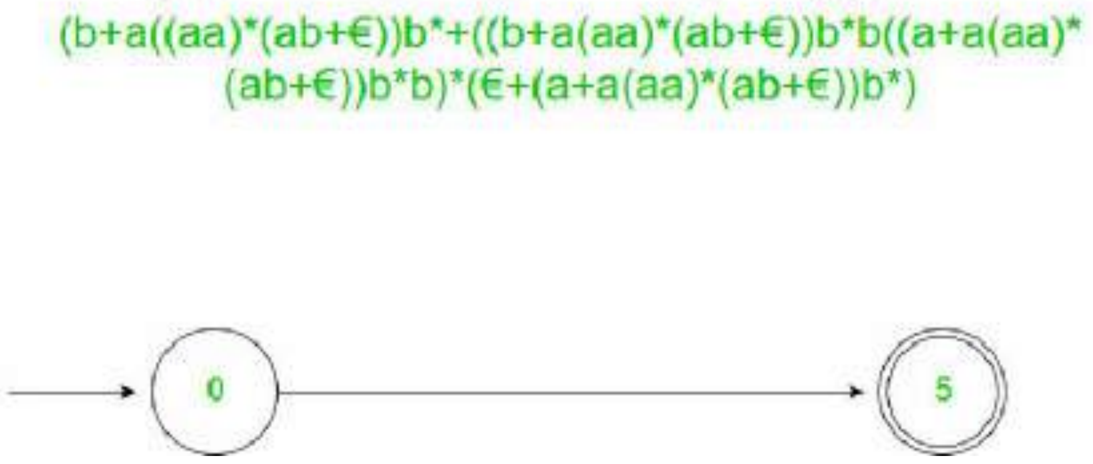
Solution:










(b+a((aa)*(ab+€))b*+((b+a(aa)*(ab+€))b*b((a+a(aa)*(ab+€))b*b)*(€+(a+a(aa)*(ab+€))b*)
0
5

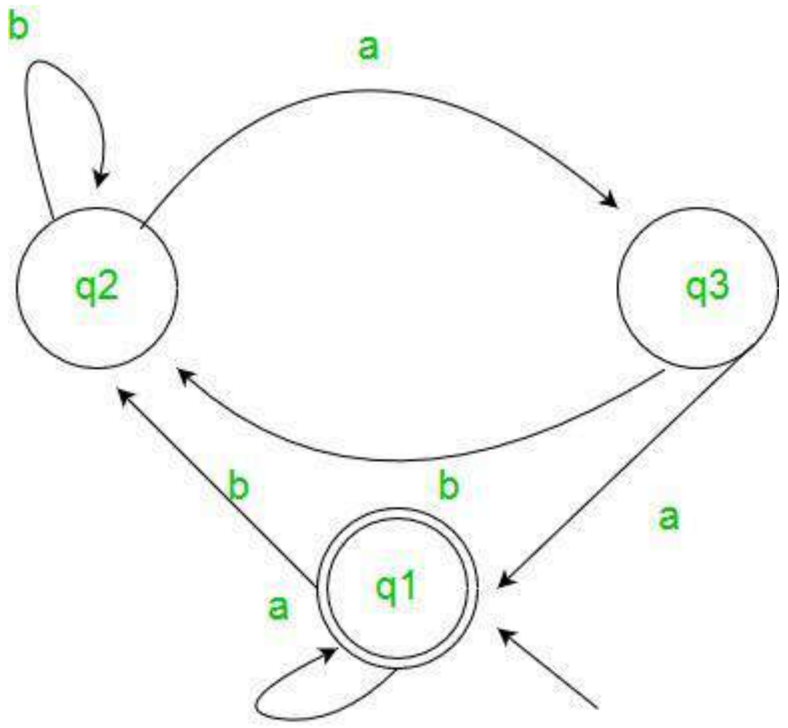
2. **Arden's Theorem:**

Let P and Q be 2 regular expressions. If P does not contain null string, then the following equation in R, viz R = Q + RP, Has a unique solution by R = QP* **Assumptions –**

- The transition diagram should not have €-moves.
- It must have only one initial state.

Using Arden's Theorem to find Regular Expression of Deterministic Finite automata –

1. For getting the regular expression for the automata we first create equations of the given form for all the states **q1 = q1w11 +q2w21 +…+qnwn1 +€ (q1 is the initial state) q2 = q1w12 +q2w22 +…+qnwn2 . . . qn = q1w1n +q2w2n +…+qnwnn** wij is the regular expression representing the set of labels of edges from qi to qj **Note** – For parallel edges there will be that many expressions for that state in the expression.
2. Then we solve these equations to get the equation for qi in terms of wij and that expression is the required solution, where qi is a final state.



Example :-
Solution :- Here the initial state is q1 and the final state is q1. The equations for the three states q1, q2, and q3 are as follows ? q1 = q1a + q3a + € (€ move is because q1 is the initial state) q2 = q1b + q2b + q3b q3 = q2a Now, we will solve these three equations ? q2 = q1b + q2b + q3b = q1b + q2b + (q2a)b (Substituting value of q3) = q1b + q2(b + ab)
= q1b (b + ab)* (Applying Arden's Theorem) q1 = q1a + q3a + € = q1a
+ q2aa + € (Substituting value of q3) = q1a + q1b(b + ab*)aa + € (Substituting value of q2) = q1(a + b(b + ab)*aa) + € = € (a+ b(b + ab)*aa)* = (a + b(b + ab)*aa)* Hence, the regular expression is (a + b(b
+ ab)*aa)*.

Pumping lemma for Regular languages

- It gives a method for pumping (generating) many substrings from a given string.
- In other words, we say it provides means to break a given long input string into several substrings.
- Lt gives necessary condition(s) to prove a set of strings is not regular.

Theorem

For any regular language L, there exists an integer P, such that for all w in L

|w|>=P

We can break w into three strings, w=xyz such that.

(1) lxyl < P

(2) lyl > 1

(3) for all k>= 0: the string xy^kz is also in L

Application of pumping lemma

Pumping lemma is to be applied to show that certain languages are not regular.

It should never be used to show a language is regular.

- If L is regular, it satisfies the Pumping lemma.
- If L does not satisfy the Pumping Lemma, it is not regular.

Steps to prove that a language is not regular by using PLare as follows−

- step 1 − We have to assume that L is regular
- step 2 − So, the pumping lemma should hold for L.
- step 3 − It has to have a pumping length (say P).
- step 4 − All strings longer that P can be pumped |w|>=p.
- step 5 − Now find a string 'w' in L such that |w|>=P
- step 6 − Divide w into xyz.

- step 7 − Show that $xy^iz \notin L$ for some i.
- step 8 − Then consider all ways that w can be divided into xyz.
- step 9 − Show that none of these can satisfy all the 3 pumping conditions at same time.
- step 10 − w cannot be pumped = CONTRADICTION.

Context free grammar

Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language.

Context free grammar G can be defined by four tuples as:

1. G= (V, T, P, S)

Where,

G describes the grammar

T describes a finite set of terminal symbols.

V describes a finite set of non-terminal symbols

P describes a set of production rules

S is the start symbol.

In CFG, the start symbol is used to derive the string. You can derive the string by repeatedly replacing a non-terminal by the right hand side of the production, until all non-terminal have been replaced by terminal symbols.

Example:

L= {wcwR | w € (a, b)*}

Production rules:

1. S → aSa
2. S → bSb
3. S → c

Now check that abbcbba string can be derived from the given CFG.

1. S ⇒ aSa
2. S ⇒ abSba
3. S ⇒ abbSbba
4. S ⇒ abbcbba

By applying the production S → aSa, S → bSb recursively and finally applying the production S → c, we get the string abbcbba.

Derivation

Derivation is a sequence of production rules. It is used to get the input string through these production rules. During parsing we have to take two decisions. These are as follows:

- We have to decide the non-terminal which is to be replaced.
- We have to decide the production rule by which the non-terminal will be replaced.

We have two options to decide which non-terminal to be replaced with production rule.

Left-most Derivation

In the left most derivation, the input is scanned and replaced with the production rule from left to right. So in left most derivatives we read the input string from left to right.

Example:

Production rules:

1. S = S + S
2. S = S - S
3. S = a | b |c

Input:

a - *b* + *c*

The left-most derivation is:

1. S = S + S
2. S = S - S + S
3. S = a - S + S

4. S = a - b + S
5. S = a - b + c

Right-most Derivation

In the right most derivation, the input is scanned and replaced with the production rule from right to left. So in right most derivatives we read the input string from right to left.

Example:

1. S = S + S
2. S = S - S
3. S = a | b |c

   Input:

   a - *b* + *c*

The right-most derivation is:

1. S = S - S
2. S = S - S + S
3. S = S - S + c
4. S = S - b + c
5. S = a - b + c

Ambiguity in Grammar

A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivation or more than one parse tree for the given input string. If the grammar is not ambiguous, then it is called unambiguous.

If the grammar has ambiguity, then it is not good for compiler construction. No method can automatically detect and remove the ambiguity, but we can remove ambiguity by re-writing the whole grammar without ambiguity.

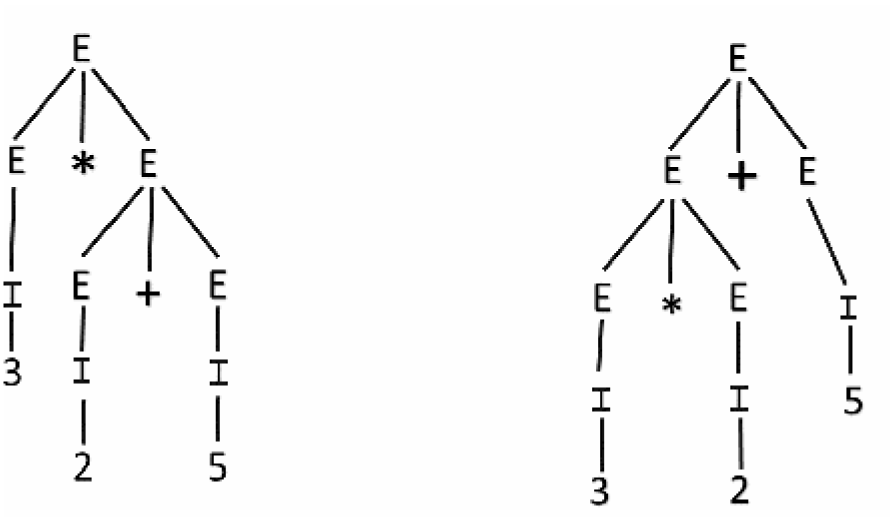
Example 1:

Let us consider a grammar G with the production rule

1. E → I
2. E → E + E
3. E → E * E
4. E → (E)
5. I → ε | 0 | 1 | 2 | ... | 9

Solution:

For the string "3 * 2 + 5", the above grammar can generate two parse trees by leftmost derivation:



Since there are two parse trees for a single string "3 * 2 + 5", the grammar G is ambiguous.

Example 2:

Check whether the given grammar G is ambiguous or not.

1. E → E + E
2. E → E - E
3. E → id

Solution:

From the above grammar String "id + id - id" can be derived in 2 ways:

First Leftmost derivation

1. E → E + E
2. → id + E
3. → id + E - E

4. → id + id - E
5. → id + id- id

Second Leftmost derivation

1. E → E - E
2. → E + E - E
3. → id + E - E
4. → id + id - E
5. → id + id - id

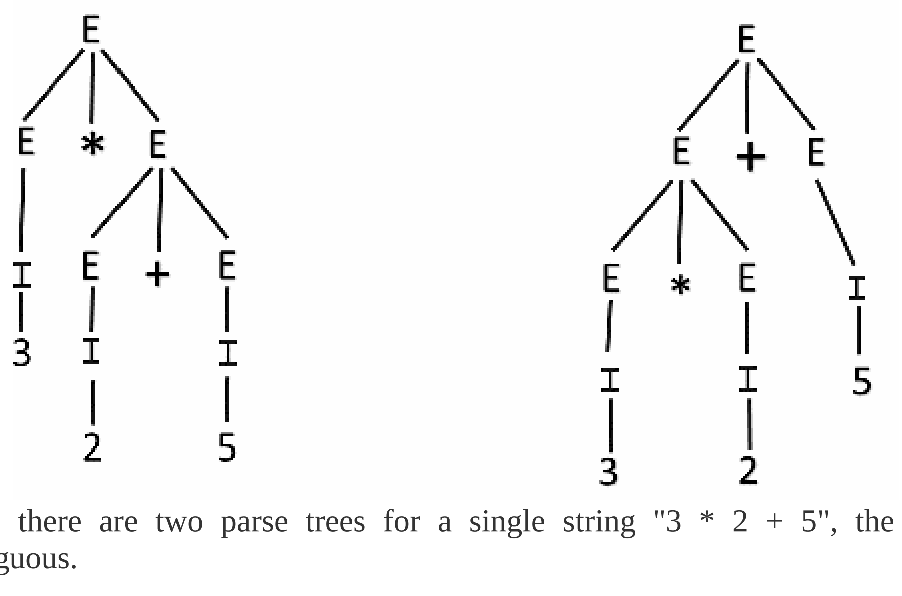
Since there are two leftmost derivation for a single string "id + id - id", the grammar G is ambiguous.

Example 3:

Check whether the given grammar G is ambiguous or not.

1. S → aSb | SS
2. S → ε

For the string "3 * 2 + 5", the above grammar can generate two parse trees by leftmost derivation:



Since there are two parse trees for a single string "3 * 2 + 5", the grammar G is ambiguous.

Example 2:

Check whether the given grammar G is ambiguous or not.

1. $E \rightarrow E + E$
2. $E \rightarrow E - E$
3. $E \rightarrow id$

Solution:

From the above grammar String "id + id - id" can be derived in 2 ways:

First Leftmost derivation

1. $E \rightarrow E + E$
2. $\rightarrow id + E$
3. $\rightarrow id + E - E$
4. $\rightarrow id + id - E$
5. $\rightarrow id + id- id$

Second Leftmost derivation

1. $E \rightarrow E - E$
2. $\rightarrow E + E - E$
3. $\rightarrow id + E - E$
4. $\rightarrow id + id - E$
5. $\rightarrow id + id - id$

Since there are two leftmost derivation for a single string "id + id - id", the grammar G is ambiguous.

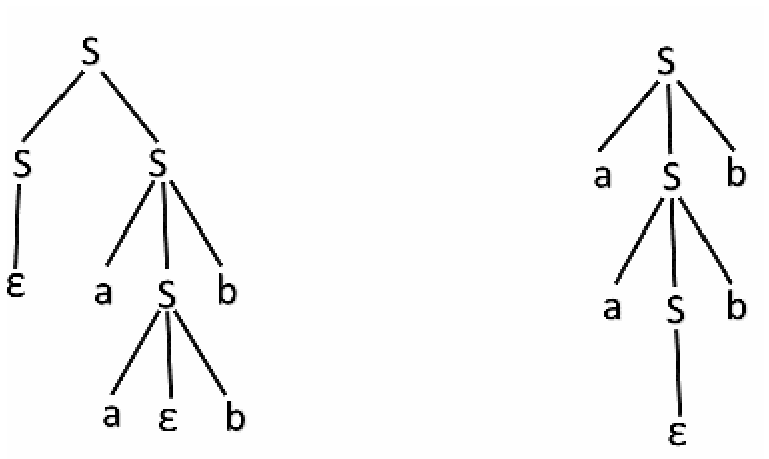
Example 3:

Check whether the given grammar G is ambiguous or not.

1. $S \rightarrow aSb \mid SS$
2. $S \rightarrow \varepsilon$

Solution:

For the string "aabb" the above grammar can generate two parse trees



Since there are two parse trees for a single string "aabb", the grammar G is ambiguous.

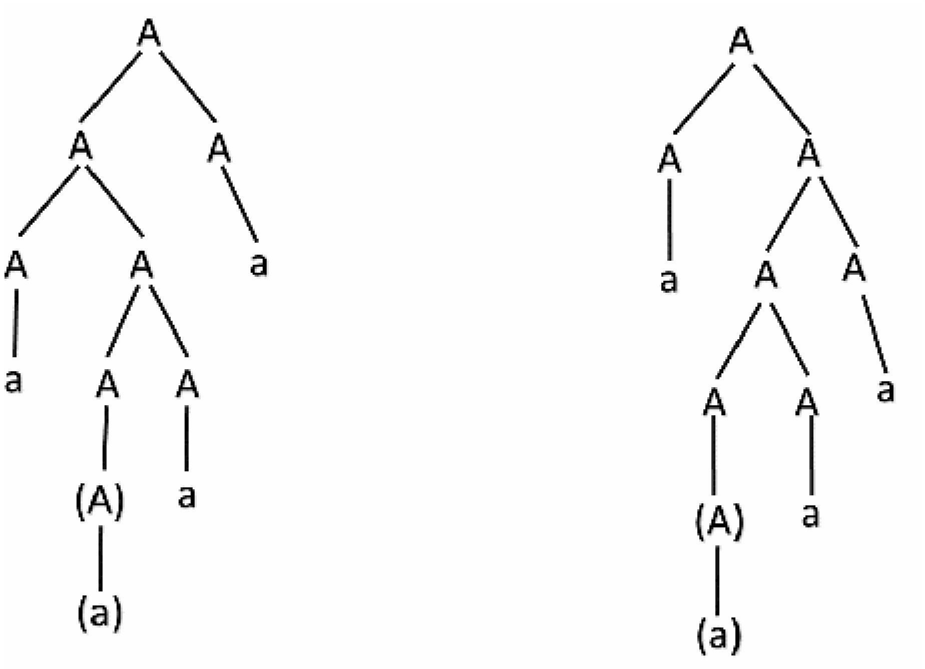
Example 4:

Check whether the given grammar G is ambiguous or not.

1. A → AA
2. A → (A)
3. A → a

Solution:

For the string "a(a)aa" the above grammar can generate two parse trees:



Since there are two parse trees for a single string "a(a)aa", the grammar G is ambiguous.

Unambiguous Grammar

A grammar can be unambiguous if the grammar does not contain ambiguity that means if it does not contain more than one leftmost derivation or more than one rightmost derivation or more than one parse tree for the given input string.

To convert ambiguous grammar to unambiguous grammar, we will apply the following rules:

1. If the left associative operators (+, -, *, /) are used in the production rule, then apply left recursion in the production rule. Left recursion means that the leftmost symbol on the right side is the same as the non-terminal on the left side. For example,

1. $X \rightarrow Xa$

2. If the right associative operates(^) is used in the production rule then apply right recursion in the production rule. Right recursion means that the rightmost symbol on the left side is the same as the non-terminal on the right side. For example,

1. $X \rightarrow aX$

Example 1:

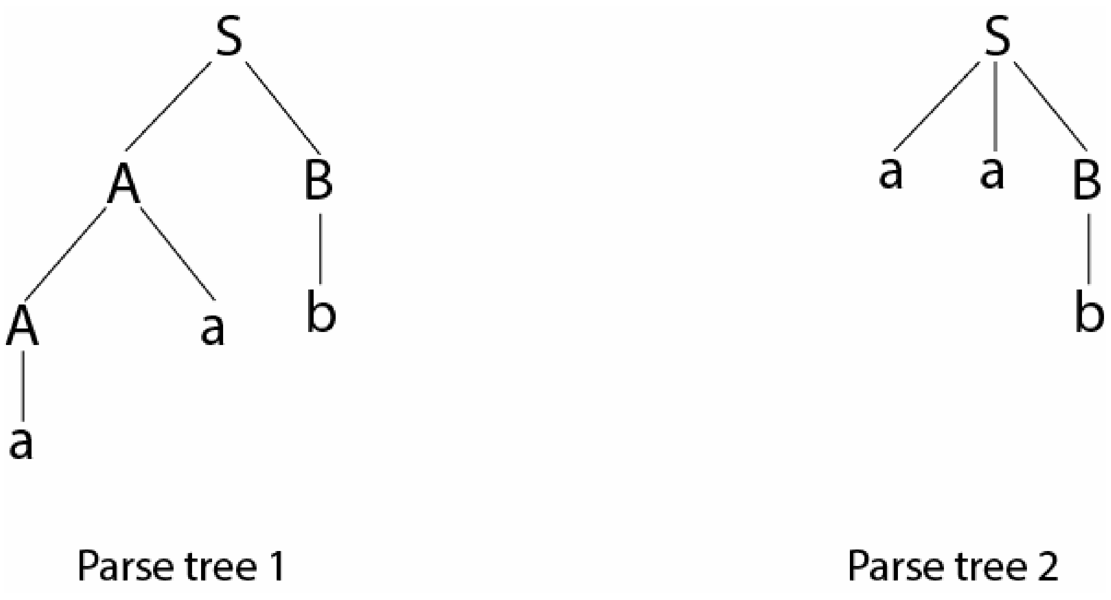
Consider a grammar G is given as follows:

1. $S \rightarrow AB \mid aaB$
2. $A \rightarrow a \mid Aa$
3. $B \rightarrow b$

Determine whether the grammar G is ambiguous or not. If G is ambiguous, construct an unambiguous grammar equivalent to G.

Solution:

Let us derive the string "aab"



1. As there are two differS → AB
2. A → Aa | a
3. B → b

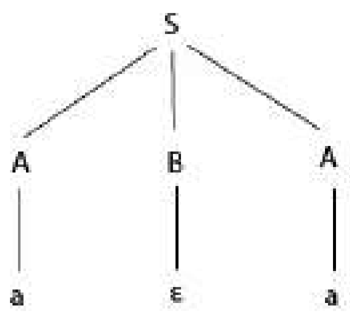
Example 2:

Show that the given grammar is ambiguous. Also, find an equivalent unambiguous grammar.

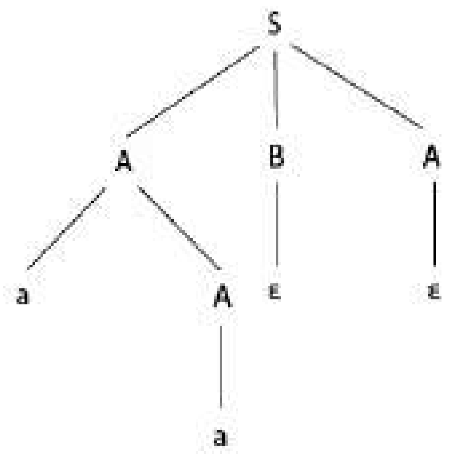
1. S → ABA
2. A → aA | ε
3. B → bB | ε

Solution:

The given grammar is ambiguous because we can derive two different parse tree for string aa.



Parse tree 1



Parse tree 2

The unambiguous grammar is:

1. S → aXY | bYZ | ε
2. Z → aZ | a
3. X → aXY | a | ε
4. Y → bYZ | b | ε

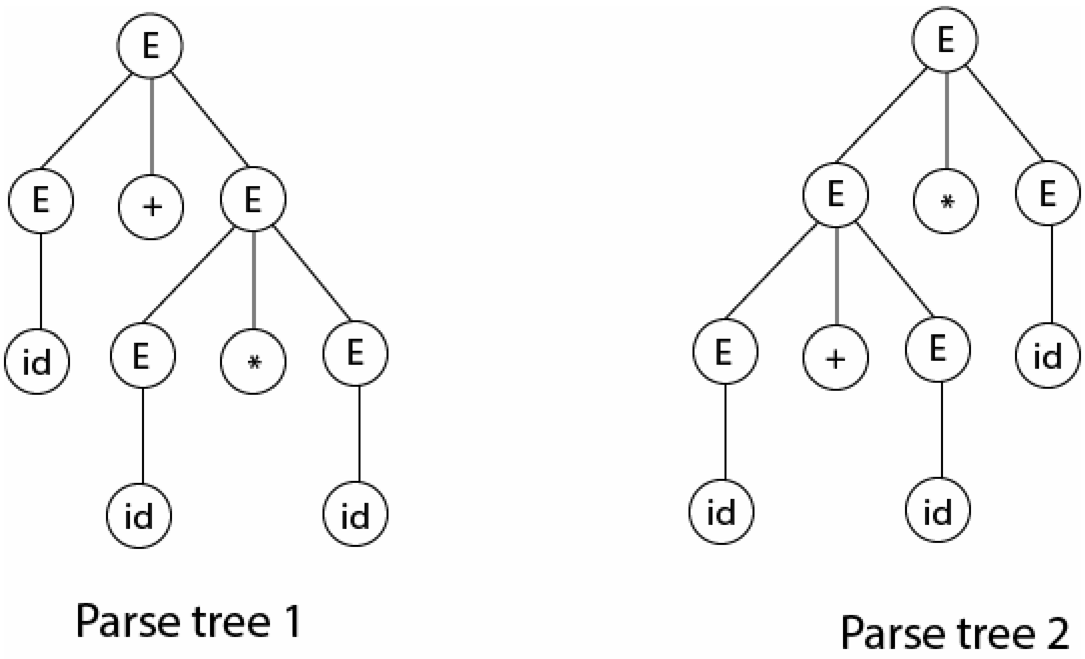
Example 3:

Show that the given grammar is ambiguous. Also, find an equivalent unambiguous grammar.

1. E → E + E
2. E → E * E
3. E → id

Solution:

Let us derive the string "id + id * id"



As there are two different parse tree for deriving the same string, the given grammar is ambiguous.

Unambiguous grammar will be:

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → id

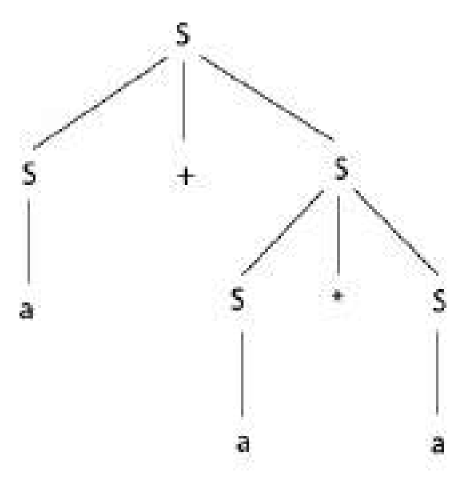
Example 4:

Check that the given grammar is ambiguous or not. Also, find an equivalent unambiguous grammar.

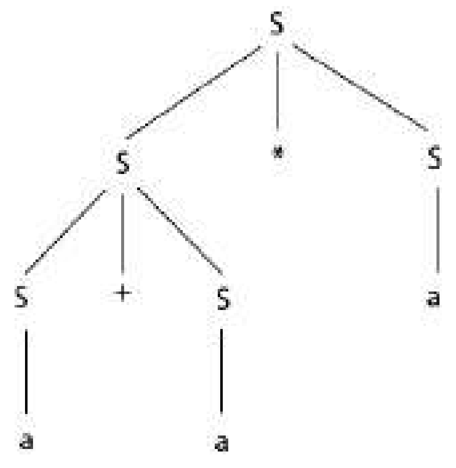
1. S → S + S
2. S → S * S
3. S → S ^ S
4. S → a

Solution:

The given grammar is ambiguous because the derivation of string aab can be represented by the following string:



Parse tree 1



Parse tree 2

Unambiguous grammar will be:

1. S → S + A |
2. A → A * B | B
3. B → C ^ B | C
4. C → a

Simplification of CFG

ent parse tree for deriving the same string, the given grammar is ambiguous.

1. S → AB
2. A → Aa | a
3. B → b

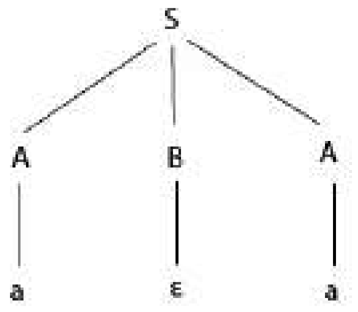
Example 2:

Show that the given grammar is ambiguous. Also, find an equivalent unambiguous grammar.

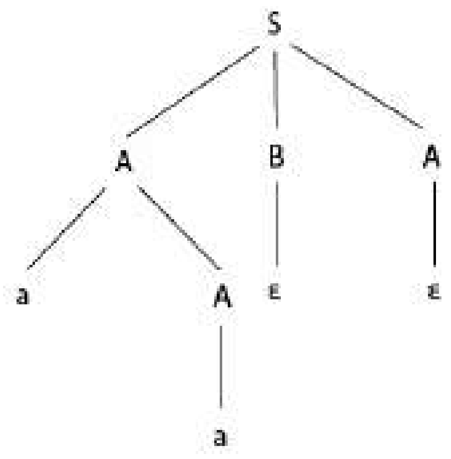
1. S → ABA
2. A → aA | ε
3. B → bB | ε

Solution:

The given grammar is ambiguous because we can derive two different parse tree for string aa.



Parse tree 1



Parse tree 2

The unambiguous grammar is:

1. S → aXY | bYZ | ε
2. Z → aZ | a
3. X → aXY | a | ε
4. Y → bYZ | b | ε

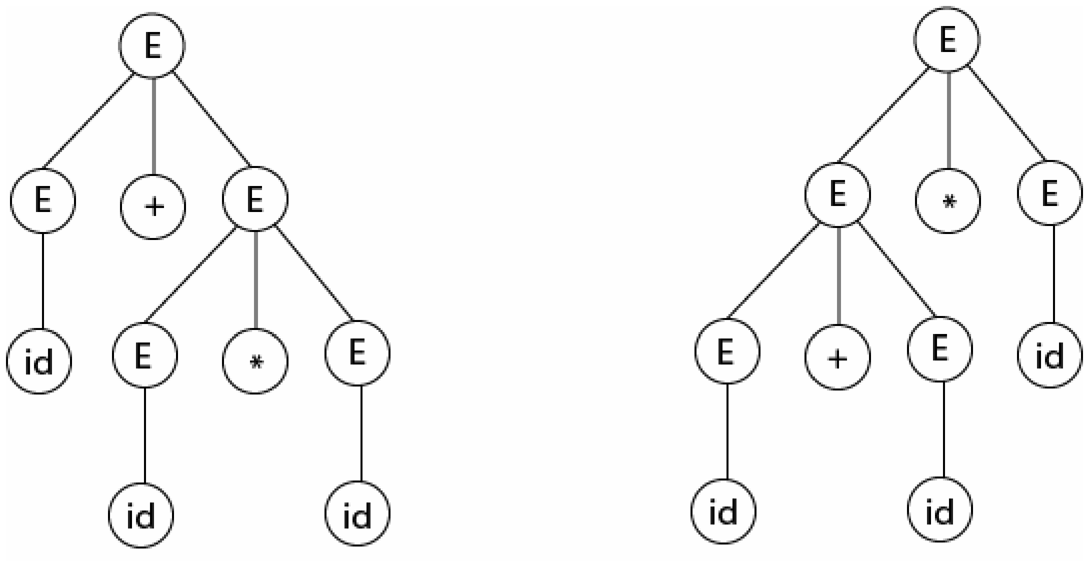
Example 3:

Show that the given grammar is ambiguous. Also, find an equivalent unambiguous grammar.

1. E → E + E
2. E → E * E
3. E → id

Solution:

Let us derive the string "id + id * id"



Parse tree 1

Parse tree 2

As there are two different parse tree for deriving the same string, the given grammar is ambiguous.

Unambiguous grammar will be:

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → id

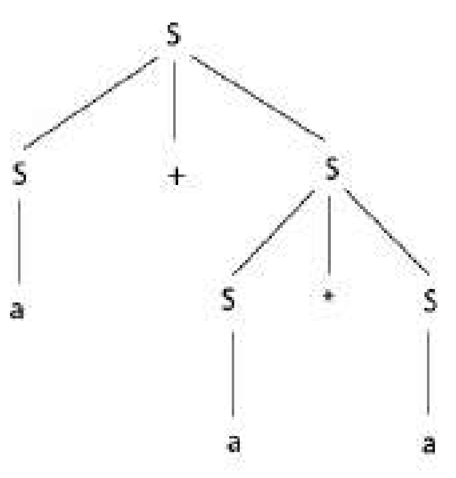
Example 4:

Check that the given grammar is ambiguous or not. Also, find an equivalent unambiguous grammar.

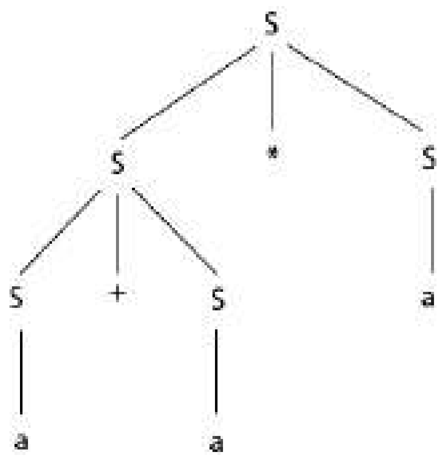
1. S → S + S
2. S → S * S
3. S → S ^ S
4. S → a

Solution:

The given grammar is ambiguous because the derivation of string aab can be represented by the following string:



Parse tree 1



Parse tree 2

Unambiguous grammar will be:

1. S → S + A |
2. A → A * B | B
3. B → C ^ B | C
4. C → a

Parse Trees and Their Role in Parsing

A parse tree, also known as derivation tree is a graphical representation of the derivation of a string according to the production rules of a CFG. It shows how a start symbol of a CFG can be transformed into a terminal string, using applying production rules in a sequence.

Each node in a parse tree represents a symbol of the grammar, while the edges represent the application of production rules.

- **Tree Nodes** − The nodes in a parse tree represent either terminal or non-terminal symbols of the grammar.
- **Tree Edges** − The edges represent the application of production rules leading from one node to another.

Types of Parsers

Parsers can be broadly categorized into two types based on how they construct the parse tree −

- **Top-down Parsers** − These parsers build the parse tree starting from the root and proceed towards the leaves. They typically perform a leftmost derivation. They expand the leftmost non-terminal at each step.
- **Bottom-up Parsers** − These parsers start from the leaves and move towards the root, performing a rightmost derivation in reverse order.

Example of Parse Tree Construction

Let us consider an example of a parse tree construction using a simple grammar −

Given grammar GG −

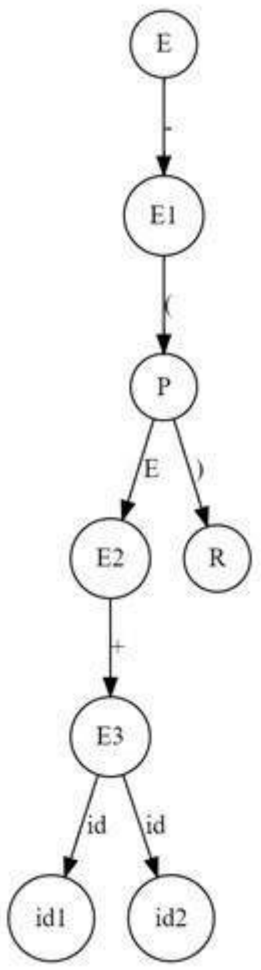
E→E+E|E×E|(E)||−E|idE→E+E|E×E|(E)||−E|id

Deriving the String "-(id + id)"

To determine whether the string **"-(id + id)"** is a valid sentence in this grammar, we can construct a parse tree as follows −

- Start with EE.
- Apply the production E→−EEE→−EE to get the intermediate string "\mathrm{- E}".
- Apply E→(E)E→(E) to derive "−(E)−(E)".
- Apply E→E+EE→E+E to derive "−(E+E)−(E+E)".
- Finally, apply E→idE→id to get the final string "−(id+id)−(id+id)".

Here is the corresponding **parse tree** −



This representation shows the hierarchical structure of the derivation, with the root representing the start symbol and the leaves representing the terminal symbols of the string.

Another Example of Parse Tree

Consider a different grammar for a second example − Given grammar

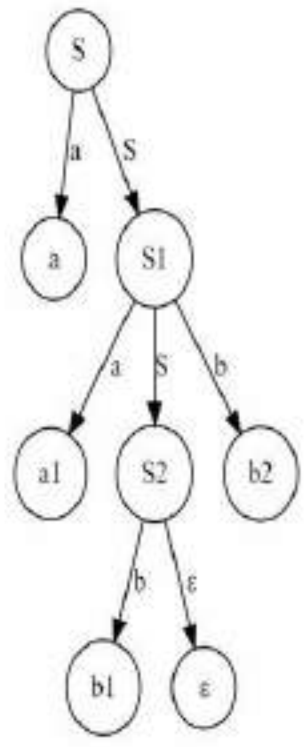
GG

S→SS|aSb|εS→SS|aSb|ε

To derive the string **"aabb"**, we proceed as follows −

- Start with SS.
- Apply S→aSbS→aSb to get "aSbaSb".
- Apply S→asbS→asb again within the non-terminal SS to derive "aaSbbaaSbb".
- Finally, apply S→εS→ε to replace the last SS, resulting in "aabbaabb".

The **corresponding parse tree** can be represented like below −



UNIT-3

Pushdown Automata is a finite automata with extra memory called stack which helps Pushdown automata to recognize Context Free Languages. This article describes pushdown automata in detail.

Pushdown Automata
A Pushdown Automata (PDA) can be defined as :

- Q is the set of states
- ∑is the set of input symbols
- Γ is the set of pushdown symbols (which can be pushed and popped from the stack)
- q0 is the initial state
- Z is the initial pushdown symbol (which is initially present in the stack)
- F is the set of final states
- δ is a transition function that maps Q x {Σ ∪ ∈} x Γ into Q x Γ*. In a given state, the PDA will read the input symbol and stack symbol (top of the stack) move to a new state, and change the symbol of the stack.

Instantaneous Description (ID)
Instantaneous Description (ID) is an informal notation of how a PDA "computes" an input string and makes a decision whether that string is accepted or rejected.

An ID is a triple (q, w, α), where:
1. q is the current state.
2. w is the remaining input.
3.α is the stack contents, top at the left.

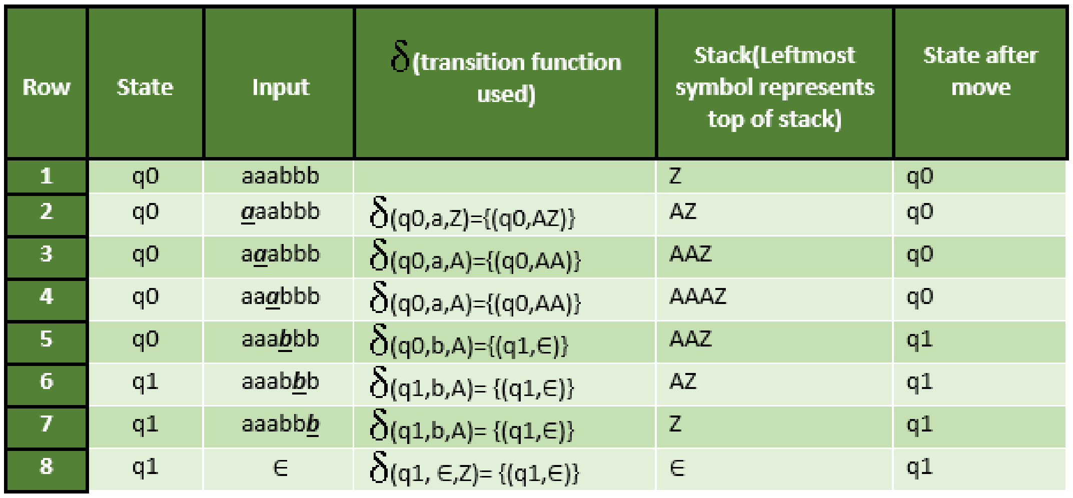
Turnstile Notation

⊢ sign is called a "turnstile notation" and represents
one move.
⊢* sign represents a sequence of moves.
Eg- (p, b, T) ⊢ (q, w, α)
This implies that while taking a transition from state p to state q, the input symbol 'b' is consumed, and the top of the stack 'T' is replaced by a new string 'α'

Example : Define the pushdown automata for language {anbn | n > 0}
Solution : M = where Q = { q0, q1 } and Σ = { a, b } and Γ = { A, Z } and δ is given by :

δ(q0, a, Z) = { (q0, AZ) }
δ(q0, a, A) = { (q0, AA) }
δ(q0, b, A) = { (q1, ∈) }
δ(q1, b, A) = { (q1, ∈) }
δ(q1, ∈, Z) = { (q1, ∈) }

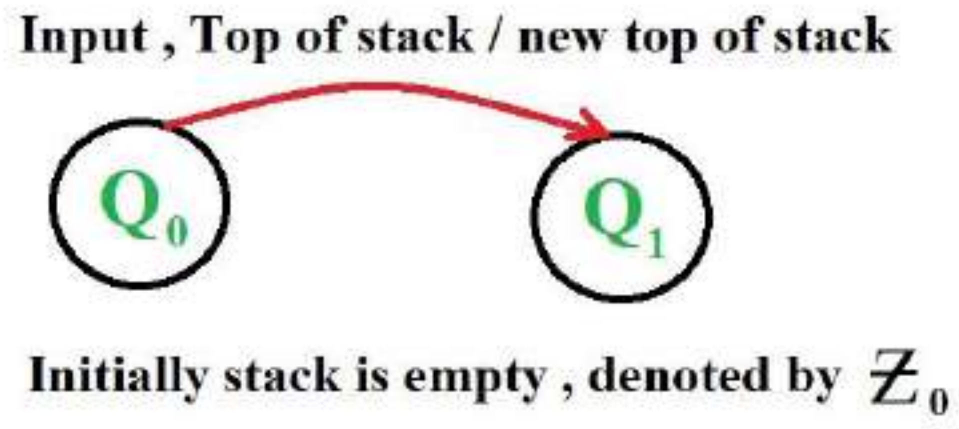
Let us see how this automata works for aaabbb.



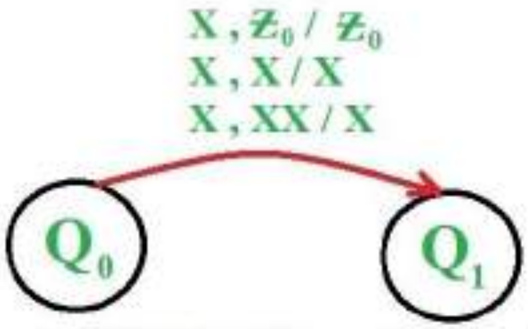
Row	State	Input	δ(transition function used)	Stack(Leftmost symbol represents top of stack)	State after move
1	q0	aaabbb		Z	q0
2	q0	*a*aabbb	δ(q0,a,Z)={(q0,AZ)}	AZ	q0
3	q0	a*a*abbb	δ(q0,a,A)={(q0,AA)}	AAZ	q0
4	q0	aa*a*bbb	δ(q0,a,A)={(q0,AA)}	AAAZ	q0
5	q0	aaa*b*bb	δ(q0,b,A)={(q1,∈)}	AAZ	q1
6	q1	aaab*b*b	δ(q1,b,A)= {(q1,∈)}	AZ	q1
7	q1	aaabb*b*	δ(q1,b,A)= {(q1,∈)}	Z	q1
8	q1	∈	δ(q1, ∈,Z)= {(q1,∈)}	∈	q1

Explanation : Initially, the state of automata is q0 and symbol on stack is Z and the input is aaabbb as shown in row 1. On reading 'a' (shown in bold in row 2), the state will remain q0 and it will push symbol A on stack. On next 'a' (shown in row 3), it will push another symbol A on stack. After reading 3 a's, the stack will be AAAZ with A on the top. After reading 'b' (as shown in row 5), it will pop A and move to state q1 and stack will be AAZ. When all b's are read, the state will be q1 and stack will be Z. In row 8, on input symbol '∈' and Z on stack, it will pop Z and stack will be empty. This type of acceptance is known as **acceptance by empty stack.**

Representation of State Transition -

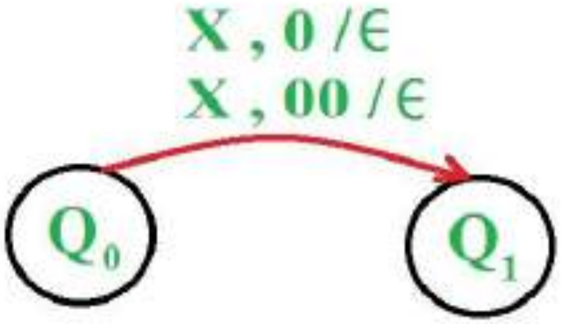

Input , Top of stack / new top of stack
Q_0
Q_1
Initially stack is empty , denoted by Z_0

Representation of Push in a PDA -



Push an element 'X' if stack is empty (denoted by Z_0), or if there is 1 'X' on top of stack or if there are 2 or more 'X' on top of stack

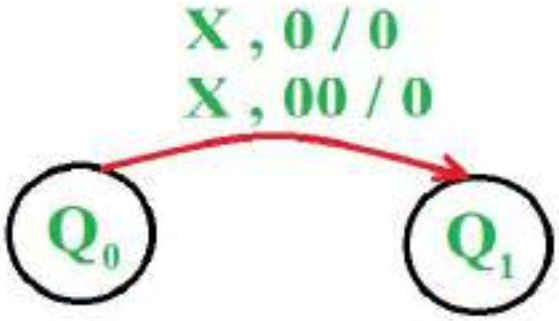
Representation of Pop in a PDA -



Pop an element 'X' if we have 1 or more 0's on top of stack

∈ shows deletion or pop

Representation of Ignore in a PDA -

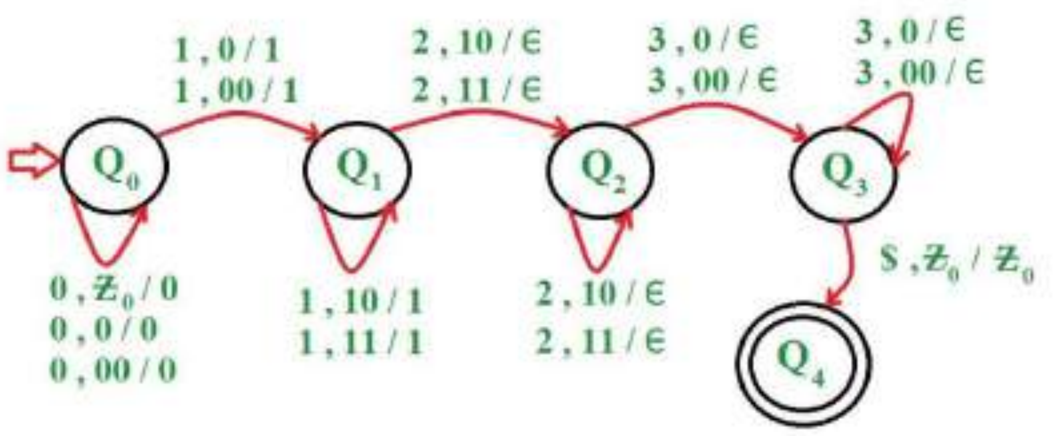


Ignore an element 'X' if we have 1 or more 0's on top of stack

Q) Construct a PDA for language L = {0n1m2m3n | n>=1, m>=1}

Approach used in this PDA -
First 0's are pushed into stack. Then 1's are pushed into stack.
Then for every 2 as input a 1 is popped out of stack. If some 2's are still left and top of stack is a 0 then string is not accepted by the PDA. Thereafter if 2's are finished and top of stack is a 0 then for every 3 as input equal number of 0's are popped out of stack. If string is finished and stack is empty then string is accepted by the PDA otherwise not accepted.

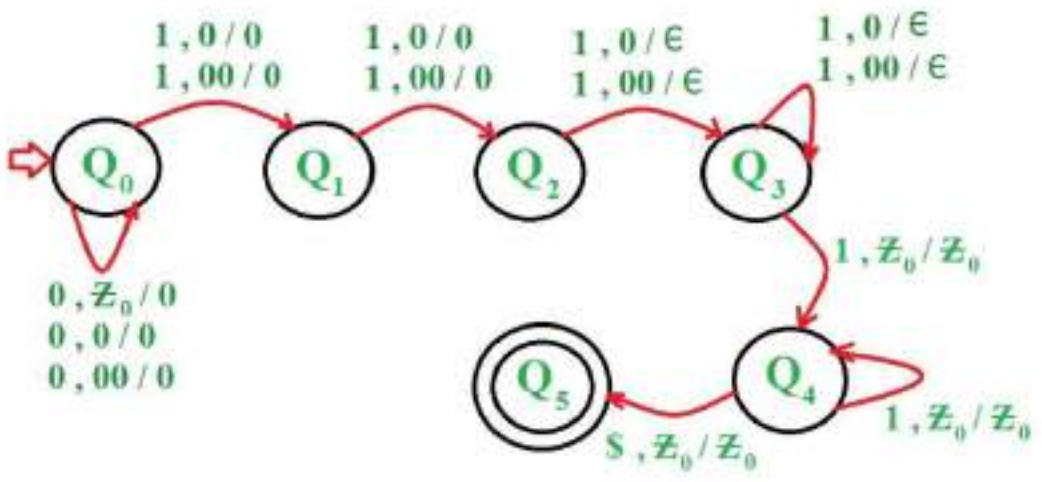
- **Step-1:** On receiving 0 push it onto stack. On receiving 1, push it onto stack and goto next state
- **Step-2:** On receiving 1 push it onto stack. On receiving 2, pop 1 from stack and goto next state
- **Step-3:** On receiving 2 pop 1 from stack. If all the 1's have been popped out of stack and now receive 3 then pop a 0 from stack and goto next state
- **Step-4:** On receiving 3 pop 0 from stack. If input is finished and stack is empty then goto last state and string is accepted



Q) Construct a PDA for language L = {0n1m | n >= 1, m >= 1, m > n+2}

Approach used in this PDA -
First 0's are pushed into stack.When 0's are finished, two 1's are ignored. Thereafter for every 1 as input a 0 is popped out of stack. When stack is empty and still some 1's are left then all of them are ignored.

- **Step-1:** On receiving 0 push it onto stack. On receiving 1, ignore it and goto next state
- **Step-2:** On receiving 1, ignore it and goto next state
- **Step-3:** On receiving 1, pop a 0 from top of stack and go to next state
- **Step-4:** On receiving 1, pop a 0 from top of stack. If stack is empty, on receiving 1 ignore it and goto next state
- **Step-5:** On receiving 1 ignore it. If input is finished then goto last state



In automata theory and context-free grammars, the relationship between Pushdown Automata (PDA) and Context-Free Grammars (CFG) is fundamental.

Relation Between PDA and CFG

A Pushdown Automaton (PDA) is a type of automaton that uses a stack to manage additional information. Using memory as stack makes it more powerful than a finite automaton.

Pushdown automata can recognize context-free languages, which are languages generated by Context-Free Grammars (CFGs).

A Context-free Grammar consists of:

A set of variables or non-terminal symbols.

A set of terminal symbols.

A start symbol.

A set of production rules that define how terminals and non-terminals can be combined.

The goal here is to construct a CFG equivalent to a PDA, which means the CFG generates the same language that the PDA recognizes.

Conversion of PDA to CFG

To convert a PDA into an equivalent CFG, we assume the PDA M = (Q, , , , q0, z0, F) accepts a language L by an empty stack.

The CFG G = (VN, , P, S) is constructed following specific rules that ensure it generates the same language as the PDA.

Let us see the steps one by one to convert PDA to CFG.

Start Symbol Production

For every state qi in the PDA, we add a production rule to the CFG:

S→[q0,z0,qi]S→[q0,z0,qi]

If the PDA has n states, there will be n productions originating from the start symbol S.

Handling Transitions with Empty Stack Operation

For each transition function of the form (q, a, Y) → (r,), where (q, r) &in; Q, a &in; , and Y &in; , a production rule is added to the CFG as follows −

[qYr]→a[qYr]→a

This rule handles cases where the PDA reads a symbol and pops the stack without pushing anything.

Handling Transitions with Stack Operations

For transition functions of the form (q, a, Y) → (r, Y_1, Y_2, , Y_k), a production rule is added for each combination of intermediate states q1,q2, ... ,qk as:

[qYqk]→a[rY1q1][q1Y2q2]…[qk−1Ykqk][qYqk]→a[rY1q1][q1Y2q2]…[qk−1Ykqk]

This rule addresses scenarios where the PDA pushes multiple symbols onto the stack in response to an input symbol.

Transition Table

The transition table with regards to converting CFG to PDA is as follows −

Sr	State	Unread Input	Stack	Transition
1	q0	abbccbba		1
2	q0	abbcbba	S	1
3	q0	abbcbba	aSa	2
4	q1	bbcbba	Sa	5
5	q0	bbcbba	bSba	3
6	q2	bcbba	Sba	6
7	q0	bcbba	bsbba	3
8	q2	cbba	Sbba	6

9	q0	cbba	cbba	4
10	q3	bba	bba	7
11	q2	ba	ba	6
12	q1			5

Example 1: Constructing a CFG from PDA

Lets apply these steps to an example to make the conversion process clearer.

Given PDA,

δ(q0,a,z0)→(q0,z1z0)δ(q0,a,z0)→(q0,z1z0)

δ(q0,a,z1)→(q0,z1z1)δ(q0,a,z1)→(q0,z1z1)

δ(q0,b,z1)→(q1,λ)δ(q0,b,z1)→(q1,λ)

δ(q1,b,z1)→(q1,λ)δ(q1,b,z1)→(q1,λ)

δ(q1,b,z0)→(q1,z2z0)δ(q1,b,z0)→(q1,z2z0)

δ(q1,c,z2)→(q2,λ)δ(q1,c,z2)→(q2,λ)

δ(q2,λ,z0)→(q2,λ)δ(q2,λ,z0)→(q2,λ)

This PDA has three states: q0, q1, and q2.

Step 1: Start Symbol Production

The following productions are added −

S→[q0z0q0]/[q0z0q1]/[q0z0q2]S→[q0z0q0]/[q0z0q1]/[q0z0q2]

Step 2: Transitions with Empty Stack

The transitions δ(q0, b, z1) → (q1, λ), δ(q1, b, z1) → (q1, λ), and δ(q1, c, z2) → (q2, λ) lead to the following productions:

[q0z0q1]→b[q0z0q1]→b

[q1z1q1]→b[q1z1q1]→b

[q1z2q2]→c[q1z2q2]→c

[q2z2q2]→c[q2z2q2]→c

[q2z0q2]→ϵ[q2z0q2]→ϵ

Step 3: Transitions with Stack Operations

For the transition (q0, a, z0) → (q0,z1, z0), the following productions are added −

[q0z0q0]→a[q0z1q0][q0z0q0][q0z0q0]→a[q0z1q0][q0z0q0]

[q0z0q0]→a[q0z1q0][q1z0q0][q0z0q0]→a[q0z1q0][q1z0q0]

[q0z0q1]→a[q0z1q0][q0z0q1][q0z0q1]→a[q0z1q0][q0z0q1]

[q0z0q1]→a[q0z1q1][q1z0q1][q0z0q1]→a[q0z1q1][q1z0q1]

[q0z0q2]→a[q0z1q0][q0z0q2][q0z0q2]→a[q0z1q0][q0z0q2]

[q0z0q2]→a[q0z1q1][q1z0q2][q0z0q2]→a[q0z1q1][q1z0q2]

The CFG that generates the same language as the PDA is a collection of all these production rules.

Turing Machin

A Turing Machine is an accepting device which accepts the languages (recursively enumerable set) generated by type 0 grammars. It was invented in 1936 by Alan Turing.

Definition

A Turing Machine (TM) is a mathematical model which consists of an infinite length tape divided into cells on which input is given. It consists of a head which reads the input tape. A state register stores the state of the Turing machine. After reading an input symbol, it is replaced with another symbol, its internal state is changed, and it moves from one cell to the right or left. If the TM reaches the final state, the input string is accepted, otherwise rejected.

A TM can be formally described as a 7-tuple (Q, X, ∑, δ, q0, B, F) where −

Q is a finite set of states

X is the tape alphabet

∑ is the input alphabet

δ is a transition function; δ : Q × X → Q × X × {Left_shift, Right_shift}.

q0 is the initial state

B is the blank symbol

F is the set of final states

Comparison with the previous automaton

The following table shows a comparison of how a Turing machine differs from Finite Automaton and Pushdown Automaton.

Machine	Stack Data Structure	Deterministic?
Finite Automaton	N.A	Yes
Pushdown Automaton	Last In First Out(LIFO)	No
Turing Machine	Infinite tape	Yes

Example of Turing machine

Turing machine M = (Q, X, ∑, δ, q0, B, F) with

Q = {q0, q1, q2, qf}

X = {a, b}

∑ = {1}

q0 = {q0}

B = blank symbol

F = {qf }

δ is given by −

Tape alphabet symbol	Present State q0	Present State q1	Present State q2
a	1Rq1	1Lq0	1Lqf
b	1Lq2	1Rq1	1Rqf

Here the transition 1Rq1 implies that the write symbol is 1, the tape moves right, and the next state is q1. Similarly, the transition 1Lq2 implies that the write symbol is 1, the tape moves left, and the next state is q2.

Time and Space Complexity of a Turing Machine

For a Turing machine, the time complexity refers to the measure of the number of times the tape moves when the machine is initialized for some input symbols and the space complexity is the number of cells of the tape written.

Time complexity all reasonable functions −

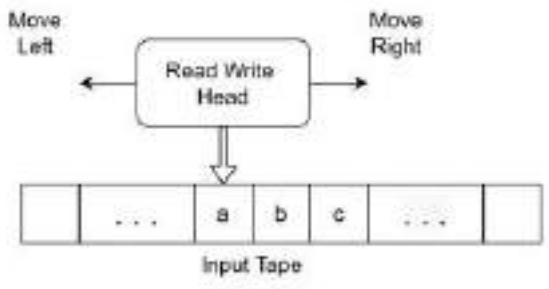
T(n) = O(n log n)

TM's space complexity −

S(n) = O(n)

Mechanical Diagram of Turing Machine

A Turing machine consists of three main parts: an input tape, a read-write head, finite control.



The input tape contains the input alphabets. It has an infinite number of blanks at the left and right side of the input symbols. The read-write head reads an input symbol from the input tape and sends it to the finite control. The machine must be in some state. In the finite control, the transitional functions are written. Based on the present state and the present input, a suitable transitional function is executed.

Operations of Turing Machine

When a transitional function is executed, the Turing machine performs these operations −

The machine goes into some state.

The machine writes a symbol in the cell of the input tape from where the input symbol was scanned.

The machine moves the reading head to the left or right or halts.

Instantaneous Description (ID) of Turing Machine

The Instantaneous Description (ID) of a Turing machine remembers the following at a given instance of time −

The contents of all the cells of the tape, starting from the rightmost cell up to at least the last cell, containing a non-blank symbol and containing all cells up to the cell being scanned.

The cell currently being scanned by the read-write head.

The state of the machine.

Example of Turing Machine

Let us look at an example of a Turing machine that accepts the language L={anbn,n≥1}L={anbn,n≥1}. This language consists of strings with equal number of a's followed by b's, where the number of a's and b's is at least 1.

Transition Functions (B denotes blank)

$\delta(q0,a)\rightarrow(q1,X,R)$$\delta(q0,a)\rightarrow(q1,X,R)$

$\delta(q1,a)\rightarrow(q1,a,R)$$\delta(q1,a)\rightarrow(q1,a,R)$

$\delta(q1,b)\rightarrow(q2,Y,L)$$\delta(q1,b)\rightarrow(q2,Y,L)$

$\delta(q2,a)\rightarrow(q2,a,L)$$\delta(q2,a)\rightarrow(q2,a,L)$

$\delta(q2,X)\rightarrow(q0,X,R)$$\delta(q2,X)\rightarrow(q0,X,R)$

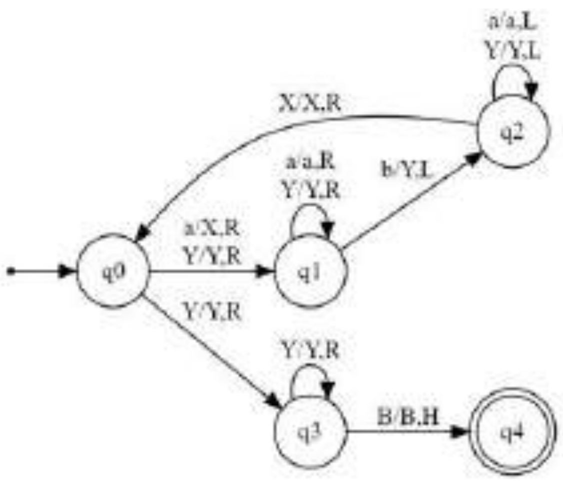
$\delta(q1,Y)\rightarrow(q1,Y,R)$$\delta(q1,Y)\rightarrow(q1,Y,R)$

$\delta(q2,Y)\rightarrow(q2,Y,L)$$\delta(q2,Y)\rightarrow(q2,Y,L)$

$\delta(q0,Y)\rightarrow(q3,Y,R)$$\delta(q0,Y)\rightarrow(q3,Y,R)$

$\delta(q3,Y)\rightarrow(q3,Y,R)$$\delta(q3,Y)\rightarrow(q3,Y,R)$

$\delta(q3,B)\rightarrow(q4,B,H)$$\delta(q3,B)\rightarrow(q4,B,H)$



Explanation of Transition Functions

Let's break down how this Turing machine works −

It starts in state q0 and replaces the first 'a' with 'X'.

It moves right, keeping 'a's unchanged, until it finds the first 'b'.

It replaces the first 'b' with 'Y' and moves left.

It moves left until it finds 'X', then goes back to step 1.

If it finds 'Y' instead of 'b', it moves right until it finds a blank, then halts in an accepting state.

Instantaneous Description (ID) for the string "aaabbb"

BaaabbbB→BXaabbbBBaaabbbB→BXaabbbB

BXaabbbB→BXaaYbbBBXaabbbB→BXaaYbbB

BXaaYbbB→BXXaYbbBBXaaYbbB→BXXaYbbB

BXXaYbbB→BXXaYYbBBXXaYbbB→BXXaYYbB

BXXaYYbB→BXXXYYbBBXXaYYbB→BXXXYYbB

BXXXYYbB→BXXXYYYB (Halt in accepting state)BXXXYYbB→BXXXYYYB (Halt in accepting state)

This ID shows how the Turing machine processes the string "aaabbb" step by step, replacing a's with X's and b's with Y's until it accepts the string.

undecidability is an important concept in Automata Theory. As per this concept, a problem cannot be solved by any algorithm, meaning there is no Turing Machine (TM) that can decide whether a given statement or problem is True or False.

Decidable Problems

A problem is considered decidable if there exists a Turing Machine that can provide a definitive "yes" or "no" answer for every possible input within a finite amount of time.

In other words, the Turing Machine must halt on every input, either accepting or rejecting it.

Example − The problem of determining whether a given string is accepted by a Deterministic Finite Automaton (DFA) is decidable. A Turing Machine can simulate the DFA and decide whether to accept or reject the string.

Recognizable Problems

A problem is recognizable or Turing Recognizable, if there exists a Turing Machine that will accept every string in the language but may either reject or run indefinitely on strings not in the language.

Unlike decidable problems, recognizable problems do not guarantee that the machine will halt for all inputs or not.

Undecidable Problems

A problem is considered undecidable if no Turing Machine can be constructed that will always halt with a correct "yes" or "no" answer for every input. In other words, there is no algorithm that can decide the problem in all cases.

Example − The Halting Problem, which asks whether a given Turing Machine will halt on a given input, is a classic example of an undecidable problem.

The Halting Problem

The Halting Problem is one of the most well-known undecidable problems. It asks whether a Turing Machine, for a given input, will halt (stop executing) or continue to run forever.

Proof of Undecidability

Suppose there is a Turing Machine H that can decide the Halting Problem. Now, construct a new Turing Machine D that does the following −

For an input x, if H(x, x) (where x is both the machine and input) halts, D(x) will run indefinitely.

If H(x, x) does not halt, then D(x) will halt.

This leads to a contradiction because if D runs on its own description, it both halts and does not halt. Therefore, H cannot exist, proving that the Halting Problem is undecidable.

The Post Correspondence Problem (PCP)

Another interesting problem is the Post Correspondence Problem; it is about finding a match between two lists of strings. We have two lists of strings, the question is whether there is a sequence of indices that, when applied to both lists, produces the same string.

Proof of Undecidability

The undecidability of PCP is proven by reducing it to another undecidable problem. Suppose there exists a solution to PCP, then it can be used to solve the Halting Problem, which we know is undecidable. Since solving the Halting Problem is impossible, PCP must also be undecidable.

The Blank Tape Halting Problem

Another example of undecidability is the Blank Tape Halting Problem, which asks whether a Turing Machine will halt when started with a blank tape.

Proof of Undecidability

The problem is reduced from the Halting Problem. If we could decide whether a Turing Machine halts on a blank tape, we could decide whether it halts on any arbitrary input. However, since the Halting Problem is undecidable, the Blank Tape Halting Problem is also undecidable.

The Virus Detection Problem

Another interesting problem is Virus Detection Problem which is also undecidable. It asks whether there exists an algorithm that can decide if a program is a virus or not.

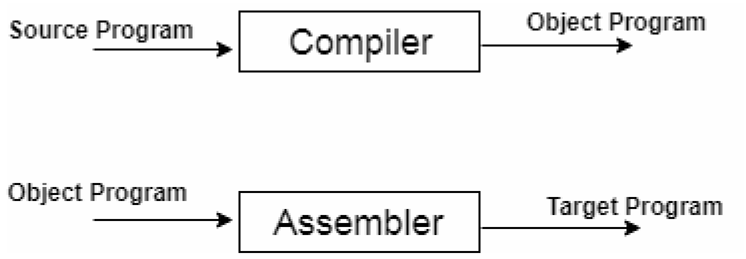
Proof of Undecidability

If a virus detection program could perfectly identify whether a program halts (runs to completion) or runs indefinitely (spreads like a virus), it could be used to solve the Halting Problem. However, since the Halting Problem is undecidable, virus detection is also undecidable.

UNIT IV

Introduction to Compiler

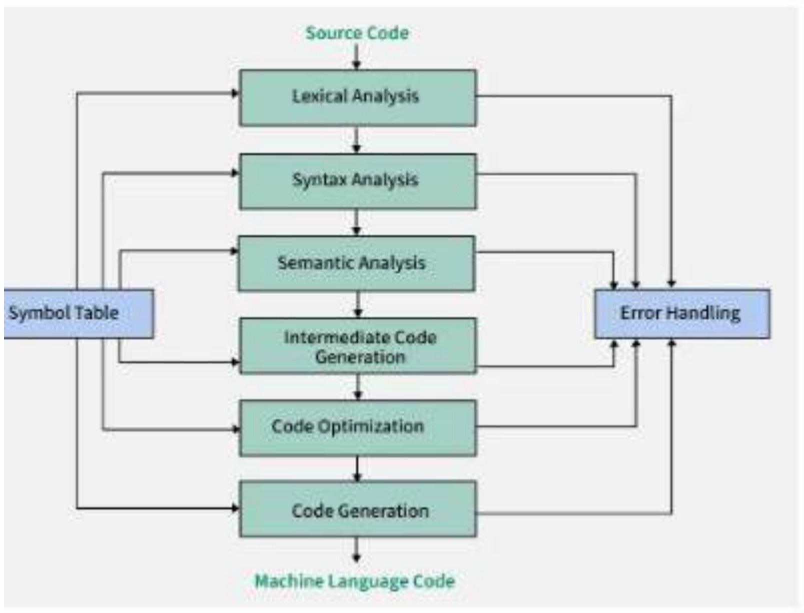
- A compiler is a translator that converts the high-level language into the machine language.
- High-level language is written by a developer and machine language can be understood by the processor.
- Compiler is used to show errors to the programmer.
- The main purpose of compiler is to change the code written in one language without changing the meaning of the program.
- When you execute a program which is written in HLL programming language then it executes into two parts.
- In the first part, the source program compiled and translated into the object program (low level language).
- In the second part, object program translated into the target program through the assembler.



Compiler Phases

The compilation process contains the sequence of various phases. Each phase takes source program in one representation and produces output in another representation. Each phase takes input from its previous stage.

There are the various phases of compiler:



Phases of Compiler

Lexical Analysis:

Lexical analyzer phase is the first phase of compilation process. It takes source code as input. It reads the source program one character at a time and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens.

Syntax Analysis

Syntax analysis is the second phase of compilation process. It takes tokens as input and generates a parse tree as output. In syntax analysis phase, the parser checks that the expression made by the tokens is syntactically correct or not.

Semantic Analysis

Semantic analysis is the third phase of compilation process. It checks whether the parse tree follows the rules of language. Semantic analyzer keeps track of identifiers, their types and expressions. The output of semantic analysis phase is the annotated tree syntax.

Intermediate Code Generation

In the intermediate code generation, compiler generates the source code into the intermediate code. Intermediate code is generated between the high-level language and the machine language. The intermediate code should be generated in such a way that you can easily translate it into the target machine code.

Code Optimization

Code optimization is an optional phase. It is used to improve the intermediate code so that the output of the program could run faster and take less space. It removes the unnecessary lines of the code and arranges the sequence of statements in order to speed up the program execution.

Code Generation

Code generation is the final stage of the compilation process. It takes the optimized intermediate code as input and maps it to the target machine language. Code generator translates the intermediate code into the machine code of the specified computer.

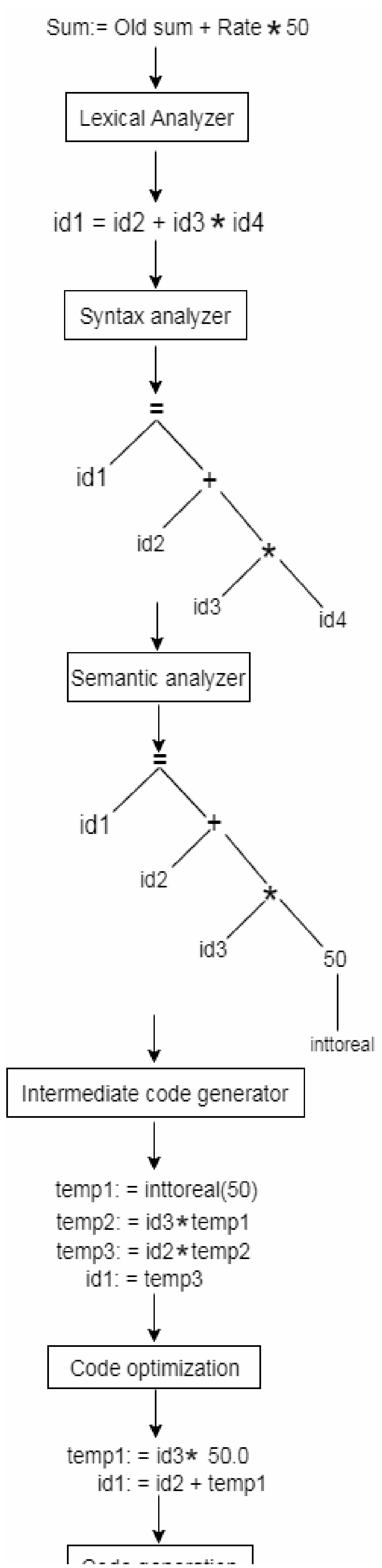
Symbol Table – It is a data structure being used and maintained by the compiler, consisting of all the identifier's names along with their types. It helps the compiler to function smoothly by finding the identifiers quickly.

Error Handling in Phases of Compiler

Error Handling refers to the mechanism in each phase of the compiler to detect, report and recover from errors without terminating the entire compilation process.

- **Lexical Analysis**: Detects errors in the character stream and ensures valid token formation.
    - **Example**: Identifies illegal characters or invalid tokens (e.g., @var as an identifier).
- **Syntax Analysis:** Checks for structural or grammatical errors based on the language's grammar.
    - **Example**: Detects missing semicolons or unmatched parentheses.

Example:

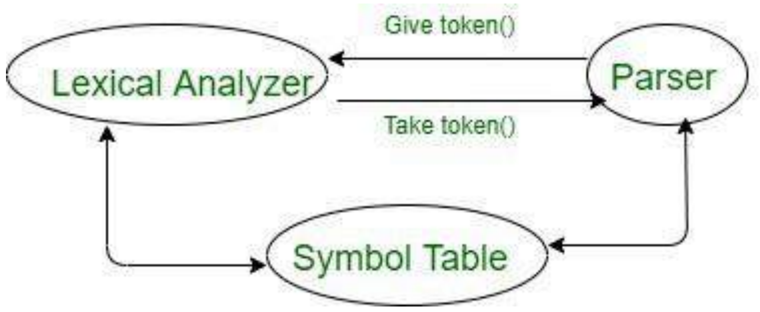


```
temp1: = inttoreal(50)
temp2: = id3∗temp1
temp3: = id2∗temp2
   id1: = temp3
```

```
temp1: = id3∗ 50.0
   id1: = id2 + temp1
```

Lexical Analyzer :

- It is the first phase of a compiler is known as **Scanner** (It's scan the program).
- Lexical Analyzer will divide the program into some meaningful strings which are known as a token.



Types of token as following –

1. Identifier
2. Keyword
3. Operator
4. Constants
5. Special symbol(@, $, #)

Categories of Tokens

- **Keywords:** In C programming, keywords are reserved words with specific meanings used to define the language's structure like if, else, for, and void. These cannot be used as variable names or identifiers, as doing so causes compilation errors. C programming has a total of 32 keywords.
- **Identifiers:** Identifiers in C are names for variables, functions, arrays, or other user-defined items. They must start with a letter or an underscore (_) and can include letters, digits, and underscores. C is case-sensitive, so uppercase and lowercase letters are different. Identifiers cannot be the same as keywords like if, else or for.
- **Constants:** Constants are fixed values that cannot change during a program's execution, also known as literals. In C, constants include types like integers, floating-point numbers, characters, and strings.
- **Operators:** Operators are symbols in C that perform actions on variables or other data items, called operands.
- **Special Symbols:** Special symbols in C are compiler tokens used for specific purposes, such as separating code elements or defining operations. Examples include **;** (semicolon) to end statements, **,** (comma) to separate values, **{}** (curly braces) for code blocks, and [] (square brackets) for arrays. These symbols play a crucial role in the program's structure and syntax.

Above is the terminologies of token which is the key component for working in Lexical Analyzer. Now, with the help of example, you will see how it works.

Let’s consider the following C program given below to understands the working.

int main)(

}

x = y+z;

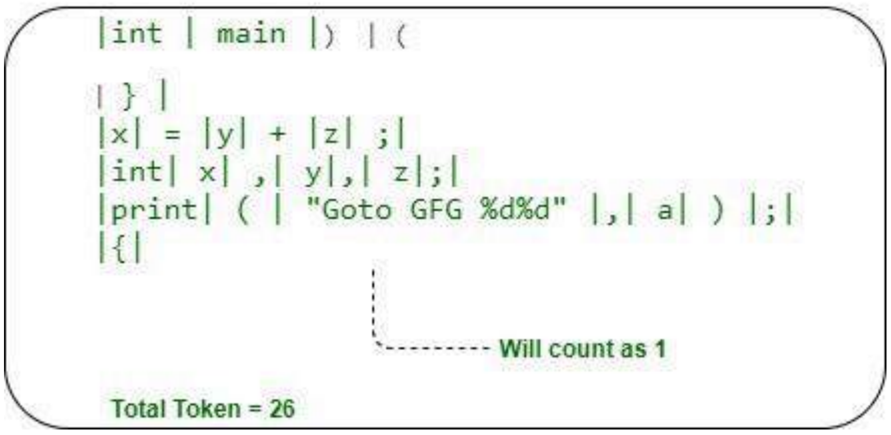
int x, y, z;

print("Goto GFG %d%d", a);

{

In the first phase, the compiler doesn’t check the syntax. So, here this program as input to the lexical analyzer and convert it into the tokens. So, tokenization is one of the important functioning of lexical analyzer.

The total number of token for this program is 26. Below given is the diagram of how it will count the token.



```
|int | main |) | (

| } |
|x| = |y| + |z| ;|
|int| x| ,| y|,| z|;|
|print| ( | "Goto GFG %d%d" |,| a| ) |;|
|{|
```

In this above diagram, you can check and count the number of tokens and can understand how tokenization works in lexical analyzer phase.

This is how you can understand each phase in compiler with clarity and will get an idea of how compiler works internally and each phase of the compiler is the key step.

Role of Lexical Analyzer

Clean up the source code: Source code often contains extra characters that the compiler or interpreter doesn’t need, like spaces, tabs, newlines, and comments. The lexical analyzer removes these to make the next stage of

processing easier. Think of it like cleaning up your desk before starting a project.
Keep track of errors: While cleaning up, the lexical analyzer might encounter invalid characters or sequences of characters that don't form a valid token. For example, it might find a character that's not allowed in the programming language. When this happens, it reports an error message and ideally pinpoints the location of the error in the original source code (e.g., line number and character position). This helps programmers find and fix mistakes.
Figure out the basic building blocks (tokens): This is the core job. The lexical analyzer scans the cleaned-up source code and groups characters together into meaningful units called *tokens*. These tokens are the fundamental building blocks of the program. Examples of tokens include: The lexical analyzer doesn't understand the *meaning* of these tokens; it just identifies what they are.

- **Keywords:** if, else, while, for, int, float, etc. (reserved words with special meanings)
- **Identifiers:** Variable names, function names, etc. (user-defined names)
- **Operators:** +, -, *, /, =, ==, <, >, etc. (symbols that perform operations)
- **Literals:** Numbers (e.g., 10, 3.14), strings (e.g., "hello"), boolean values (true, false), etc. (represent constant values)
- **Punctuation:** ;, ,, (,), {, }, etc. (used for structure and grouping)

Read the source code character by character: The lexical analyzer reads the source code one character at a time, grouping these characters into tokens. It's like reading a sentence word by word, but at a more fundamental level – character by character to form the words (tokens). This character-by- character reading allows it to recognize even complex tokens.

Input Buffering in Compiler Design

The lexical analyzer scans the input from left to right one character at a time. It uses two pointers begin ptr(**bp**) and forward ptr(**fp**) to keep track of the pointer of the input scanned.
Input buffering is an important concept in compiler design that refers to the way in which the compiler reads input from the source code. In many cases, the compiler reads input one character at a time, which can be a slow and inefficient process. Input buffering is a technique that allows the compiler to read input in larger chunks, which can improve performance and reduce overhead.

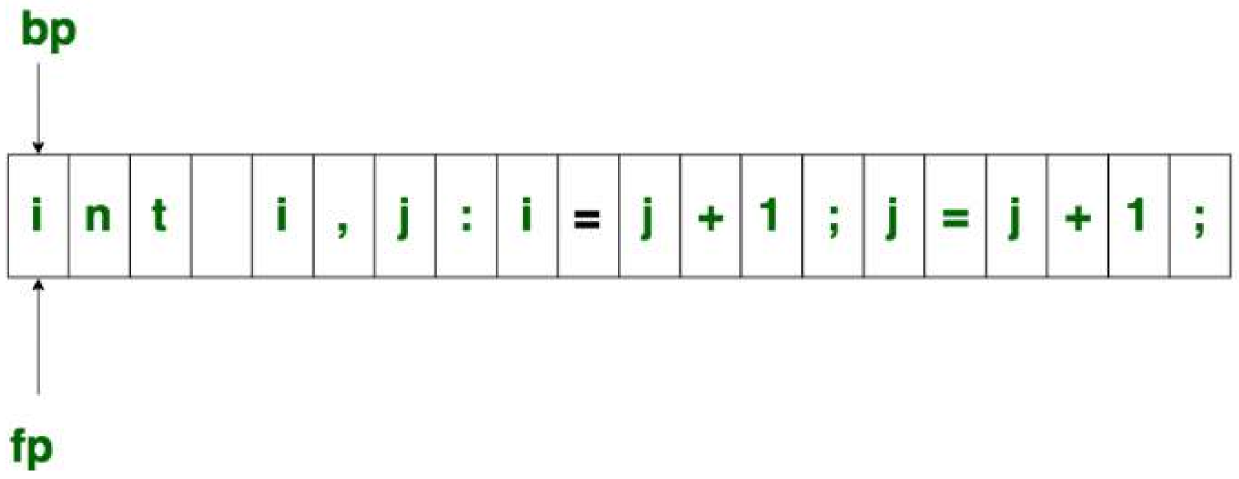
1. The basic idea behind input buffering is to read a block of input from the source code into a buffer, and then process that buffer before reading the next block. The size of the buffer can vary depending on the specific needs

of the compiler and the characteristics of the source code being compiled. For example, a compiler for a high-level programming language may use a larger buffer than a compiler for a low-level language, since high-level languages tend to have longer lines of code.

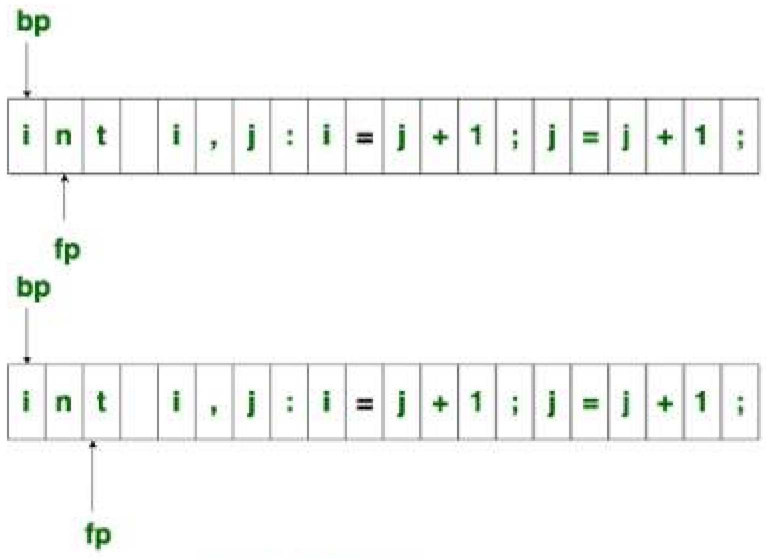
2. One of the main advantages of input buffering is that it can reduce the number of system calls required to read input from the source code. Since each system call carries some overhead, reducing the number of calls can improve performance. Additionally, input buffering can simplify the design of the compiler by reducing the amount of code required to manage input.

However, there are also some potential disadvantages to input buffering. For example, if the size of the buffer is too large, it may consume too much memory, leading to slower performance or even crashes. Additionally, if the buffer is not properly managed, it can lead to errors in the output of the compiler.

Overall, input buffering is an important technique in compiler design that can help improve performance and reduce overhead. However, it must be used carefully and appropriately to avoid potential problems.

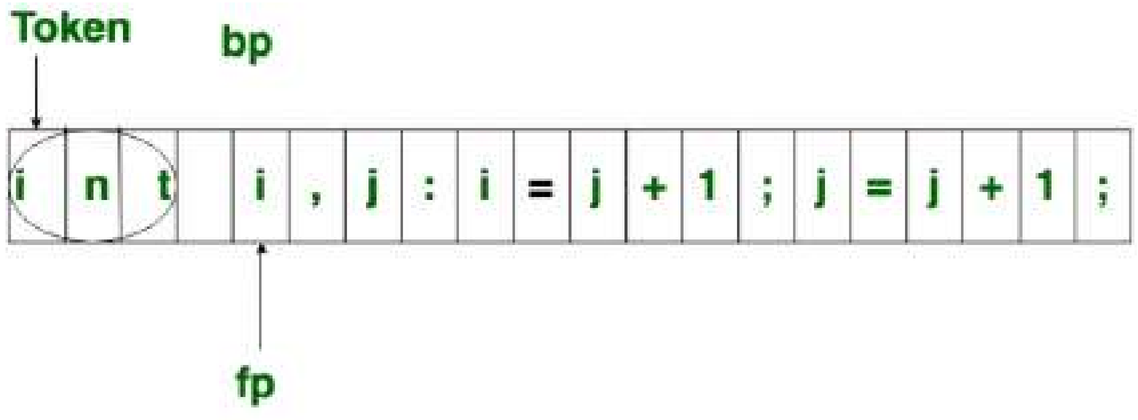


Initially both the pointers point to the first character of the input string as shown below

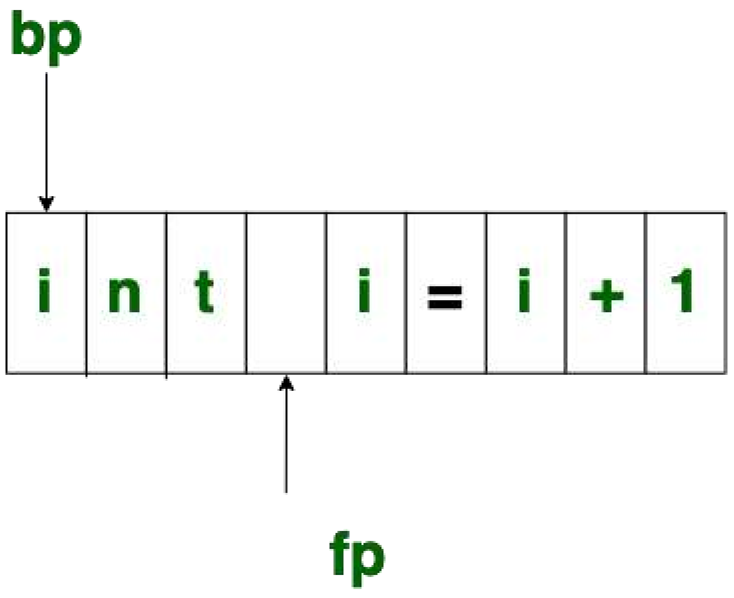

bp
i n t i , j : i = j + 1 ; j = j + 1 ;
fp
bp
i n t i , j : i = j + 1 ; j = j + 1 ;
fp

Input Buffering

The forward ptr moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. In above example as soon as ptr (fp) encounters a blank space the lexeme "int" is identified. The fp will be moved ahead at white space, when fp encounters white space, it ignore and moves ahead. then both the begin ptr(bp) and forward ptr(fp) are set at next token. The input character is thus read from secondary storage, but reading in this way from secondary storage is costly. hence buffering technique is used.A block of data is first read into a buffer, and then second by lexical analyzer. there are two methods used in this context: One Buffer Scheme, and Two Buffer Scheme. These are explained as following below.

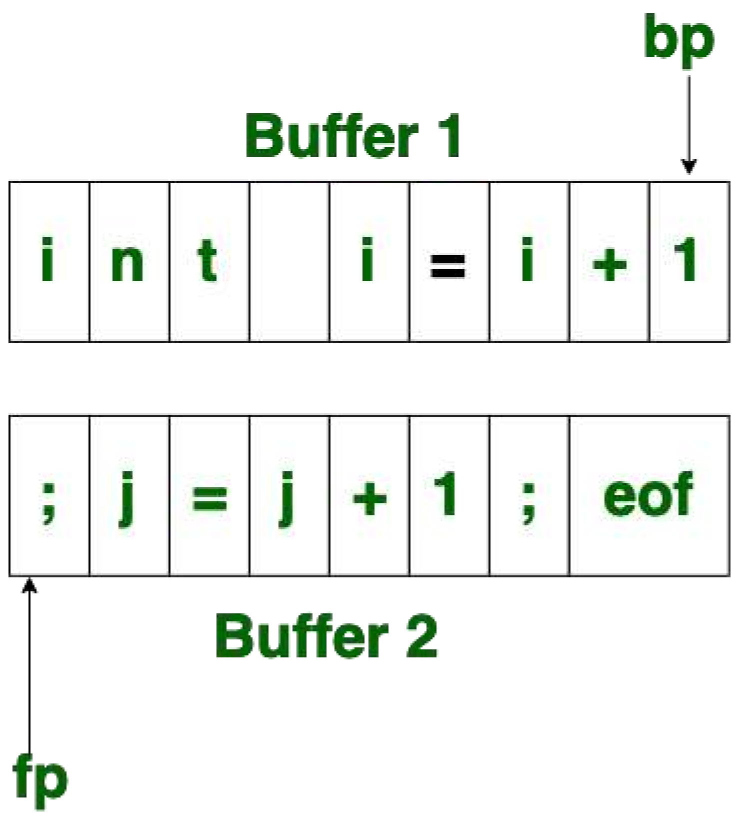


1. **One Buffer Scheme:** In this scheme, only one buffer is used to store the input string but the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer
   has to be refilled, that makes overwriting the first of lexeme.



One buffer scheme storing input string

2. **Two Buffer Scheme:** To overcome the problem of one buffer scheme, in this method two buffers are used to store the input string. the first buffer and second buffer are scanned alternately. when end of current buffer is reached the other buffer is filled. the only problem with this method is that if length of the lexeme is longer than length of the buffer then scanning input cannot be scanned completely. Initially both the bp and fp are pointing to the first character of first buffer. Then the fp moves towards right in search of end of lexeme. as soon as blank character is recognized, the string between bp and fp is identified as corresponding token. to identify, the boundary of first buffer end of buffer character should be placed at the end first buffer. Similarly end of second buffer is also recognized by the end of buffer mark present at the end of second buffer. when fp encounters first **eof**, then one can recognize end of first buffer and hence filling up second buffer is started. in the same way when second **eof** is obtained then it indicates of second buffer. alternatively both the buffers can be filled up until end of the input program and stream of tokens is identified. This **eof** character introduced at the end is calling **Sentinel** which is used to identify the end of buffer.



Two buffer scheme storing input string

Advantages:

Input buffering can reduce the number of system calls required to read input from the source code, which can improve performance.
Input buffering can simplify the design of the compiler by reducing the amount of code required to manage input.

Disadvantages:

If the size of the buffer is too large, it may consume too much memory, leading to slower performance or even crashes.
If the buffer is not properly managed, it can lead to errors in the output of the compiler.
Overall, the advantages of input buffering generally outweigh the

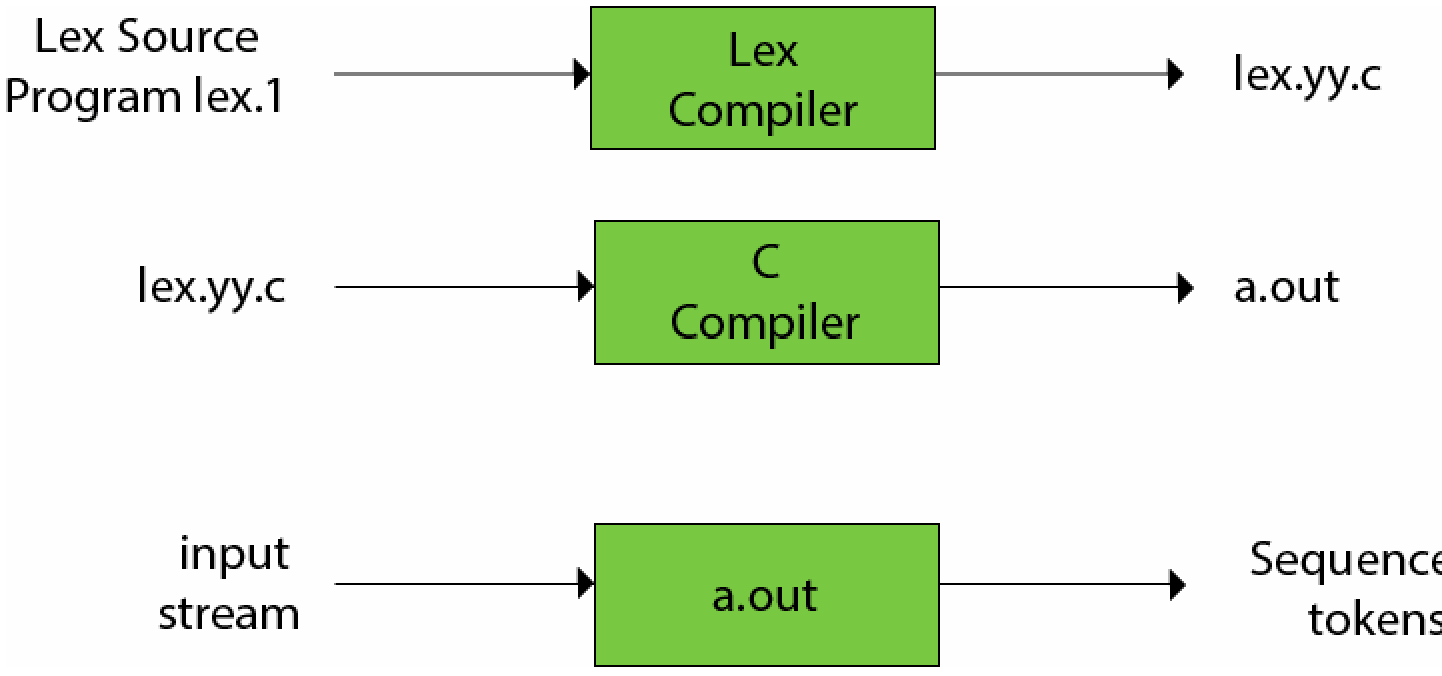
disadvantages when used appropriately, as it can improve performance and simplify the compiler design.

LEX

- Lex is a program that generates lexical analyzer. It is used with YACC parser generator.
- The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
- It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

The function of Lex is as follows:

- Firstly lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.
- Finally C compiler runs the lex.yy.c program and produces an object program a.out.
- a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



Lex file format

A Lex program is separated into three sections by %% delimiters. The formal of Lex source is as follows:

1. { definitions }
2. %%
3. { rules }
4. %%
5. { user subroutines }

Definitions include declarations of constant, variable and regular definitions.

Rules define the statement of form p1 {action1} p2 {action2}.......... pn {action}.

Where **pi** describes the regular expression and **action1** describes the actions what action the lexical analyzer should take when pattern pi matches a lexeme.

User subroutines are auxiliary procedures needed by the actions. The subroutine can be loaded with the lexical analyzer and compiled separate ly.

Recognition of Tokens

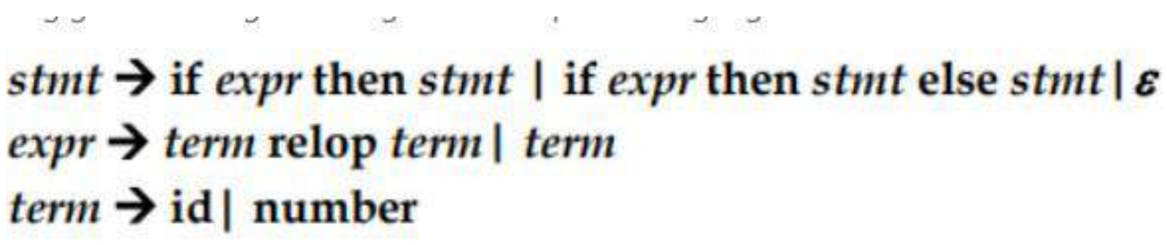
- Tokens obtained during lexical analysis are **recognized by Finite Automata**.
- Finite Automata (FA) is a simple idealized machine that can be used to recognize patterns within input taken from a character set or alphabet (denoted as C). The primary task of an FA is to accept or reject an input based on whether the defined pattern occurs within the input.
- There are two notations for representing Finite Automata. They are:

1. **Transition Table**
2. **Transition Diagram**

1. Transition Table

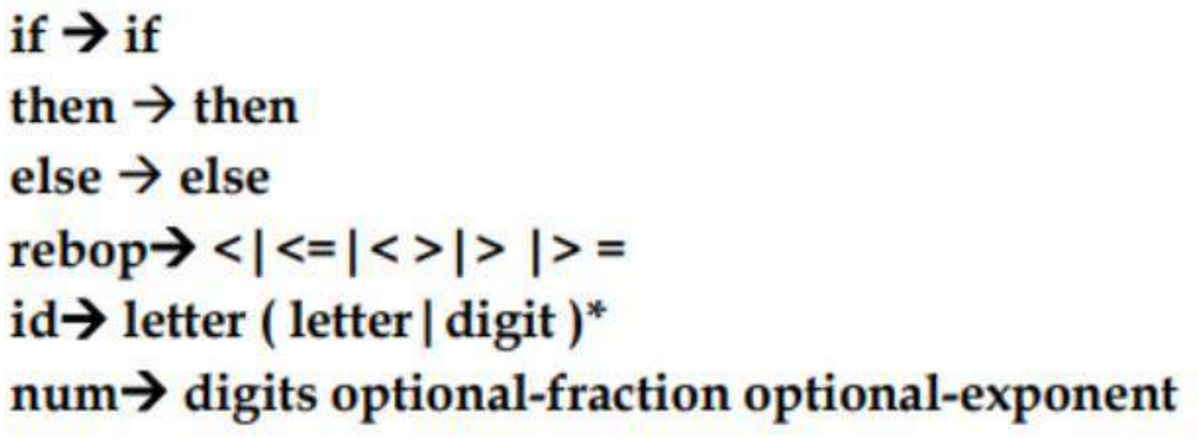
It is a tabular representation that lists all possible transitions for each state and input symbol combination.

EXAMPLE
Assume the following grammar fragment to generate a specific language



stmt → if *expr* then *stmt* | if *expr* then *stmt* else *stmt* | ε
expr → *term* relop *term* | *term*
term → id | number

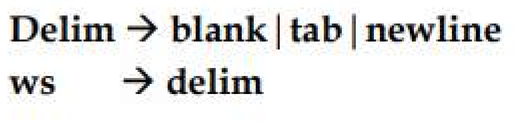
where the terminals **if**, **then**, **else**, **relop**, **id** and **num** generates sets of strings given by following regular definitions.



if → if
then → then
else → else
rebop→ < | <= | < > | > | > =
id→ letter (letter | digit)*
num→ digits optional-fraction optional-exponent

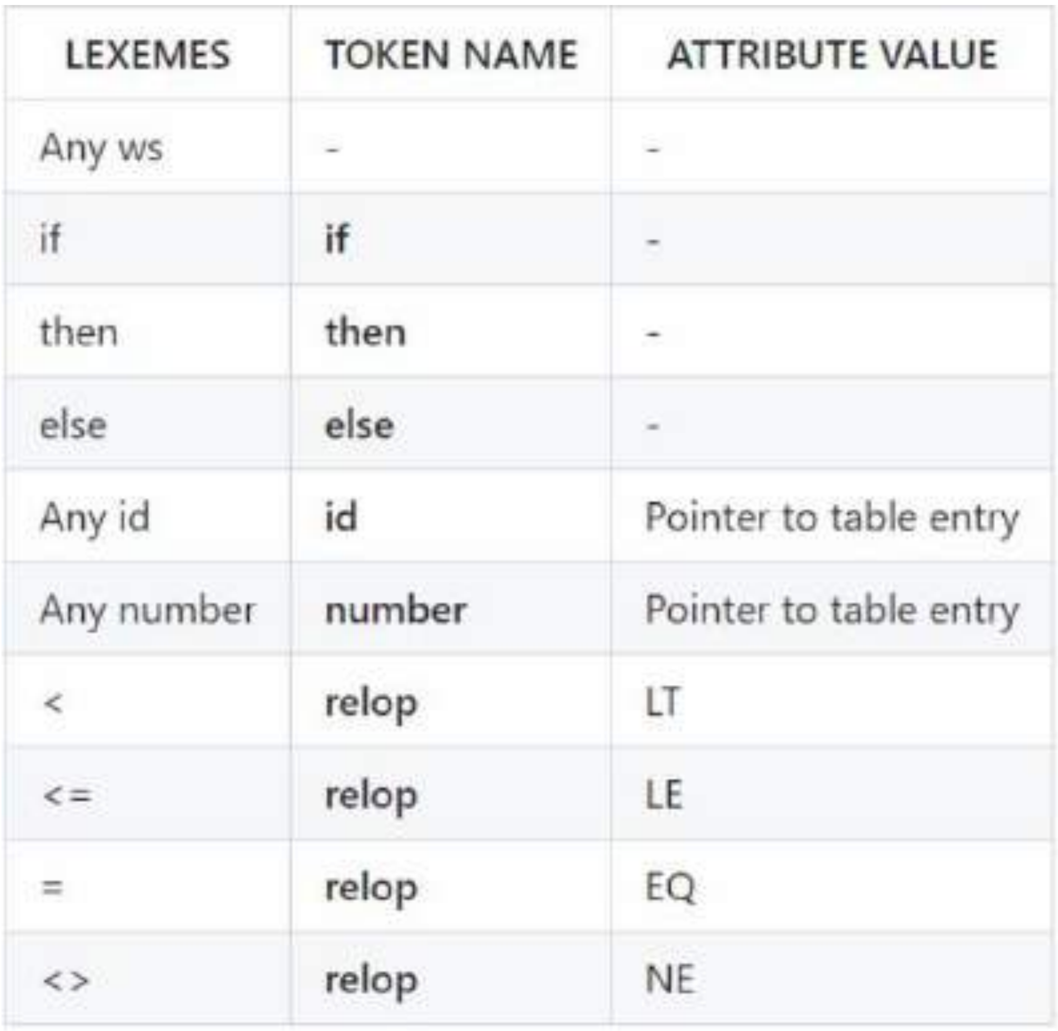
where letter and digits are defined as **-** (**letter → [A-Z a-z]** & **digit → [0-9]**)

- For this language, the lexical analyzer will recognize the keywords **if**, **then**, and **else**, as well as lexemes that match the patterns for **relop**, **id**, and **number**.
- To simplify matters, we make the common assumption that keywords are also reserved words: that is they cannot be used as identifiers.
- The num represents the unsigned integer and real numbers of Pascal.
- In addition, we assume lexemes are separated by white space, consisting of nonnull sequences of blanks, tabs, and newlines.
- Our lexical analyzer will strip out white space. It will do so by comparing a string against the regular definition **ws**, below.



Delim → blank | tab | newline
ws → delim

- If a match for **ws** is found, the lexical analyzer does not return a token to the parser.
- It is the following token that gets returned to the parser.



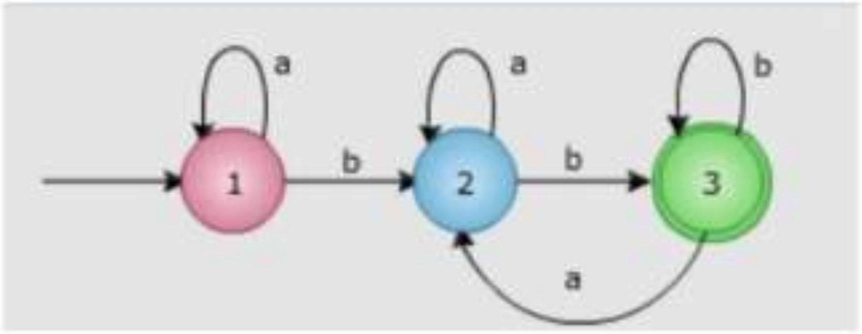
LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any ws	-	-
if	**if**	-
then	**then**	-
else	**else**	-
Any id	**id**	Pointer to table entry
Any number	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE

2. Transition Diagram

It is a directed labeled graph consisting of nodes and edges. Nodes represent states, while edges represent state transitions.

Components of Transition Diagram

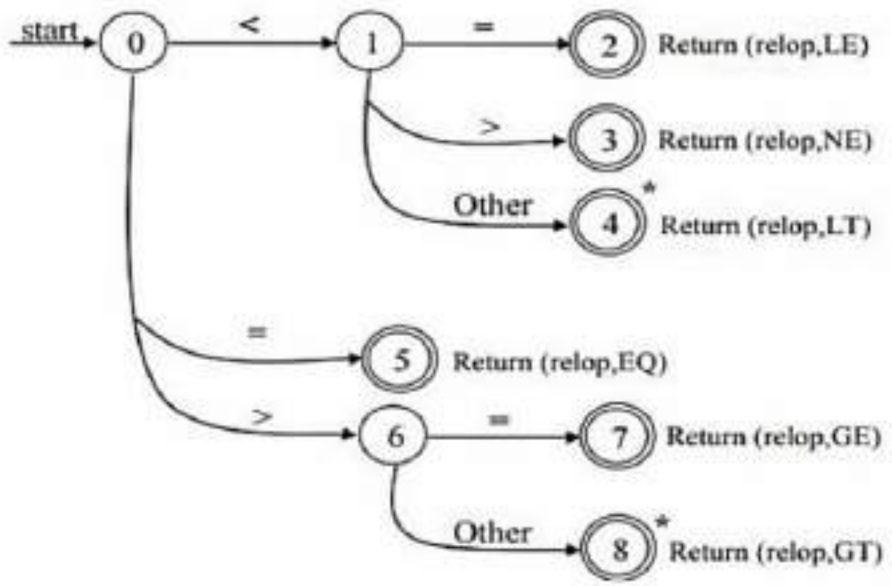
1. One state is labelled the **Start State.** It is the initial state of transition diagram where control resides when we begin to recognize a token. start→○
2. Position is a transition diagram are drawn as circles and are called states. ○
3. The states are connected by **Arrows** called edges. Labels on edges are indicating the input characters →
4. Zero or more **final** states or **Accepting** states are represented by double circle in which the tokens has been found. ◎
5. **Example:**



- Where state "1" is initial state and state 3 is final state.

Here is the transition diagram of Finite Automata that recognizes the lexemes matching the token **relop.**

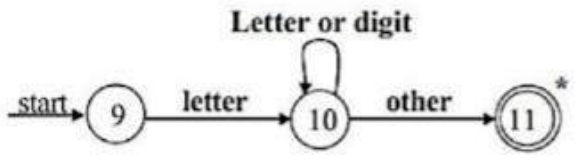
Example: A Transition Diagram for the token relation operators **"relop"** is shown in Figure below:



Here is the Finite Automata Transition Diagram for the Identifiers and Keywords.

Here is the Finite Automata Transition Diagram for the Identifiers and Keywords.

Example: A Transition Diagram for the **identifiers and keywords**



Context free grammar

Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language.

Context free grammar G can be defined by four tuples as:

1. G= (V, T, P, S)

Where,

G describes the grammar

T describes a finite set of terminal symbols.

V describes a finite set of non-terminal symbols

P describes a set of production rules

S is the start symbol.

In CFG, the start symbol is used to derive the string. You can derive the string by repeatedly replacing a non-terminal by the right hand side of the production, until all non-terminal have been replaced by terminal symbols.

Example:

L= {wcw^R | w € (a, b)*}

Production rules:

1. S → aSa
2. S → bSb
3. S → c

Now check that abbcbba string can be derived from the given CFG.

1. S ⇒ aSa
2. S ⇒ abSba
3. S ⇒ abbSbba
4. S ⇒ abbcbba

By applying the production S → aSa, S → bSb recursively and finally applying the production S → c, we get the string abbcbba.

Capabilities of CFG

There are the various capabilities of CFG:

- Context free grammar is useful to describe most of the programming languages.
- If the grammar is properly designed then an efficientparser can be constructed automatically.
- Using the features of associatively & precedence information, suitable grammars for expressions can be constructed.
- Context free grammar is capable of describing nested structures like: balanced parentheses, matching begin-end, corresponding if-then-else's & so on.

Derivation

Derivation is a sequence of production rules. It is used to get the input string through these production rules. During parsing we have to take two decisions. These are as follows:

- We have to decide the non-terminal which is to be replaced.
- We have to decide the production rule by which the non-terminal will be replaced.

We have two options to decide which non-terminal to be replaced with production rule.

Left-most Derivation

In the left most derivation, the input is scanned and replaced with the production rule from left to right. So in left most derivatives we read the input string from left to right.

Example:

Production rules

1. S = S + S
2. S = S - S
3. S = a | b |c

Input:

```
a - b + c
```

The left-most derivation is:

1. S = S + S
2. S = S - S + S
3. S = a - S + S
4. S = a - b + S
5. S = a - b + c

Right-most Derivation

In the right most derivation, the input is scanned and replaced with the production rule from right to left. So in right most derivatives we read the input string from right to left.

Example:

1. S = S + S
2. S = S - S
3. S = a | b |c

Input:

```
a - b + c
```

The right-most derivation is:

1. S = S - S
2. S = S - S + S
3. S = S - S + c
4. S = S - b + c
5. S = a - b + c

Parse tree

- Parse tree is the graphical representation of symbol. The symbol can be terminal or non- terminal.
- In parsing, the string is derived using the start symbol. The root of the parse tree is that start symbol.
- It is the graphical representation of symbol that can be terminals or non-terminals.
- Parse tree follows the precedence of operators. The deepest sub-tree traversed first. So, the operator in the parent node has less precedence over the operator in the sub-tree.

The parse tree follows these points:

- All leaf nodes have to be terminals.
- All interior nodes have to be non-terminals.
- In-order traversal gives original input string.

Example:

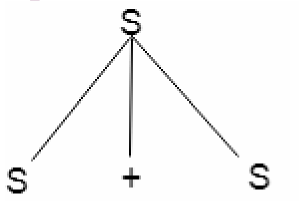
Production rules:

1. T= T + T | T * T
2. T = a|b|c

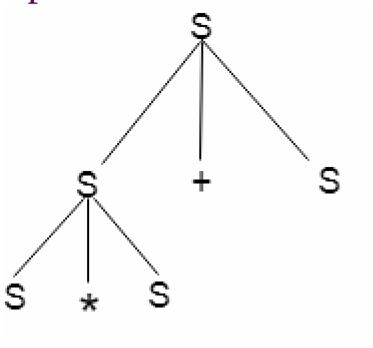
Input:

```
a * b + c
```

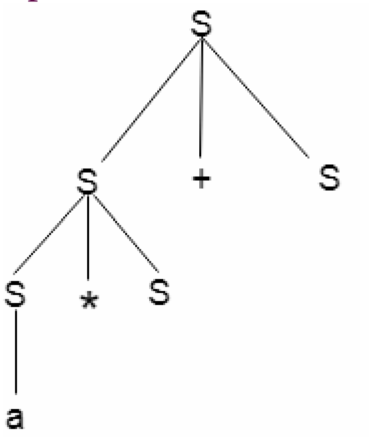
Step 1:



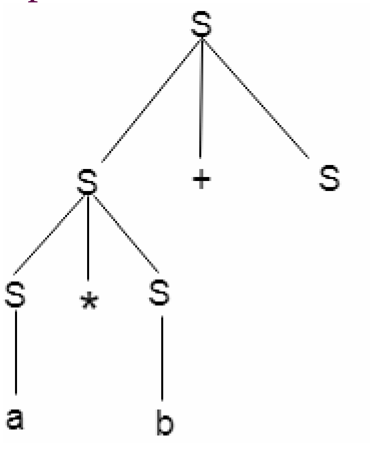
Step 2:



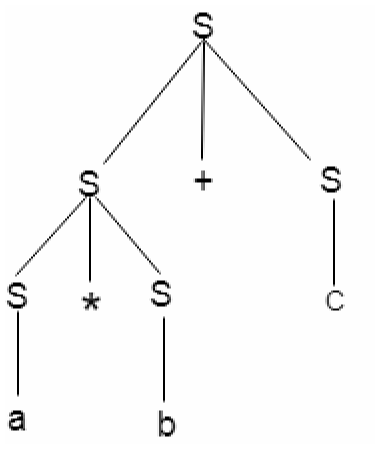
Step 3:



Step 4:



Step 5:



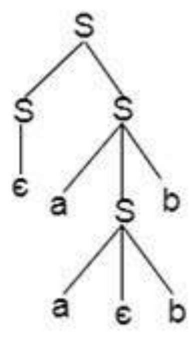
Ambiguity

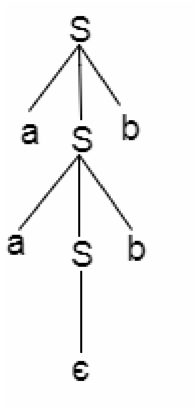
A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivative or more than one parse tree for the given input string. If the grammar is not ambiguous then it is called unambiguous.

Example:

1. S = aSb | SS
2. S = ∈

For the string aabb, the above grammar generates two parse trees:





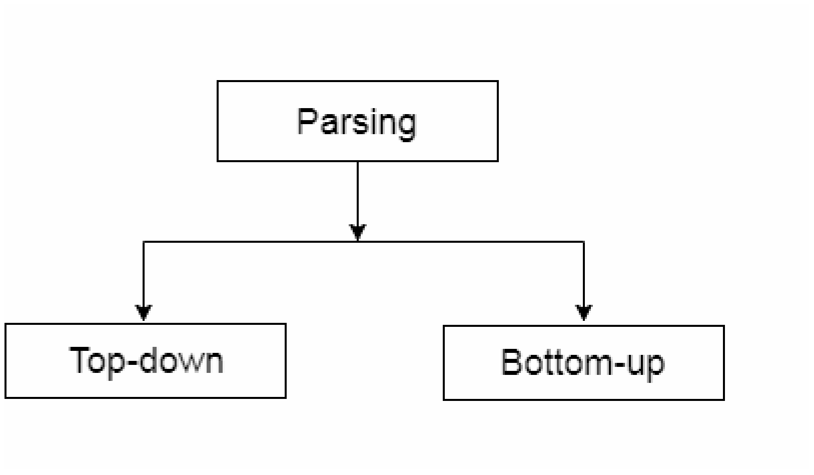
If the grammar has ambiguity then it is not good for a compiler construction. No method can automatically detect and remove the ambiguity but you can remove ambiguity by re- writing the whole grammar without ambiguity.

Parser

Parser is a compiler that is used to break the data into smaller elements coming from lexical analysis phase.

A parser takes input in the form of sequence of tokens and produces output in the form of parse tree.

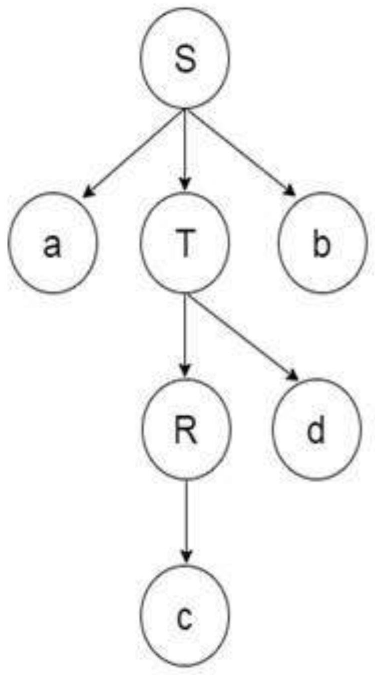
Parsing is of two types: top down parsing and bottom up parsing.



Top down paring

- The top down parsing is known as recursive parsing or predictive parsing.
- Bottom up parsing is used to construct a parse tree for an input string.
- In the top down parsing, the parsing starts from the start symbol and transform it into the input symbol.

Parse Tree representation of input string "acdb" is as follows:



Bottom up parsing

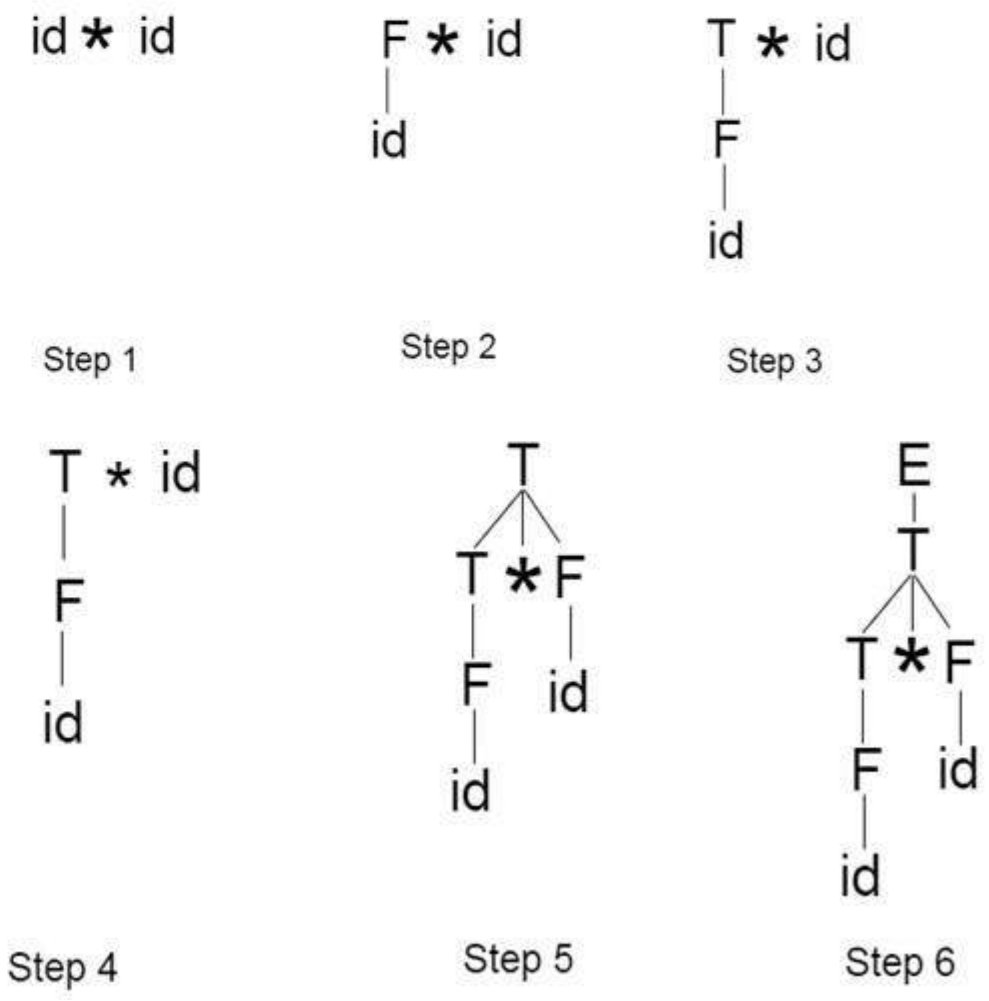
- Bottom up parsing is also known as shift-reduce parsing.
- Bottom up parsing is used to construct a parse tree for an input string.
- In the bottom up parsing, the parsing starts with the input symbol and construct the parse tree up to the start symbol by tracing out the rightmost derivations of string in reverse.

Example

Production

1. E → T
2. T → T * F
3. T → id
4. F → T
5. F → id

Parse Tree representation of input string "id * id" is as follows:



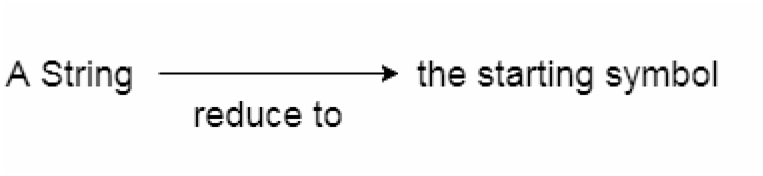
Bottom up parsing is classified in to various parsing. These are as follows:

1. Shift-Reduce Parsing
2. Operator Precedence Parsing
3. Table Driven LR Parsing

a. LR(1)
b. SLR(1)
c. CLR (1)
d. LALR(1)

Shift reduce parsing

- Shift reduce parsing is a process of reducing a string to the start symbol of a grammar.
- Shift reduce parsing uses a stack to hold the grammar and an input tape to hold the string.



- Sift reduce parsing performs the two actions: shift and reduce. That's why it is known as shift reduces parsing.
- At the shift action, the current symbol in the input string is pushed to a stack.
- At each reduction, the symbols will replaced by the non-terminals. The symbol is the right side of the production and non-terminal is the left side of the production.

Example:

Grammar:

1. S → S+S
2. S → S-S
3. S → (S)
4. S → a

Input string:

1. a1-(a2+a3)

Parsing table:

Stack contents	Input string	Actions
\$	a1-(a2+a3)\$	shift a1
\$a1	-(a2+a3)\$	reduce by $S \rightarrow a$
\$S	-(a2+a3)\$	shift -
\$S-	(a2+a3)\$	shift (
\$S-(	a2+a3)\$	shift a2
\$S-(a2	+a3)\$	reduce by $S \rightarrow a$
\$S-(S	+a3) \$	shift +
\$S-(S+	a3) \$	shift a3
\$S-(S+a3	) \$	reduce by $S \rightarrow a$
\$S-(S+S	) \$	shift)
\$S-(S+S)	\$	reduce by $S \rightarrow S+S$
\$S-(S)	\$	reduce by $S \rightarrow (S)$
\$S-S	\$	reduce by $S \rightarrow S-S$
\$S	\$	Accept

There are two main categories of shift reduce parsing as follows:

1. Operator-Precedence Parsing
2. LR-Parser

Operator precedence parsing

Operator precedence grammar is kinds of shift reduce parsing method. It is applied to a small class of operator grammars.

A grammar is said to be operator precedence grammar if it has two properties:

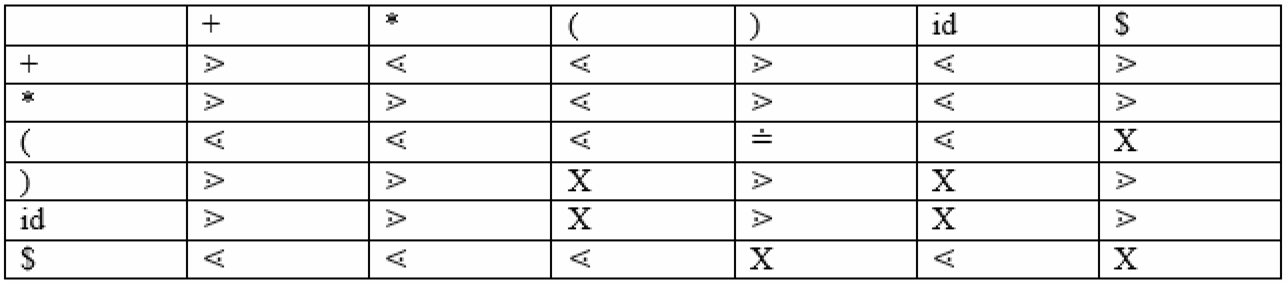
- No R.H.S. of any production has a∈.
- No two non-terminals are adjacent.

Operator precedence can only established between the terminals of the grammar. It ignores the non-terminal.

There are the three operator precedence relations:

a ⋗ b means that terminal "a" has the higher precedence than terminal "b". a ⋖ b means that terminal "a" has the lower precedence than terminal "b". a ≐ b means that the terminal "a" and "b" both have same precedence.

Precedence table:



	+	*	(	)	id	$
+	⋗	⋖	⋖	⋗	⋖	⋗
*	⋗	⋗	⋖	⋗	⋖	⋗
(	⋖	⋖	⋖	≐	⋖	X
)	⋗	⋗	X	⋗	X	⋗
id	⋗	⋗	X	⋗	X	⋗
$	⋖	⋖	⋖	X	⋖	X

Parsing Action

- Both end of the given input string, add the $ symbol.
- Now scan the input string from left right until the ⋗ is encountered.
- Scan towards left over all the equal precedence until the first left most ⋖ is encountered.
- Everything between left most ⋖ and right most ⋗ is a handle.
- $ on $ means parsing is successful.

Example

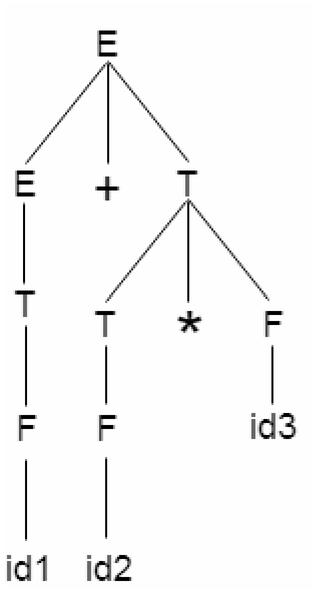
Grammar:

1. E → E+T/T
2. T → T*F/F
3. F → id

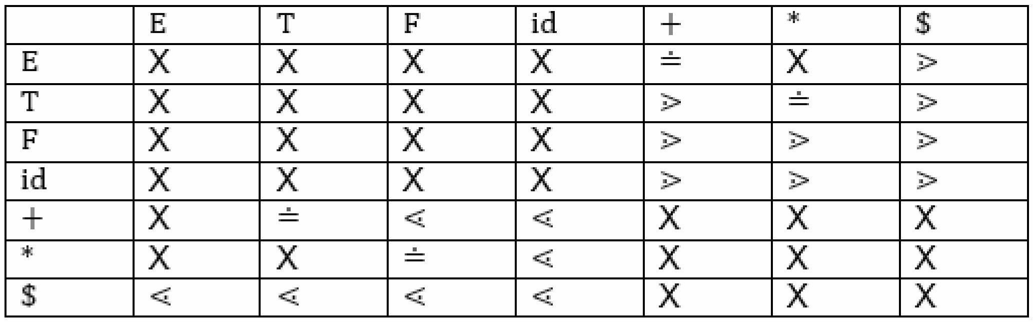
Given string:

1. w = id + id * id

   Let us consider a parse tree for it as follows:

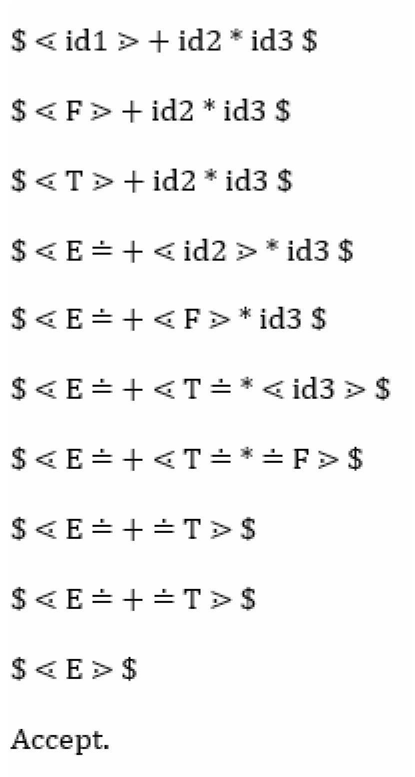


On the basis of above tree, we can design following operator precedence table:



	E	T	F	id	+	*	$
E	X	X	X	X	$\doteq$	X	$\gtrdot$
T	X	X	X	X	$\gtrdot$	$\doteq$	$\gtrdot$
F	X	X	X	X	$\gtrdot$	$\gtrdot$	$\gtrdot$
id	X	X	X	X	$\gtrdot$	$\gtrdot$	$\gtrdot$
+	X	$\doteq$	$\lessdot$	$\lessdot$	X	X	X
*	X	X	$\doteq$	$\lessdot$	X	X	X
$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	X	X	X

Now let us process the string with the help of the above precedence table:



$\$ \lessdot \text{id1} \gtrdot + \text{id2} * \text{id3}\ \$$

$\$ \lessdot F \gtrdot + \text{id2} * \text{id3}\ \$$

$\$ \lessdot T \gtrdot + \text{id2} * \text{id3}\ \$$

$\$ \lessdot E \doteq + \lessdot \text{id2} \gtrdot * \text{id3}\ \$$

$\$ \lessdot E \doteq + \lessdot F \gtrdot * \text{id3}\ \$$

$\$ \lessdot E \doteq + \lessdot T \doteq * \lessdot \text{id3} \gtrdot \$$

$\$ \lessdot E \doteq + \lessdot T \doteq * \doteq F \gtrdot \$$

$\$ \lessdot E \doteq + \doteq T \gtrdot \$$

$\$ \lessdot E \doteq + \doteq T \gtrdot \$$

$\$ \lessdot E \gtrdot \$$

Accept.

LR Parser

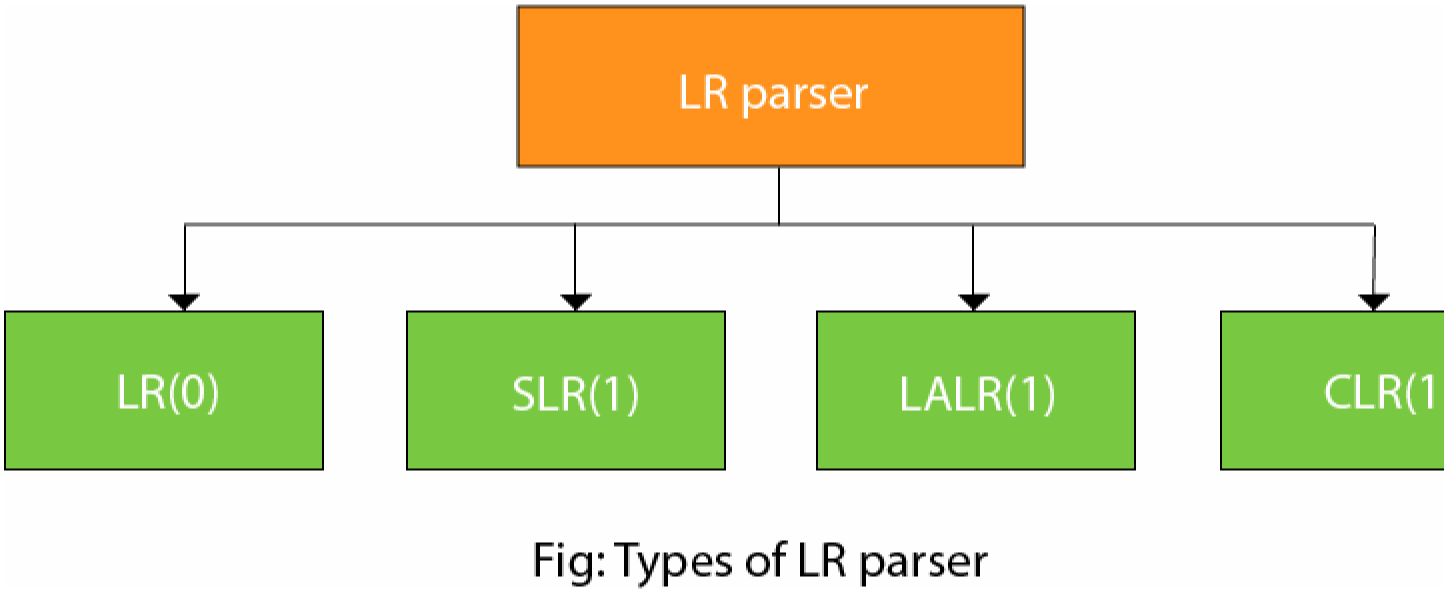
LR parsing is one type of bottom up parsing. It is used to parse the large class of grammars.

In the LR parsing, "L" stands for left-to-right scanning of the input. "R"

stands for constructing a right most derivation in reverse.

"K" is the number of input symbols of the look ahead used to make number of parsing decision.

LR parsing is divided into four parts: LR (0) parsing, SLR parsing, CLR parsing and LALR parsing.



LR algorithm:

The LR algorithm requires stack, input, output and parsing table. In all type of LR parsing, input, output and stack are same but parsing table is different.

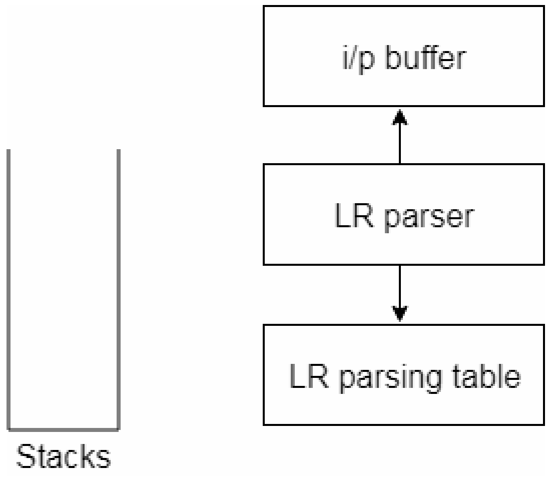


Fig: Block diagram of LR parser

Input buffer is used to indicate end of input and it contains the string to be parsed followed by a $ Symbol.

A stack is used to contain a sequence of grammar symbols with a $ at the bottom of the stack.

Parsing table is a two dimensional array. It contains two parts: Action part and Go To part.

LR (1) Parsing

Various steps involved in the LR (1) Parsing:

- For the given input string write a context free grammar.
- Check the ambiguity of the grammar.
- Add Augment production in the given grammar.
- Create Canonical collection of LR (0) items.
- Draw a data flow diagram (DFA).
- Construct a LR (1) parsing table.

Augment Grammar

Augmented grammar G` will be generated if we add one more production in the given grammar G. It helps the parser to identify when to stop the parsing and announce the acceptance of the input.

Example

Given grammar

1. S → AA
2. A → aA | b

The Augment grammar G` is represented by

1. S`→ S
2. S → AA
3. A → aA | b

Canonical Collection of LR(0) items

An LR (0) item is a production G with dot at some position on the right side of the production.

LR(0) items is useful to indicate that how much of the input has been scanned up to a given point in the process of parsing.

In the LR (0), we place the reduce node in the entire row.

Example

Given gramm

S → AA

1. A → aA | b

Add Augment Production and insert '•' symbol at the first position for every production in

1. S` → •S
2. S → •AA
3. A → •aA
4. A → •b

I0 State:

Add Augment production to the I0 State and Compute the Closure

I0 = Closure (S` → •S)

Add all productions starting with S in to I0 State because "•" is followed by the non-terminal. So, the I0 State becomes

I0 = S` → •S
S → •AA

Add all productions starting with "A" in modified I0 State because "•" is followed by the non-terminal. So, the I0 State becomes.

I0= S` → •S
S → •AA
A → •aA
A → •b

I1= Go to (I0, S) = closure (S` → S•) = S` → S• Here,

the Production is reduced so close the State. **I1=** S` → S•

I2= Go to (I0, A) = closure (S → A•A)

Add all productions starting with A in to I2 State because "•" is followed by the non-terminal. So, the I2 State becomes

I2 =S→A•A
A → •aA
A → •b

Go to (I2,a) = Closure (A → a•A) = (same as I3) Go to

(I2, b) = Closure (A → b•) = (same as I4) **I3=** Go to

(I0,a) = Closure (A → a•A)

Add productions starting with A in I3.

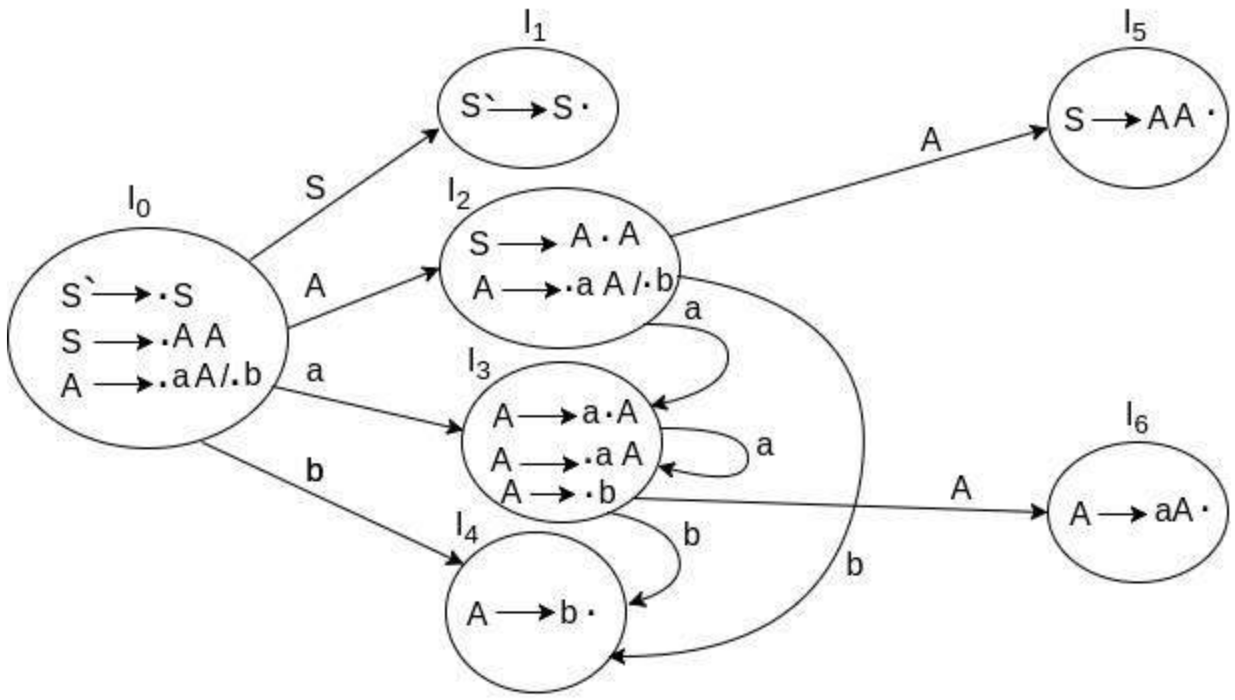
A → a•A
A → •aA
A → •b

Go to (I3, a) = Closure (A → a•A) = (same as I3) Go to (I3, b) = Closure (A → b•) = (same as I4)

I4= Go to (I0, b) = closure (A → b•) = A → b•
I5= Go to (I2, A) = Closure (S → AA•) = SA → A•
I6= Go to (I3, A) = Closure (A → aA•) = A → aA•

Drawing DFA:

The DFA contains the 7 states I0 to I6.



LR(0) Table

- If a state is going to some other state on a terminal then it correspond to a shift move.
- If a state is going to some other state on a variable then it correspond to go to move.
- If a state contain the final item in the particular row then write the reduce node completely.

States	Action			Go to	
	a	**b**	**$**	**A**	**S**
I_0	S3	S4		2	1
I_1			accept		
I_2	S3	S4		5	
I_3	S3	S4		6	
I_4	r3	r3	r3		
I_5	r1	r1	r1		
I_6	r2	r2	r2		

Explanation:

- I0 on S is going to I1 so write it as 1.
- I0 on A is going to I2 so write it as 2.

- I2 on A is going to I5 so write it as 5.
- I3 on A is going to I6 so write it as 6.
- I0, I2and I3on a are going to I3 so write it as S3 which means that shift 3.
- I0, I2 and I3 on b are going to I4 so write it as S4 which means that shift 4.
- I4, I5 and I6 all states contains the final item because they contain • in the right most end. So rate the production as production number.

Productions are numbered as follows:

1. S → AA ... (1)
2. A → aA ... (2)
3. A → b ... (3)

- I1 contains the final item which drives(S` → S•), so action {I1, $} = Accept.
- I4 contains the final item which drives A → b• and that production corresponds to the production number 3 so write it as r3 in the entire row.
- I5 contains the final item which drives S → AA• and that production corresponds to the production number 1 so write it as r1 in the entire row.
- I6 contains the final item which drives A → aA• and that production corresponds to the production number 2 so write it as r2 in the entire row.

SLR (1) Parsing

SLR (1) refers to simple LR Parsing. It is same as LR(0) parsing. The only difference is in the parsing table.To construct SLR (1) parsing table, we use canonical collection of LR (0) item.

In the SLR (1) parsing, we place the reduce move only in the follow of left hand side. Various

steps involved in the SLR (1) Parsing:

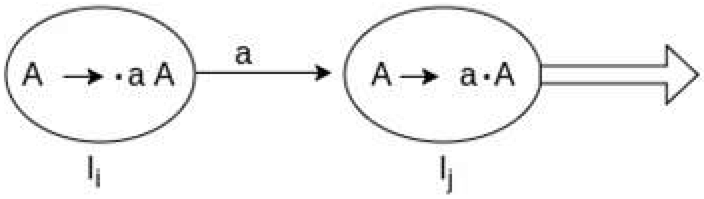
- For the given input string write a context free grammar
- Check the ambiguity of the grammar
- Add Augment production in the given grammar
- Create Canonical collection of LR (0) items
- Draw a data flow diagram (DFA)

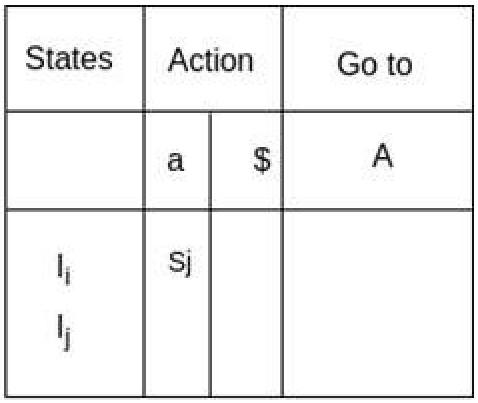
- Construct a SLR (1) parsing table

SLR (1) Table Construction

The steps which use to construct SLR (1) Table is given below:

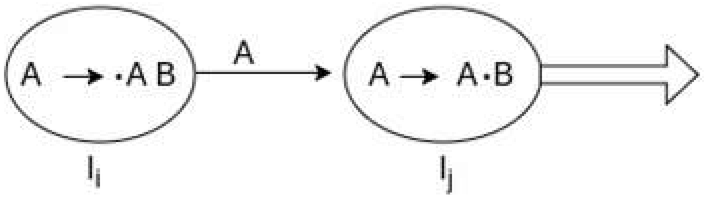
If a state (I_i) is going to some other state (I_j) on a terminal then it corresponds to a shift move in the action part.

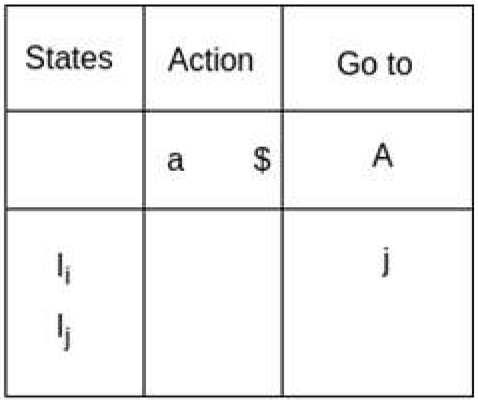




States	Action		Go to
	a	$	A
I_i	Sj		
I_j			

If a state (I_i) is going to some other state (I_j) on a variable then it correspond to go to move in the Go to part.





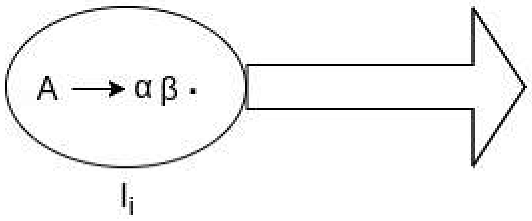
States	Action		Go to
	a	$	A
I_i			j
I_j			

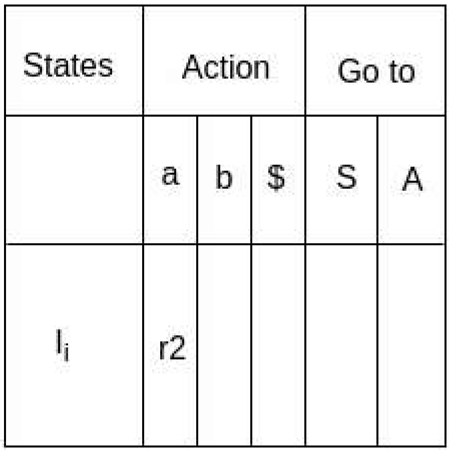
If a state (I_i) contains the final item like A → ab• which has no transitions to the next state then the production is known as reduce production. For all terminals X in FOLLOW (A), write the reduce entry along with their production numbers.

Example

1. S -> •Aa
2. A->αβ•
1. Follow(S) = {$}

2. Follow (A) = {a}





States	Action			Go to	
	a	b	$	S	A
I_i	r2				

SLR (1) Grammar

S → E
E → E + T | T
T → T * F | F
F → id

Add Augment Production and insert '•' symbol at the first position for every production in G

S` → •E
E → •E + T
E → •T
T → •T * F
T → •F
F → •id

I0 State:

Add Augment production to the I0 State and Compute the Closure

I0 = Closure (S` → •E)

Add all productions starting with E in to I0 State because "." is followed by the non-terminal. So, the I0 State becomes

I0 = S` → •E
E → •E + T
E → •T

Add all productions starting with T and F in modified I0 State because "." is followed by the non-terminal. So, the I0 State becomes.

I0= S` → •E
E → •E + T
E → •T
T → •T * F
T → •F
F → •id

I1= Go to (I0, E) = closure (S` → E•, E → E• + T) **I2=** Go to (I0, T) = closure (E → T•T, T• → * F) **I3=** Go to (I0, F) = Closure (T → F•) = T → F• **I4=** Go to (I0, id) = closure (F → id•) = F → id• **I5=** Go to (I1, +) = Closure (E → E +•T)

Add all productions starting with T and F in I5 State because "." is followed by the non-terminal. So, the I5 State becomes

I5 = E → E +•T
T → •T * F
T → •F
F → •id

Go to (I5, F) = Closure (T → F•) = (same as I3) Go to (I5, id) = Closure (F → id•) = (same as I4)

I6= Go to (I2, *) = Closure (T → T * •F)

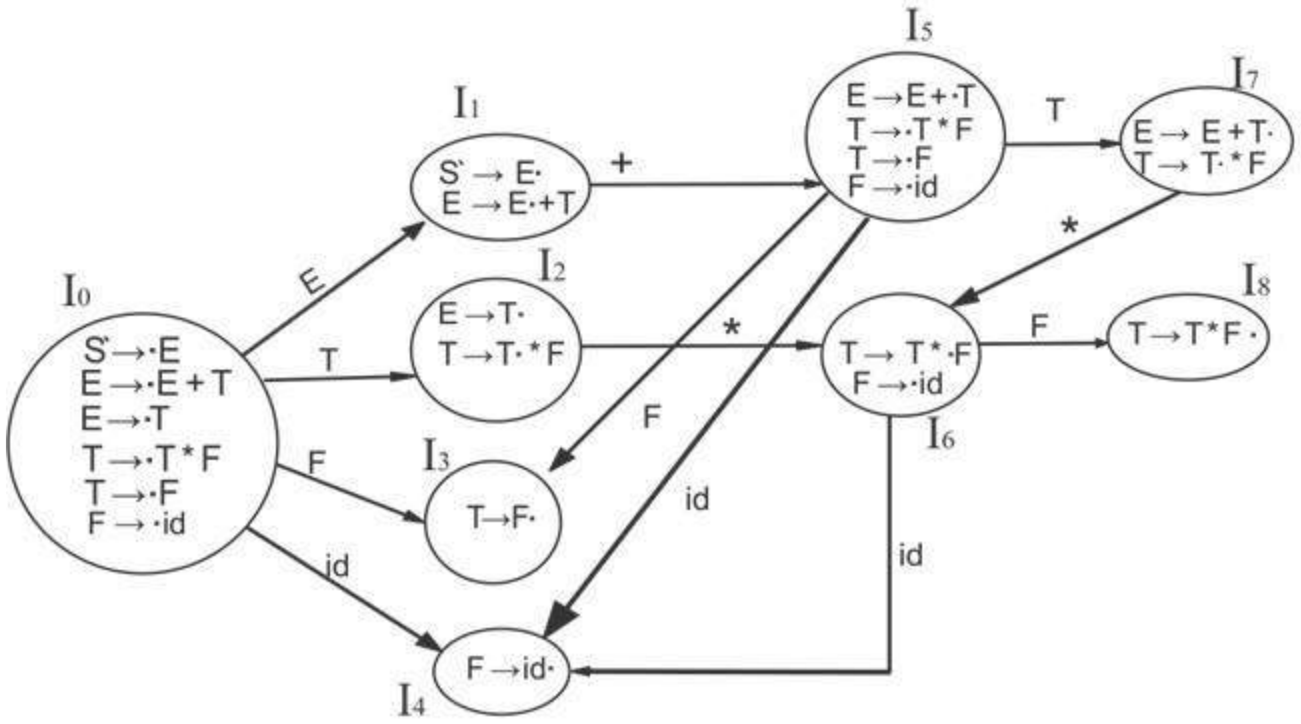
Add all productions starting with F in I6 State because "." is followed by the non-terminal. So, the I6 State becomes

I6 = T → T * •F
F → •id

Go to (I6, id) = Closure (F → id•) = (same as I4)

I7= Go to (I5, T) = Closure (E → E + T•) = E → E + T•
I8= Go to (I6, F) = Closure (T → T * F•) = T → T * F•

Drawing DFA:



SLR (1) Table

States	Action				Go to		
	id	**+**	*	**$**	**E**	**T**	**F**
I_0	S_4				1	2	3
I_1		S_5		Accept			
I_2		R_2	S_6	R2			
I_3		R_4	R_4	R4			
I_4		R_5	R5	R5			
I_5	S4					7	3
I_6	S4						8
I_7		R1	S6	R1			
I_8		R3	R3	R3			

Explanation:

First (E) = First (E + T) ∪ First
(T) First (T) = First (T * F) ∪
First (F) First (F) = {id}
First (T) = {id}
First (E) = {id}
Follow (E) = First (+T) ∪ {$} = {+, $}
Follow (T) = First (*F) ∪ First (F)

= {*, +, $}
Follow (F) = {*, +, $}

- I1 contains the final item which drives S → E• and follow (S) = {$}, so action {I1, $} = Accept
- I2 contains the final item which drives E → T• and follow (E) = {+, $}, so action {I2, +} = R2, action {I2, $} = R2
- I3 contains the final item which drives T → F• and follow (T) = {+, *, $}, so action {I3, +} = R4, action {I3, *} = R4, action {I3, $} = R4
- I4 contains the final item which drives F → id• and follow (F) = {+, *, $}, so action {I4, +} = R5, action {I4, *} = R5, action {I4, $} = R5
- I7 contains the final item which drives E → E + T• and follow (E) = {+, $}, so action {I7, +} = R1, action {I7, $} = R1
- I8 contains the final item which drives T → T * F• and follow (T) = {+, *, $}, so action {I8, +} = R3, action {I8, *} = R3, action {I8, $} = R3.

CLR (1) Parsing

CLR refers to canonical lookahead. CLR parsing use the canonical collection of LR (1) items to build the CLR (1) parsing table. CLR (1) parsing table produces the more number of states as compare to the SLR (1) parsing.

In the CLR (1), we place the reduce node only in the lookahead symbols.

Various steps involved in the CLR (1) Parsing:

- For the given input string write a context free grammar
- Check the ambiguity of the grammar
- Add Augment production in the given grammar
- Create Canonical collection of LR (0) items
- Draw a data flow diagram (DFA)
- Construct a CLR (1) parsing table

LR (1) item

LR (1) item is a collection of LR (0) items and a look ahead symbol.

LR (1) item = LR (0) item + look ahead

The look ahead is used to determine that where we place the final item. The look

ahead always add $ symbol for the argument production.

Example

CLR (1) Grammar

1. S → AA
2. A → aA
3. A → b

Add Augment Production, insert '•' symbol at the first position for every production in G and also add the lookahead.

1. S\` → •S, $
2. S → •AA, $
3. A → •aA, a/b
4. A → •b, a/b

I0 State:

Add Augment production to the I0 State and Compute the Closure

I0 = Closure (S\` → •S)

Add all productions starting with S in to I0 State because "." is followed by the non-terminal. So, the I0 State becomes

I0 = S\` → •S, $
S → •AA, $

Add all productions starting with A in modified I0 State because "." is followed by the non-terminal. So, the I0 State becomes.

I0= S\` → •S, $
S → •AA, $
A → •aA, a/b
A → •b, a/b

I1= Go to (I0, S) = closure (S\` → S•, $) = S\` → S•, $
I2= Go to (I0, A) = closure (S → A•A, $)

Add all productions starting with A in I2 State because "." is followed by the non-terminal. So, the I2 State becomes

I2= S → A•A, $
A → •aA, $
A → •b, $

I3= Go to (I0, a) = Closure (A → a•A, a/b)

Add all productions starting with A in I3 State because "." is followed by the non-terminal. So, the I3 State becomes

I3= A → a•A, a/b
A → •aA, a/b
A → •b, a/b

Go to (I3, a) = Closure (A → a•A, a/b) = (same as I3) Go to
(I3, b) = Closure (A → b•, a/b) = (same as I4)

I4= Go to (I0, b) = closure (A → b•, a/b) = A → b•, a/b **I5=**
Go to (I2, A) = Closure (S → AA•, $) =S → AA•, $ **I6=** Go
to (I2, a) = Closure (A → a•A, $)

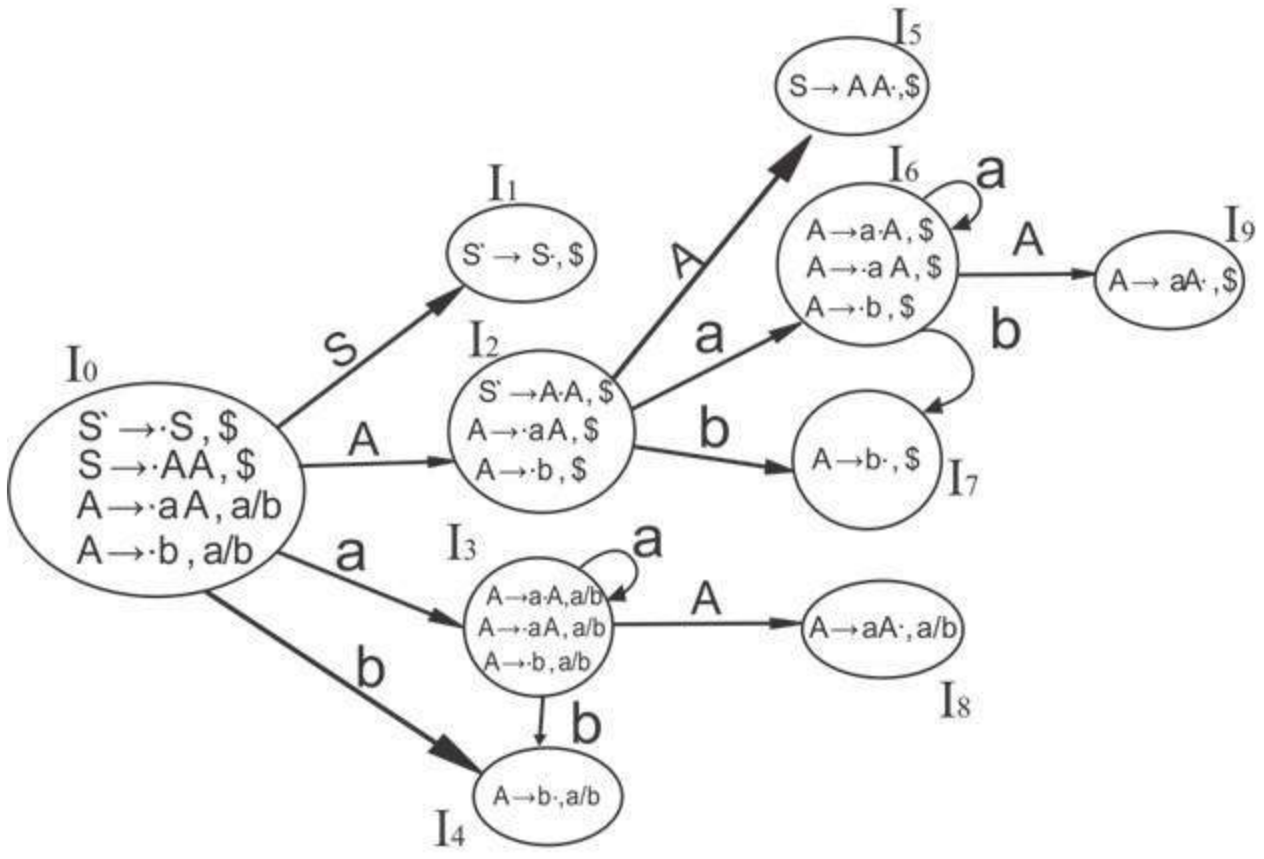
Add all productions starting with A in I6 State because "." is followed by the non-terminal. So, the I6 State becomes

I6 = A → a•A, $
A → •aA, $
A → •b, $

Go to (I6, a) = Closure (A → a•A, $) = (same as I6) Go to
(I6, b) = Closure (A → b•, $) = (same as I7)

I7= Go to (I2, b) = Closure (A → b•, $) = A → b•, $
I8= Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b
I9= Go to (I6, A) = Closure (A → aA•, $) = A → aA•, $

Drawing DFA:



CLR (1) Parsing table:

States	a	b	$	S	A
I_0	S_3	S_4			2
I_1			Accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	R_3	R_3			
I_5			R_1		
I_6	S_6	S_7			9
I_7			R_3		
I_8	R_2	R_2			
I_9			R_2		

Productions are numbered as follows:

1. S → AA ... (1)
2. A → aA............(2)

3. A → b ... (3)

The placement of shift node in CLR (1) parsing table is same as the SLR (1) parsing table. Only difference in the placement of reduce node.

I4 contains the final item which drives (A → b•, a/b), so action {I4, a} = R3, action {I4, b} = R3.
I5 contains the final item which drives (S → AA•, $), so action {I5, $} = R1. I7 contains the final item which drives (A → b•,$), so action {I7, $} = R3.
I8 contains the final item which drives (A → aA•, a/b), so action {I8, a} = R2, action {I8, b} = R2.
I9 contains the final item which drives (A → aA•, $), so action {I9, $} = R2.

LALR (1) Parsing:

LALR refers to the lookahead LR. To construct the LALR (1) parsing table, we use the canonical collection of LR (1) items.

In the LALR (1) parsing, the LR (1) items which have same productions but different look ahead are combined to form a single set of items

LALR (1) parsing is same as the CLR (1) parsing, only difference in the parsing table.

Example

LALR (1) Grammar

1. S → AA
2. A → aA
3. A → b

Add Augment Production, insert '•' symbol at the first position for every production in G and also add the look ah

1. S` → •S, $
2. S → •AA, $
3. A → •aA, a/b

4. A → •b, a/b

I0 State:

Add Augment production to the I0 State and Compute the ClosureL

I0 = Closure (S` → •S)

Add all productions starting with S in to I0 State because "•" is followed by the non-terminal. So, the I0 State becomes

I0 = S` → •S, $
S → •AA, $

Add all productions starting with A in modified I0 State because "•" is followed by the non-terminal. So, the I0 State becomes.

I0= S` → •S, $
S → •AA, $
A → •aA, a/b
A → •b, a/b

I1= Go to (I0, S) = closure (S` → S•, $) = S` → S•, $
I2= Go to (I0, A) = closure (S → A•A, $)

Add all productions starting with A in I2 State because "•" is followed by the non-terminal. So, the I2 State becomes

I2= S → A•A, $
A → •aA, $
A → •b, $

I3= Go to (I0, a) = Closure (A → a•A, a/b)

Add all productions starting with A in I3 State because "•" is followed by the non-terminal. So, the I3 State becomes

I3= A → a•A, a/b
A → •aA, a/b
A → •b, a/b

Go to (I3, a) = Closure (A → a•A, a/b) = (same as I3) Go to
(I3, b) = Closure (A → b•, a/b) = (same as I4)

I4= Go to (I0, b) = closure (A → b•, a/b) = A → b•, a/b
I5= Go to (I2, A) = Closure (S → AA•, $) =S → AA•, $

I6= Go to (I2, a) = Closure (A → a•A, $)

Add all productions starting with A in I6 State because "•" is followed by the non-terminal. So, the I6 State becomes

I6 = A → a•A, $
A → •aA, $
A → •b, $

Go to (I6, a) = Closure (A → a•A, $) = (same as I6) Go to
(I6, b) = Closure (A → b•, $) = (same as I7)

I7= Go to (I2, b) = Closure (A → b•, $) = A → b•, $
I8= Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b
I9= Go to (I6, A) = Closure (A → aA•, $) A → aA•, $

If we analyze then LR (0) items of I3 and I6 are same but they differ only in their lookahead.

I3 = { A → a•A, a/b
A → •aA, a/b
A → •b, a/b
}

I6= { A → a•A, $
A → •aA, $
A → •b, $
}

Clearly I3 and I6 are same in their LR (0) items but differ in their lookahead, so we can combine them and called as I36.

I36 = { A → a•A, a/b/$
A → •aA, a/b/$
A → •b, a/b/$
}

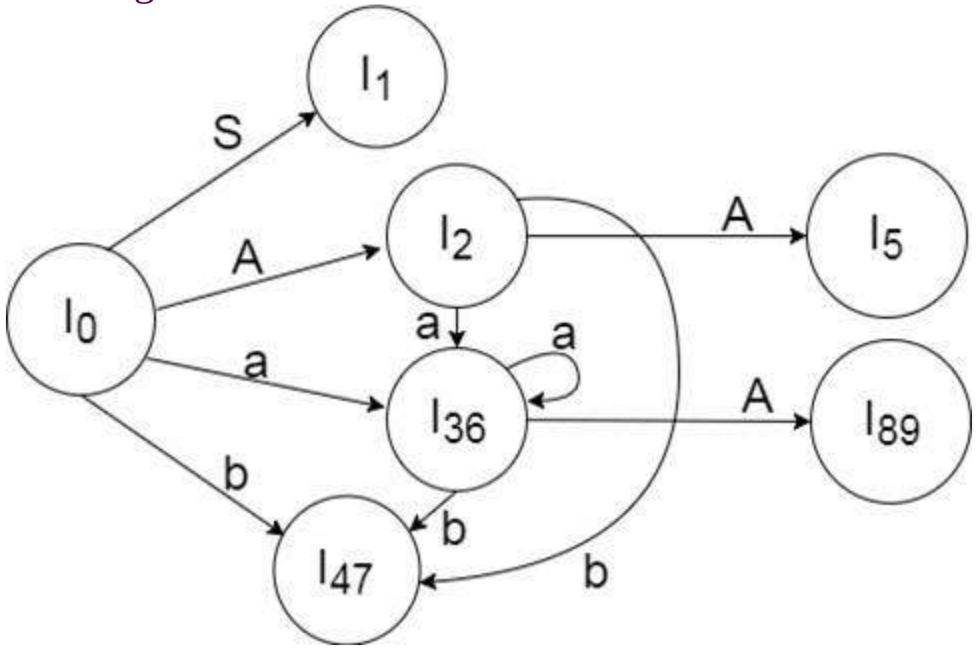
The I4 and I7 are same but they differ only in their look ahead, so we can combine them and called as I47.

I47 = {A → b•, a/b/$}

The I8 and I9 are same but they differ only in their look ahead, so we can combine them and called as I89.

I89 = {A → aA•, a/b/$}

Drawing DFA:



LALR (1) Parsing table:

States	a	b	$	S	A
I_0	S_{36}	S_{47}		12	
I_1		accept			
I_2	S_{36}	S_{47}			5
I_{36}	$S_{36}S_{47}$				89
I_{47}	R_3R_3	R_3			
I_5			R_1		
I_{89}	R_2	R_2	R_2		

o

o

o

o

o

o

o

o

o

-
-

UNIT-V

Syntax directed translation

In syntax directed translation, along with the grammar we associate some informal notations and these notations are called as semantic rules.

So we can say that

Grammar + semantic rule = SDT (syntax directed translation)

- In syntax directed translation, every non-terminal can get one or more than one attribute or sometimes 0 attribute depending on the type of the attribute. The value of these attributes is evaluated by the semantic rules associated with the production rule.
- In the semantic rule, attribute is VAL and an attribute may hold anything like a string, a number, a memory location and a complex record
- In Syntax directed translation, whenever a construct encounters in the programming language then it is translated according to the semantic rules define in that particular programming language.

Example

Production	**Semantic Rules**
E → E + T	E.val := E.val + T.val
E → T	E.val := T.val
T → T * F	T.val := T.val + F.val
T → F	T.val := F.val
F → (F)	F.val := F.val

F → num	F.val := num.lexval

E.val is one of the attributes of E.

num.lexval is the attribute returned by the lexical analyzer.

Syntax directed translation scheme

- The Syntax directed translation scheme is a context -free grammar.
- The syntax directed translation scheme is used to evaluate the order of semantic rules.
- In translation scheme, the semantic rules are embedded within the right side of the productions.
- The position at which an action is to be executed is shown by enclosed between braces. It is written within the right side of the production.

Example

Production	**Semantic Rules**
S → E $	{ printE.VAL }
E → E + E	{E.VAL := E.VAL + E.VAL }
E → E * E	{E.VAL := E.VAL * E.VAL }
E → (E)	{E.VAL := E.VAL }
E → I	{E.VAL := I.VAL }
I → I digit	{I.VAL := 10 * I.VAL + LEXVAL }
I → digit	{ I.VAL:= LEXVAL}

Implementation of Syntax directed translation

Syntax direct translation is implemented by constructing a parse tree and performing the actions in a left to right depth first order.

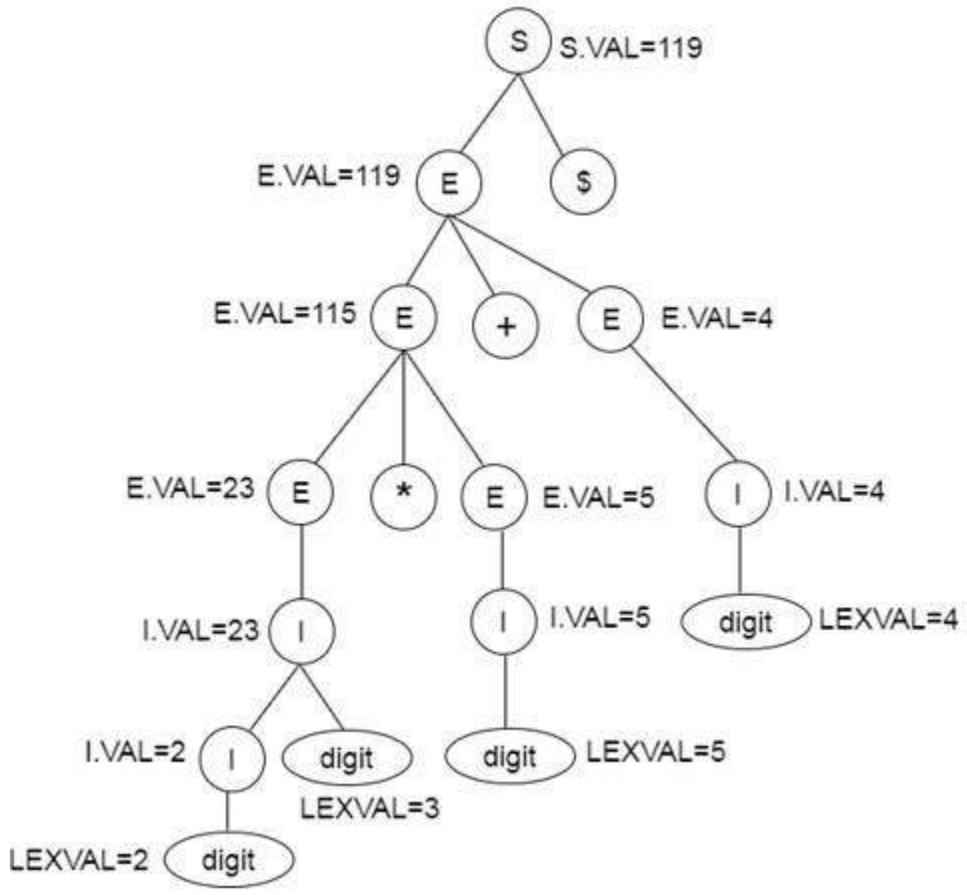
SDT is implementing by parse the input and produce a parse tree as a result.

Example

Production	**Semantic Rules**
S → E $	{ printE.VAL }

E ⟶ E + E	{E.VAL := E.VAL + E.VAL }
E ⟶ E * E	{E.VAL := E.VAL * E.VAL }
E ⟶ (E)	{E.VAL := E.VAL }
E ⟶ I	{E.VAL := I.VAL }
I ⟶ I digit	{I.VAL := 10 * I.VAL + LEXVAL }
I ⟶ digit	{ I.VAL:= LEXVAL}

Parse tree for SDT:



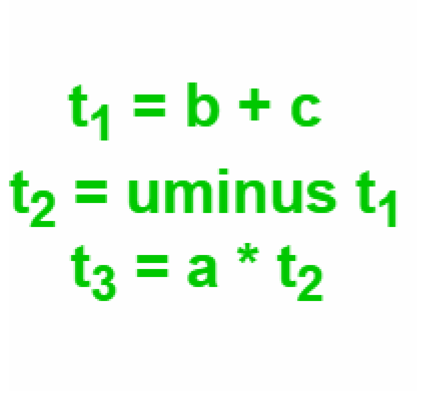
Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code.It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler. The compiler decides the order of operation given by three address code.

General representation –

a = b op c

Where a, b or c represents operands like names, constants or compiler generated temporaries and op represents the operator

Example-1: Convert the expression a * – (b + c) into three address code.



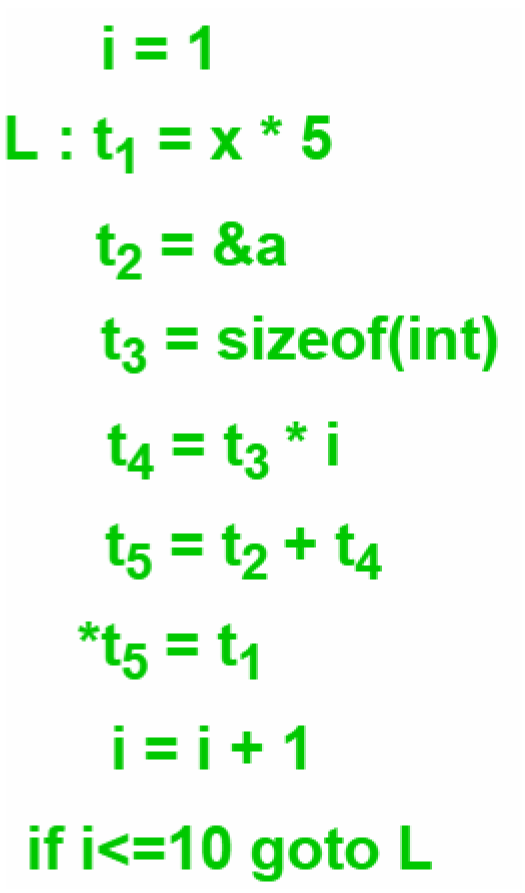
$t_1 = b + c$

$t_2 = \text{uminus } t_1$

$t_3 = a * t_2$

Example-2: Write three address code for following code

```
for(i = 1; i<=10; i++)
  {
    a[i] = x * 5;
  }
```



```
     i = 1
L : t1 = x * 5
     t2 = &a
     t3 = sizeof(int)
     t4 = t3 * i
     t5 = t2 + t4
    *t5 = t1
     i = i + 1
 if i<=10 goto L
```

Implementation of Three Address Code –
There are 3 representations of three address code namely
1. Quadruple
2. Triples
3. Indirect Triples

1. Quadruple –
It is structure with consist of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Advantage –
- Easy to rearrange code for global optimization.
- One can quickly access value of temporary variables using symbol table.

Disadvantage –
- Contain lot of temporaries.
- Temporary variable creation increases time and space complexity.

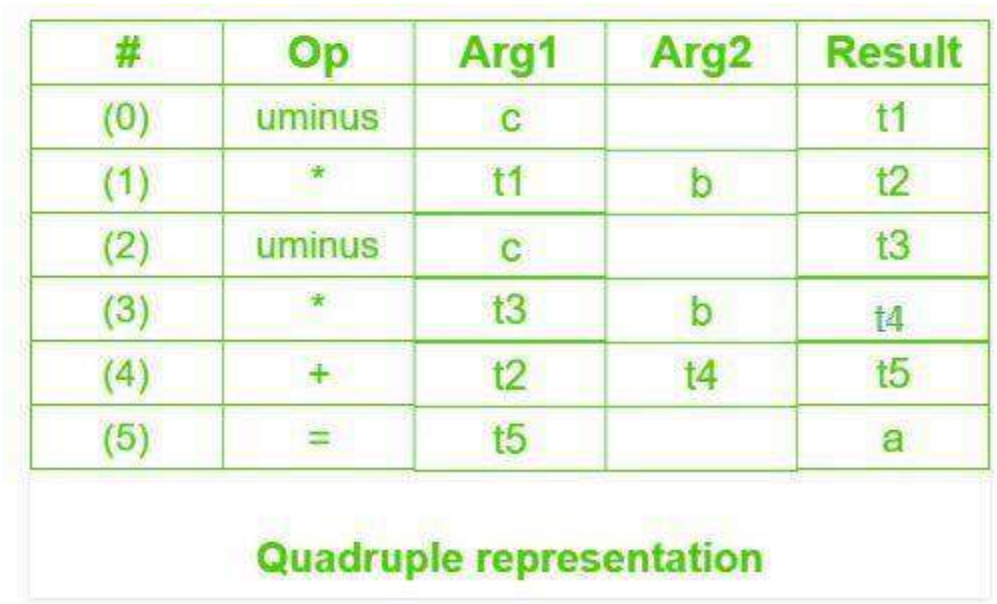
Example – Consider expression a = b * – c + b * – c. The three address code is:
t1 = uminus c t2 = b

* t1

t3 = uminus c t4

= b * t3 t5 = t2 +

t4 a = t5



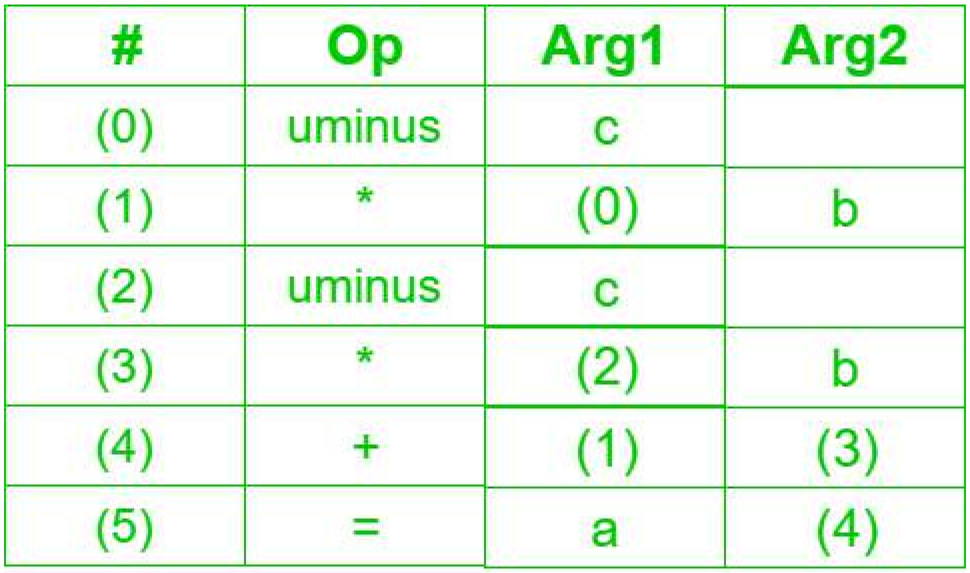
#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

Quadruple representation

2. Triples –
This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.
Disadvantage –

- Temporaries are implicit and difficult to rearrange code.
- It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

Example – Consider expression a = b * – c + b * – c

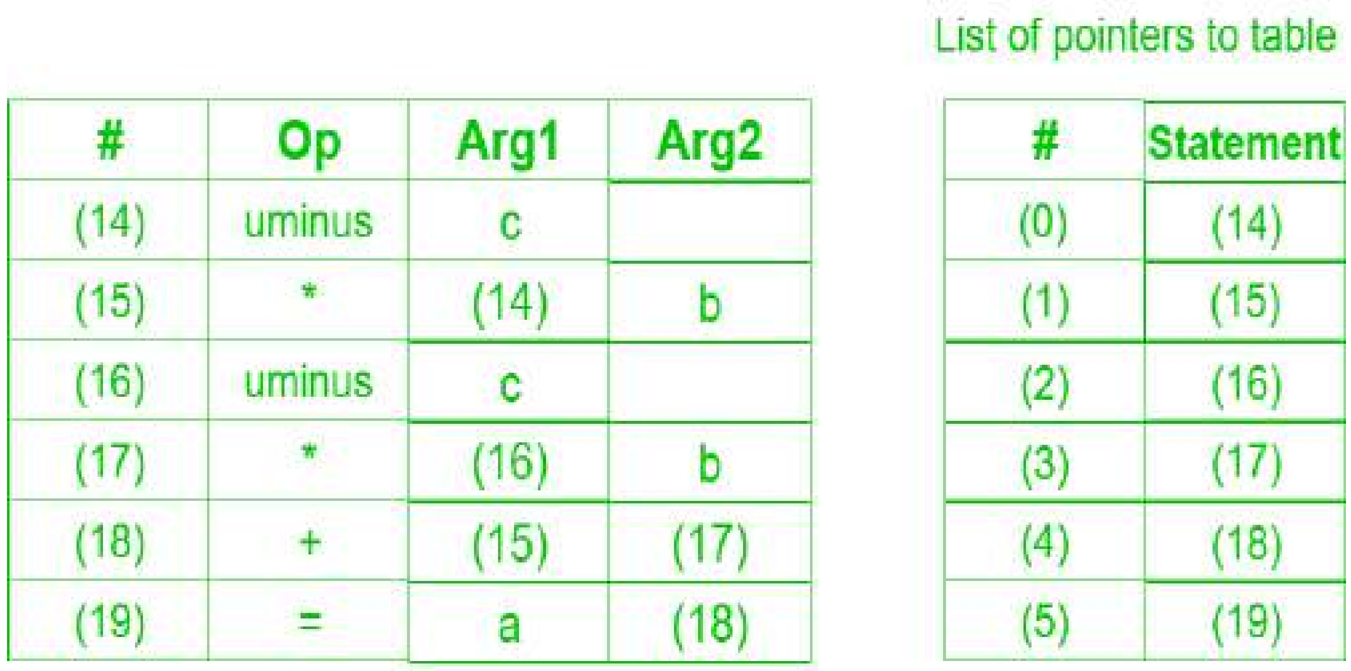


#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation

3. Indirect Triples –
This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it.
Temporaries are implicit and easier to rearrange code.

Example – Consider expression a = b * – c + b * – c



#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

List of pointers to table

#	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Indirect Triples representation

STACK ALLOCATION

The stack allocation is a runtime storage management technique. The activation records are pushed and popped as activations begin and end respectively.
Storage for the locals in each call of the procedure is contained in the activation record for that call. Thus, locals are bound to fresh storage in each activation, because a new activation record is pushed onto the stack when the call is made. It can be determined the size of the variables at a run time & hence local variables can have different storage locations & different values during various activations. Suppose that the registered top marks the top of the stack. At runtime, an activation record can be allocated and deallocated by incrementing and decrementing top, respectively, by the size of the record.
If the procedure q has an activation record of size a then the top is incremented by before the target code of q is executed. When the control returns from q, the top of the stack are decremented by a.

The memory organization for the C program on the UNIX platform is as follows
–

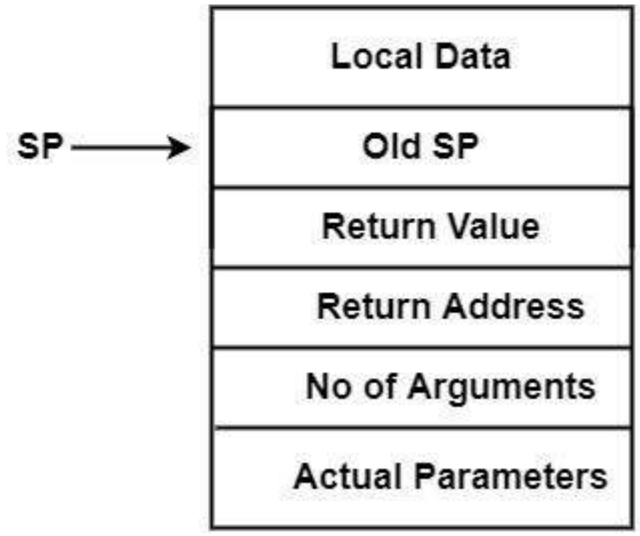
In C, data can be global, meaning it is allocated static storage and available to any procedure, or local, meaning it can be accessed only by the procedure in which it is discarded. A program consists of a list of global data declarations and procedures in which it is declared.

There are two pointers as one is stack pointer (SP) always points to a particular position in the activation record for the currently activate procedure. The second pointer is called top, which always points to the top of the stack i.e., top of the activation record.

The temporaries are used for expression evaluation and allocated above the activation record. An Activation Record is a data structure that is activated/ created when a procedure/function is invoked, and it contains the following information about the function.

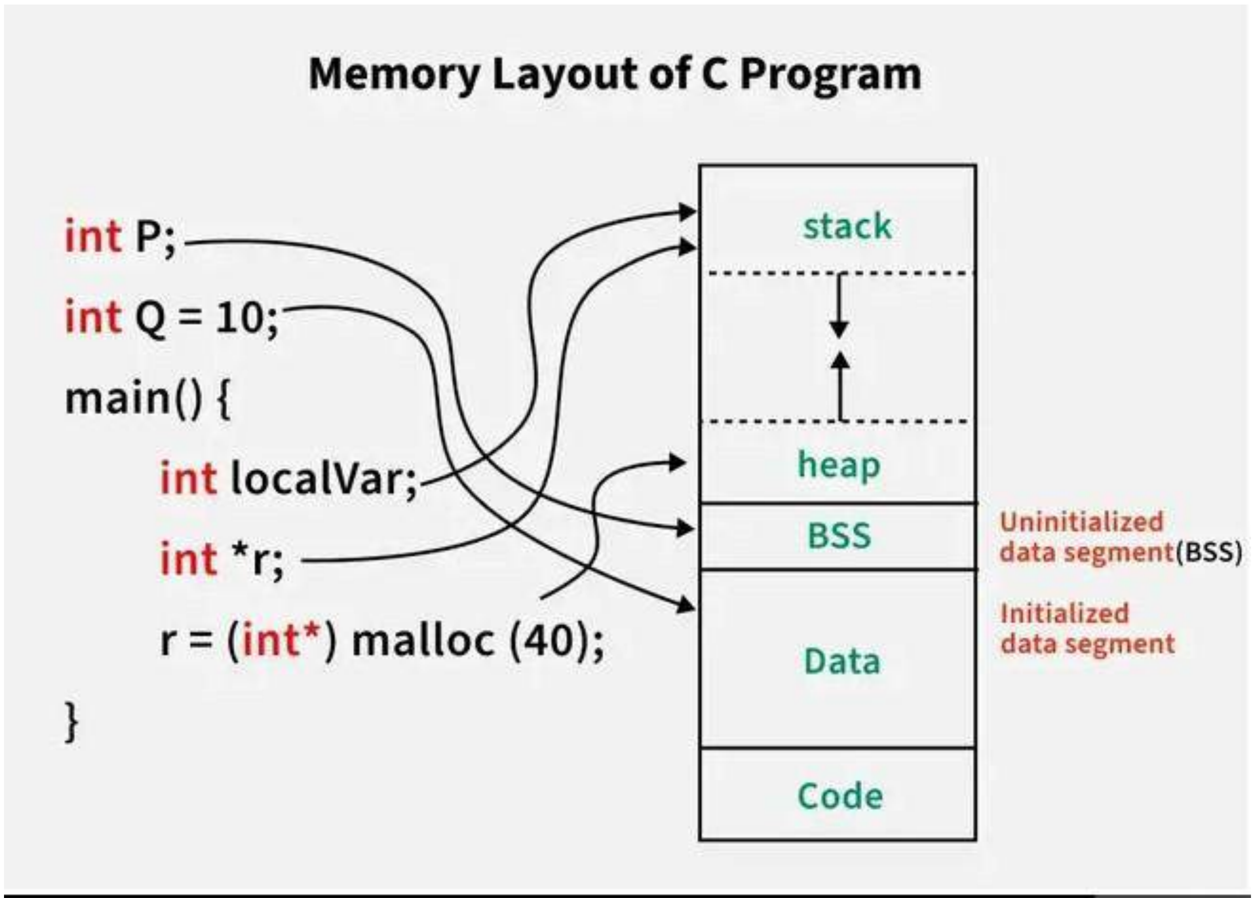
Activation Record in 'C' language consist of

- Actual Parameters
- Number of Arguments
- Return Address
- Return Value
- Old Stack Pointer (SP)
- Local Data in a function or procedure



Activation Record

Old SP stores the value of stack pointer of Activation Record of procedure which has called this procedure which leads to the generation of this Activation Record, i.e., It is a pointer to the activation record of the caller.
Optional access link– It defines non-local data held in another activation record.
Optional control link– It points to the activation record of the caller.
Actual Parameter– It is used by the calling procedure to supply parameters to the called procedure.
Return value– This field is used by the called procedure to return a value to the calling procedure.
The size of each of the above fields is determined at the time when the procedure is called. The size of almost all the fields can be determined at compilation tim



```
int P;
int Q = 10;
main() {
      int localVar;
      int *r;
      r = (int*) malloc (40);
}
```

Static Storage Allocation

Static storage allocation is the simplest form of memory allocation. In this strategy, the memory for variables is allocated at compile time. The compiler determines the memory addresses for all variables before the program starts running, and these addresses remain the same throughout the program's execution.
For example, in languages like C or C++, when you declare global variables or static variables, the memory for these variables is allocated using static storage.

```
int number = 1; // Global
static int digit = 1;
```

Advantages of Static Storage Allocation

1. The allocation process is straightforward and easy to understand.
1. The memory is allocated once only at compile time and remains the same throughout the program completion.
1. Memory allocation is done before the program starts taking memory only on compile time.

hEAP MANAGEMENT

Heap allocation is the most flexible allocation scheme. Allocation and deallocation of memory can be done at any time and any place depending upon the user's requirement. Heap allocation is used to allocate memory to the variables dynamically and when the variables are no more used then claim it back.
Heap management is specialized in data structure theory. There is generally some time and space overhead associated with heap manager. For efficiency reasons, it may be useful to handle small activation records of a particular size as a special case, as follows −

- For each size of interest, keep the linked list of free blocks of that size.
- If possible fill the request for size s with a block of size S', where S' is the smallest size greater than or equal to s. When the block is deallocated return back to the linked list.
- For a larger block of storage use the heap manager.

Garbage Collection Method

When the all-access path to an object is destroyed, but data object continues to exist, such types of objects are said to be garbage. Garbage collection is a technique that is used to reuse that object space.

In garbage collection, firstly, we mark all the active objects, and all the remaining elements whose garbage collection is 'on' are garbaged and returned to the free space list.

Reference Counter

By reference counter, an attempt is made to reclaim each element of heap storage immediately after it can no longer be accessed.

Each memory cell on the heap has a reference counter associated with it that contains a count of the number of values that points to it. The count has incremented each time a new value point to the cell and decremented each time an amount ceases to point to it. When the counter becomes zero, the cell is returned to the free list for further allocation.

Properties of Heap Allocation

There are various properties of heap allocation which are as follows −

- **Space Efficiency**− A memory manager should minimize the total heap space needed by a program.
- **Program Efficiency**− A memory manager should make good use of the memory subsystem to allow programs to run faster. As the time taken to execute an instruction can vary widely depending on where objects are placed in memory.
- **Low Overhead**− Memory allocation and deallocation are frequent operations in many programs. These operations must be as efficient as possible. That is, it is required to minimize the overhead. The fraction of execution time spent performing allocation and deallocation.