MALLA REDDY UNIVERSITY

III Year B. Tech – I Semester L/T/P/C 3/0/0/3

(MR20-1CS0112) COMPILER DESIGN

Course Objectives:

- To introduce the Finite Automata, NFA and DFA.
- To gain insight into the Context Free Language.
- To study the Phases of a Compiler and Lexical Analysis and Syntax Analysis.
- To acquaint the Intermediate Code Generation, Code Optimization and Code Generation.

UNIT –I

Finite Automata and Regular Expressions: Finite Automata- Examples and Definitions - Accepting the Union, Intersection, Difference of Two Languages. Regular Expressions: Regular Languages and Regular Expressions– Conversion from Regular Expression to NFA and Deterministic Finite Automata. Context free grammar: Derivations trees and ambiguity – Simplified forms and Normal forms.

UNIT –II

Introduction to Compiler: Introduction to Compilers: Definition of compiler – Interpreter-bootstrapping –phases of compiler.

Lexical Analysis: Roles of Lexical analyzer –Input buffering – specification of tokens – Recognition of Tokens – A language for specifying lexical analyzers – design of a Lexical analyzer.

UNIT –III

Parsing: Role of parser - Top Down Parser: Backtracking, Recursive Decent Parsing and Predictive Parsers. Bottom up Parser: Shift Reduce Parsing – LR parsers : SLR Parser, CLR parser and LALR Parsers.

UNIT –IV

Syntax Directed Translation and Intermediate code Generator: Syntax Directed Definitions – construction of Syntax tree. Intermediate code Generation : Abstract syntax tree – three address code – types of three address statements – syntax directed translations into three address code. Boolean expression and flow of control statements.

UNIT –V

Code optimization and Code generation: Basic blocks and flow graphs – optimization of basic blocks – principal sources of optimization – loop optimization – DAG representation of basic blocks. Simple code generator – register allocation and assignments – peephole optimization.

TEXTBOOKS:

1. Johne. Hopcroft and Jeffrey D. Ullman. Introduction to Automata Theory, Languages and Computation, 3rd Edition, Pearson Edition.
2. AlfredAho,V.RaviSethi,andD.JefferyUllman,"CompilersPrinciples,TechniquesandTools",Addiso n-Wesley, 2ndEdition,2007.
3. JohnC.Martin,,,IntroductiontoLanguagesandtheTheoryofComputations",McGrawHill,3rdEditio n, 2007.

REFERENCEBOOKS:

1.MishraK.L.P,,,TheoryofComputerScience:Automata,LanguagesandComputation",PHIpress2006
2. "Theory of Computation & Applications - Automata Theory Formal Languages"- by AnilMalviya,MalabikaDatta, BPBpublications,2015.
3. CharlesN.FischerandRichardJ.Leblanc,"CraftingaCompilerwithC",BenjaminCummings,2009.

Course Outcomes:

- After completion of the course, the students will be able to Understand the concept of Finite Automata, NFA and DFA.
- Understand about Context Free Language .
- Explain the concept of Phases of a Compiler, Lexical Analysis and Syntax Analysis.
- Describe the Intermediate code generation, Code Optimization and Code Generation.

UNIT –I

Finite Automata and Regular Expressions: Finite Automata- Examples and Definitions - Accepting the Union, Intersection, Difference of Two Languages. Regular Expressions: Regular Languages and Regular Expressions– Conversion from Regular Expression to NFA and Deterministic Finite Automata. Context free grammar: Derivations trees and ambiguity – Simplified forms and Normal forms

Finite Automata- Examples and Definitions

Theory of Computation(TOC) or Automata theory is a branch of computer science and mathematics, deals with the study of abstract machines and the computational problems they can solve.
It involves the analysis and classification of automata based on their capabilities, such as their ability to recognize patterns or process inputs.

Automata Theory provides a simple, elegant view of the complex machine called a computer.
There are 3 basic concepts of Theory of Computation(TOC)

1. **Automata**, abstract machine that takes strings of symbols as input and outputs either "yes" or "no."
2. **Formal Languages,** language accepted by the machine.
3. **Grammar,** set of rules for building language.

What is Automata?
Automata, also known as **automaton** (singular) or automata (plural), refer to mechanical or abstract devices that are capable of performing specific tasks or operations with little or no human intervention (hence the prefix "auto") to process some information.
In general, automata are designed to operate based on predefined rules or algorithms.

a) They can be physical machines, such as robots or self-operating mechanical devices, or
b) they can be theoretical models used in computation theory.

Automaton is nothing but a machine that accepts the strings of a language L over an input alphabet Σ .These devices are often used in the field of computer science, mathematics, and engineering to model and solve complex problems.

Types of Automata
Here are the main types of automata in the field of automata theory:

1. **Finite Automata (FA) or finite state machines(FSM)**
    a) **Deterministic Finite Automata (DFA):**
    b) **Nondeterministic Finite Automata (NFA):**
2. **Pushdown Automata (PDA)**
3. **Linear Bound Automata(LBA)**
4. **Turing Machines (TM)**.

Model	Language Recognition	Grammar used	Memory management	Computing Power
Finite Automata	Regular languages	Regular grammar	No memory management	Used for Small computing like elevators, vending machines, traffic lights, neural networks, spell checkers
Pushdown Automata	Context-free Languages	Context-free grammar	stack	Used for medium computing like compilers for programming languages, Online Transaction Processing system, Tower Of Hanoi (Recursive Solution)
Linear Bound Automata	Context sensitive Languages	Context sensitive grammar	Limited memory proportional to the size of the input.	Used for High computing like implementing algorithms. But less powerful than Turing Machines. Ex:- Genetic Programming, semantic analysis of the compiler
Turing Machine	Unrestricted Languages	Unrestricted grammar	Random Access Memory	Used for High computing like implementing any algorithms. Ex:- Artificial intelligence, Cryptography, Algorithmic complexity ,theory Compiler design, Computation theory

What is Finite Automata?

Finite automata, also known as **finite state machines (FSMs),** are abstract machine which has a set of states and rules for moving from one state to another but it depends upon the applied input symbol. Based on the states and the set of rules the input string can be either accepted or rejected.

They are a fundamental concept in automata theory and play a crucial role in understanding computation and language recognition.

Need of Finite Automata?

How are strings and languages used in in practice.

Question: Which symbols are used in the C programming language?
Answer: Σ = {a,b..z,A,B..Z,0,1..9,+, * ...},
The alphabet is finite, so you can answer it.

The program:

```
void main(){
  int a,b:
}
```

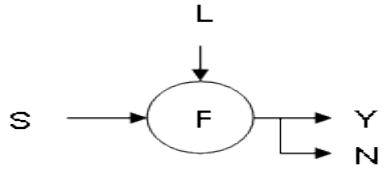
Question: If you have a program Pn how can you know if it valid?
Answer: Check if the string (S) is available in the language (L). However the language is infinite, so you can not do this.

Example:
Σ = {a,b}

L1 = {aa, ab, ba, bb}
Note: L1 is finite.
S = aa
S is available in L

L2 = {a, aa, aaa, ab, ...}
Note: L2 is infinite.
S = baba
It takes forever to check if S is available in L

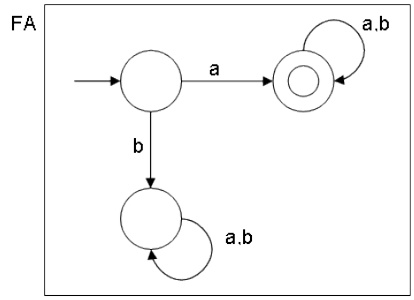
To solve the last problem you must convert the language L to a finite representation F.



The finite representation F can check if a string S is present in language L.
If present the output is Y(es) otherwise the output is N(o).

If the circle represents a machine with a finite amount of memory where the finite representation F is stored, the finite representation F is called the Finite Automata (FA).

There are two ways to create a finite representation of a language.
For example: L1 = Set of all strings which starts with an a.

Representation of a language using a finite set	**Representation of a language using a Finite Automata (FA) diagram**
L1 = {a,aa,ab,aaa,...}	

Finite Automata Model:

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

1) **Input tape:** It is a linear tape having some number of cells. Each input symbol is placed in each cell.
2) **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.

Finite automata 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.

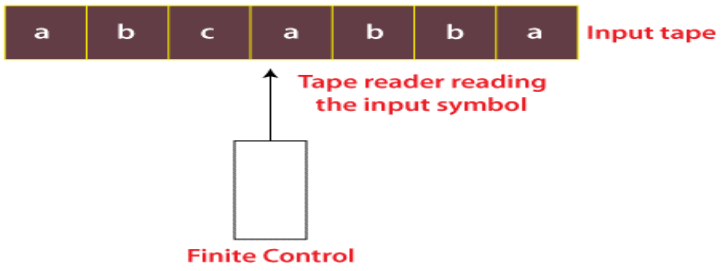


Fig: Finite automate model

A finite automata (FA) is the most restricted model of automatic machine. A finite automata is an abstract model of a computer system. As per its defining characteristics is that they have only a finite number of states. Hence, a finite automata can only "count" (that is, maintain a counter, where different states correspond to different values of the counter) a finite number of input scenarios. Hence there has some of the restriction or limitations are as follows;

1. The input tape is read only and the only memory it can have is by moving from state to state and since there are finite number of states, a finite automata has memory which is strictly finite.
2. The finite automata have only string pattern (regular expression) recognizing power.
3. The head movement is restricted in one direction, either from left to right or right to left.

We can modify or enhance the model of finite automata by removing one or more limitation like movement in both direction (Two way finite automata) but these enhancements do not increase the recognizing power of finite automata.

Examples of Limitation of finite auotmata:
There is no finite automaton that recognizes these strings:

1. The set of binary strings consisting of an equal number of a's and b's.
2. The set of strings over '(' and ')' that have "balanced" parentheses.

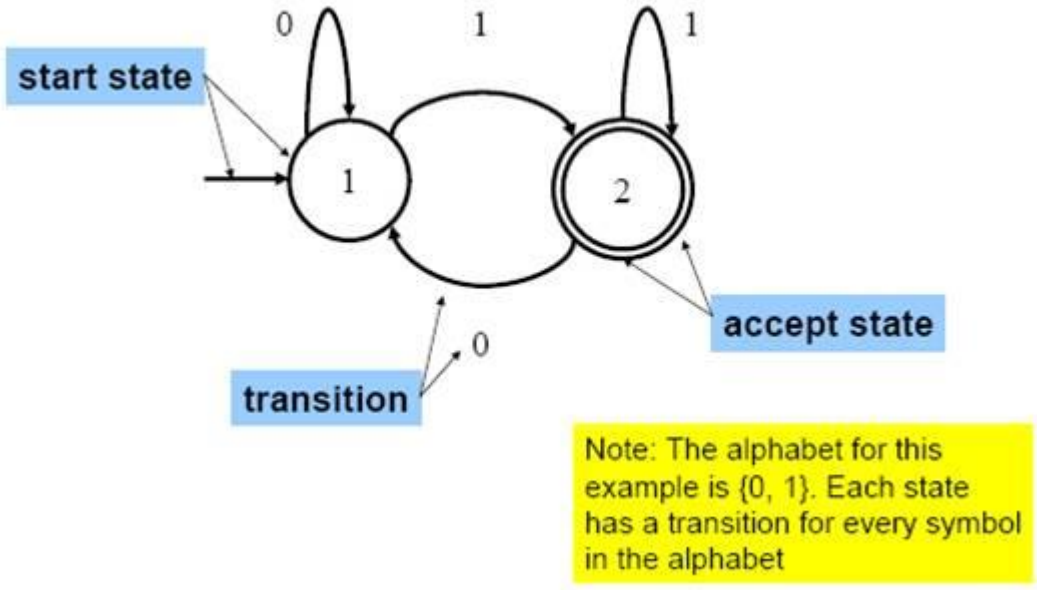
The 'pumping lemma' can be used to prove that no such FA exists for these examples.

Finite Automata – Formal Definition:
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

A finite automaton consists of the following components:

1. **States:** The system or machine can exist in a finite set of states. Each state represents a particular condition or configuration of the system.
2. **Start State:** The automaton has a designated start state from which the computation begins. It represents the initial configuration of the system.
3. **Accepting States:** Some states in the automaton are marked as accepting or final states. When the machine reaches an accepting state after processing a sequence of inputs, it signifies that the input sequence is accepted by the automaton. Accepting states indicate the successful recognition of a specific pattern or language.



4. **Transitions:** The transition diagram is also known as the **State transition diagram.** Transitions describe how the system moves from one state to another based on inputs. They are represented by arrows or directed edges and are labeled with input symbols. For a given state and input symbol, there is a defined transition to another state or states.
5. **Inputs:** The machine receives inputs from an input alphabet, which is a finite set of symbols. These symbols trigger the transitions in the automaton.

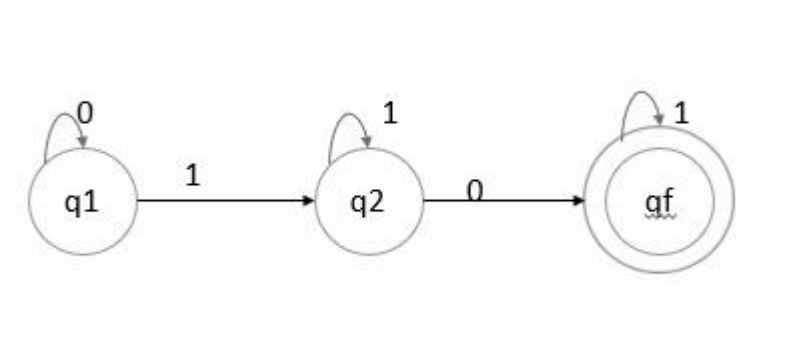
Finite Automata Representation

The finite automata can be represented in three ways, as given below −

1. Graphical (Transition diagram)
2. Tabular (Transition table)
3. Mathematical (Transition function)

Transition Diagram

It is a directed graph associated with the vertices of the graph corresponding to the state of finite automata.
An example of transition diagram is given below −



Here,

- {0,1}: Inputs
- q1: Initial state
- q2: Intermediate state
- qf: Final state

Transition table

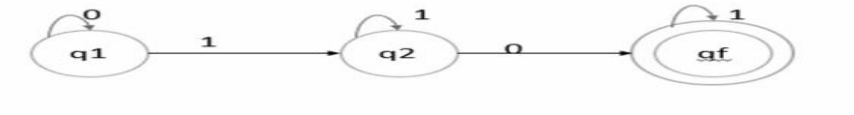
It is basically a tabular representation of the transition function that takes two arguments (a state & a symbol) and returns a value (the 'next state').

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

In transition table, the following factors are considered −

- Rows correspond to state.
- Column corresponds to the input symbol.
- Entries correspond to the next state.
- The start state is marked with ->.
- The accept state marked with *.

An example of transition table is as follows −



The transition table is as follows −

State/input symbol	0	1
->q1	q1	q2
q2	qf	q2
qf	-	qf

Transition function

The transition function is denoted by δ. The two parameters mentioned below are the passes to this transition function.

- Current state
- Input symbol

The transition function returns a state which can be called as the next state.

δ (current_state, current_input_symbol) = next_state

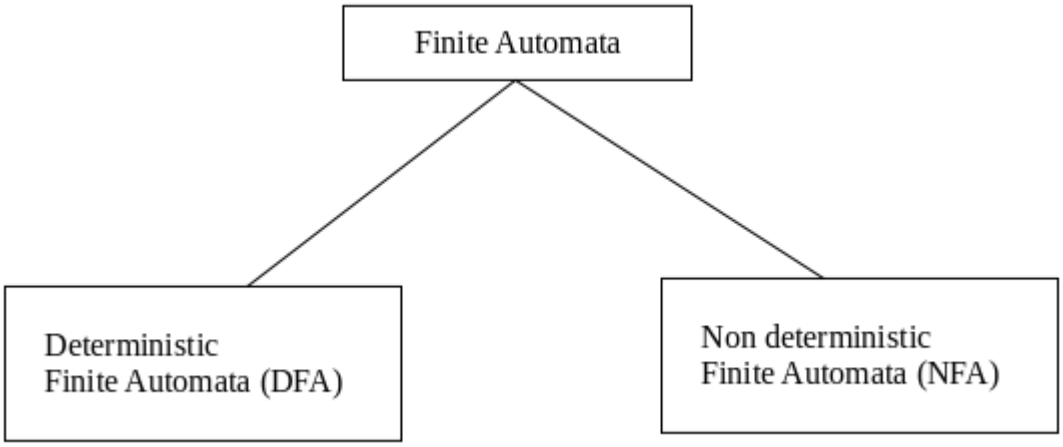
For example, δ(q0,a)=q1

Applications of Finite Automata

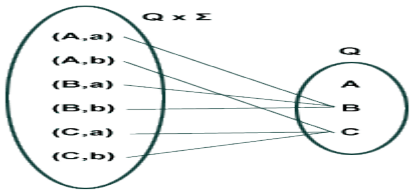
Finite automata are used in various areas, including pattern recognition, lexical analysis in compiler design, and designing simple systems with fixed behaviors. They provide a formal framework for understanding and analyzing the behavior of systems that can be described in terms of states and transitions. Finite automata have several practical applications across various fields. Here are some notable applications:

1. **Lexical Analysis:** Finite automata are extensively used in compiler design for lexical analysis, which involves tokenizing and scanning the source code of a programming language. Lexical analyzers employ finite automata to recognize and classify different lexical units such as keywords, identifiers, operators, and literals.
2. **Pattern Recognition:** Finite automata are employed in pattern-matching and recognition tasks. They can be used to search for specific patterns or sequences of characters within a given input. Applications include text processing, string matching, and searching algorithms.
3. **Network Protocol Analysis:** Finite automata are used in network protocol analysis and packet filtering. They can be employed to define rules and match patterns in network traffic, enabling functionalities like intrusion detection, firewalls, and network monitoring.
4. **Digital Circuit Design:** Finite automata can model the behavior of digital circuits and aid in their design and testing. Sequential circuits, such as counters and state machines, can be represented and analyzed using finite automata.
5. **Natural Language Processing:** Finite automata are used in natural language processing for tasks such as text tokenization, morphological analysis, and part-of-speech tagging. They help in parsing and understanding the structure of natural language sentences.
6. **Vending Machines:** Finite automata are an appropriate model for designing and implementing the control logic of vending machines. They can manage the states and transitions required to process user inputs and dispense the appropriate products.
7. **Spell checkers:** Finite automata is highly useful to design spell checkers
8. **Design text editors:** A finite automata is useful to design text editors
9. **Regular Expression Processing:** Finite automata are closely related to regular expressions, a powerful tool for specifying patterns in strings. Regular expression engines often utilize finite automata to efficiently match and process regular expressions.
10. **DNA Sequence Analysis:** Finite automata find applications in bioinformatics for analyzing DNA sequences. They can be used to identify specific patterns or motifs within DNA sequences, aiding in gene identification, sequence alignment, and genetic research.

Finite automata can be categorized into two main types:



1. **Deterministic Finite Automaton (DFA):**
   - In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called Deterministic Automaton. As it has a finite number of states, the machine is called Deterministic Finite Machine or Deterministic Finite Automaton.
   - A Deterministic Finite Automaton (DFA) consists of a finite set of states and a set of transitions from one state to another state on input symbols selected from an alphabet Σ.
   - There is only one initial state, generally denoted by q0.
   - A transition function is defined on **every state for every input symbol**.



Transition Function
δ : Q X Σ --> Q

   - There are one or more “final” or “accepting” states.
   - In DFA null (or ε) move is not allowed, i.e., DFA cannot change state without any input character.

A finite automaton is formally defined by a 5-tuple (Q, Σ, δ, q0, F),
where,
Q → Finite set of states.
Σ → Finite set of input symbols.
δ → Transition function mapping Q × Σ to Q.
q0 → Initial state and q0 ∈ Q, only one initial states
F → Set of final state/states. It is assumed that there may be more than one final state. F ⊆ Q.

NOTE:- One important thing to note is, **there can be many possible DFAs for a pattern**. A DFA with a minimum number of states is generally preferred.

Notations for DFA

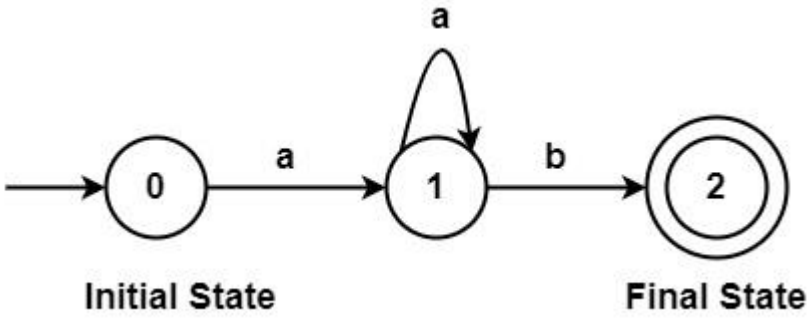
1. Transition Diagram:

- This is a graph in which vertices represent state of machine and the edges show transition of states.
- The labels on these edges indicate input/output for the corresponding transition.

A transition diagram for a DFA M = (Q, Σ, δ, q0, F) is a graph defined as follows:

- For each state in Q there is a node.
- There is labelled input symbol on transition.
- Then the transition diagram has an arc (arrow) from one node to another node.
- If multiple input symbols cause the same transition from one node to another node, then multiple input labels separated by the commas are given to that edge.
- There is an arrow into the start state q0 labeled as start. This arrow does not originate at any node.
- Final states/ accepting states are marked by concentric double circles.
- δ: Transition Function
  δ: Q X Σ --> Q.

- States not belonging to final states have a single circle. e.g.



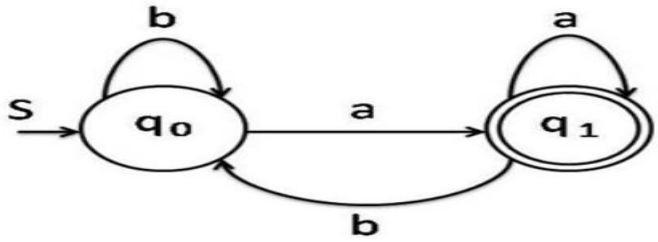
2. Transition Table:

- It specifies the resultant set of states for the corresponding current state and input to the machine.
- A transition table is a conventional tabular representation of a function like δ that takes two arguments and returns a value the rows of the table correspond to the states and the columns correspond to the inputs.
- The entry for the row corresponding to state q and the column corresponding to input q is the state δ(q, a).

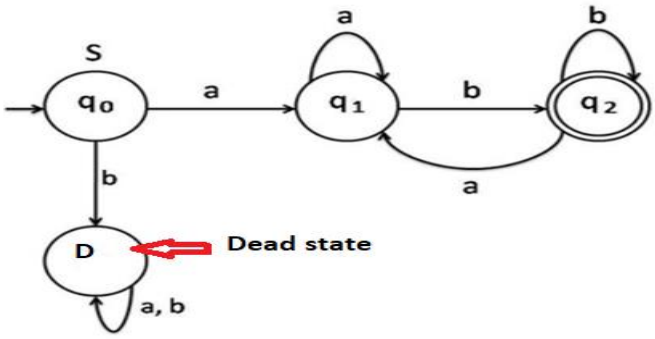
States \ Input	a	b
0	1	-
1	1	2
2	-	-

Examples of DFA

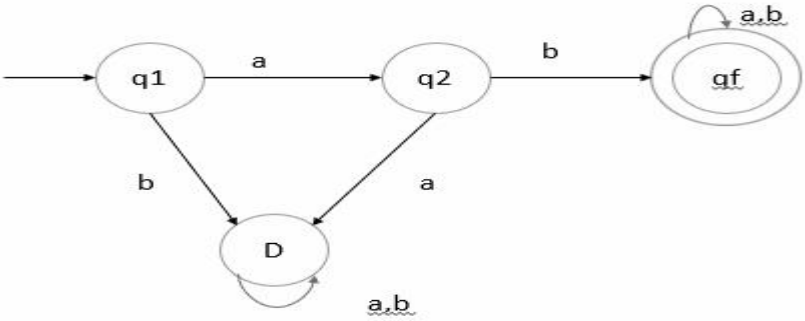
Ex:1 Design a DFA which accepts a language over the alphabets Σ={a,b} that accepts the string ending with ‘a’.
L={a,ba,aa,baa,bbaa}



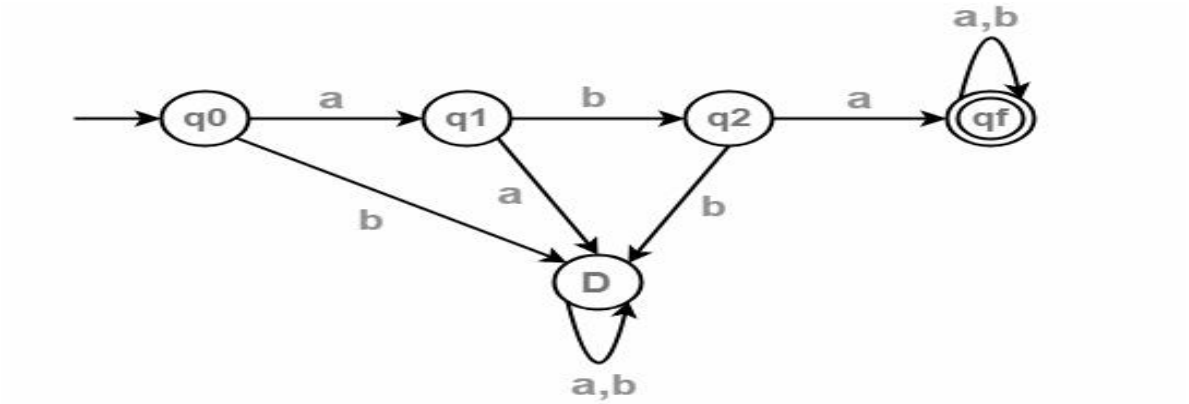
Ex:2 Design a DFA which accepts a language over the alphabets Σ={a,b} that accepts the string starting with ‘a’ ending with ‘b’.



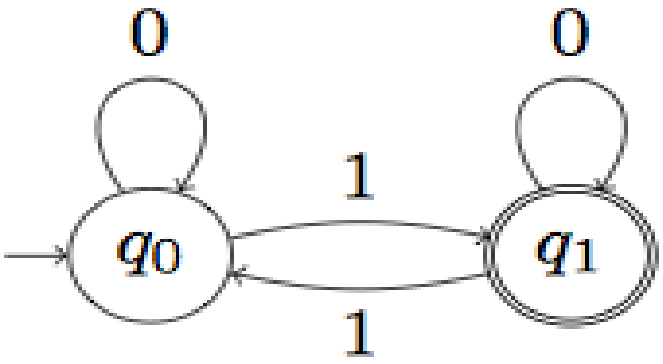
Ex:3 Design a DFA which accepts a language over the alphabets Σ={a,b} that accepts the string starting with ‘ab’.



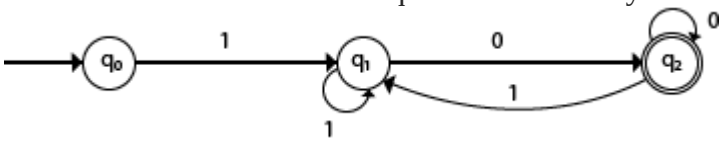
Ex:4 Design a DFA which accepts a language over the alphabets Σ = {a, b} such that L is the set of all strings starting with ‘aba’.



Ex:5 Design a DFA which accepts a language over the alphabets Σ = {0,1} such that L is the set of all strings having odd number of 1's



Example 6:
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 7:
Design a FA with ∑ = {0, 1} accepts the only input 101.
Solution:

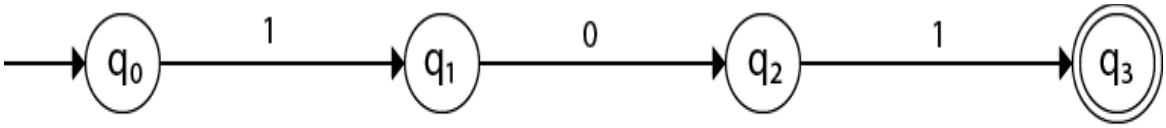


Fig: FA

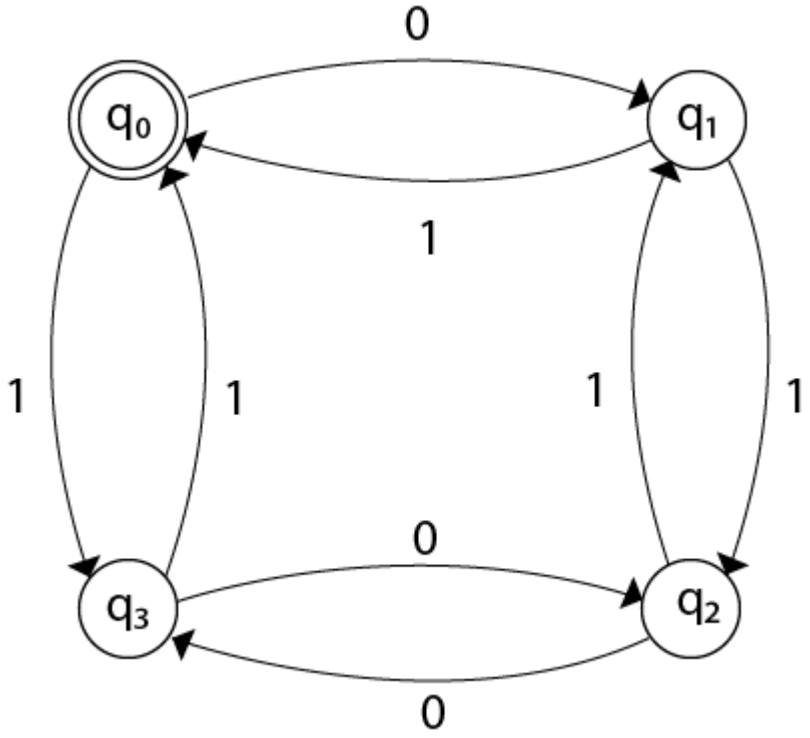
In the given solution, we can see that only input 101 will be accepted. Hence, for input 101, there is no other path shown for other input.

Example 8:

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:

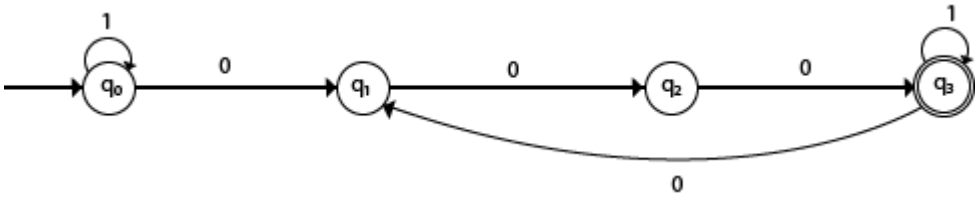
q0: state of even number of 0's and even number of 1's.
q1: state of odd number of 0's and even number of 1's.
q2: state of odd number of 0's and odd number of 1's.
q3: state of even number of 0's and odd number of 1's.

Example 9:

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:

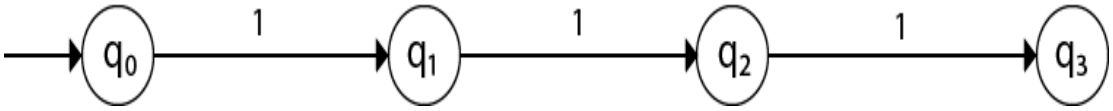


Note that the sequence of triple zeros is maintained to reach the final state.

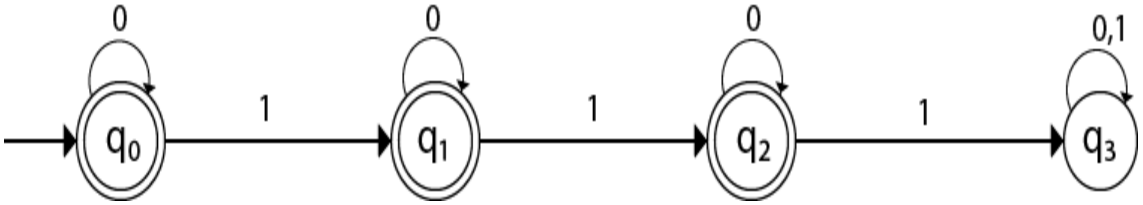
Example 10:

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:



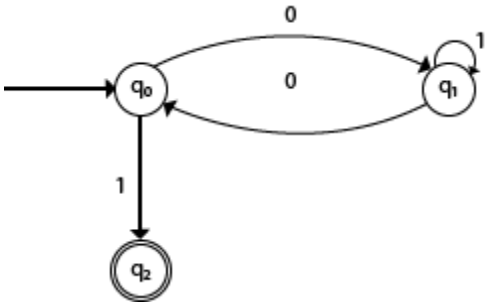
Here two consecutive 1's or single 1 is acceptable, hence



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 11:
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:



2. **Nondeterministic Finite Automaton (NFA):** NFA is similar to DFA except following additional features:
   - Null (or ε) move is allowed i.e., it can move forward without reading symbols.
   - Ability to transmit to any number of states for a particular input.

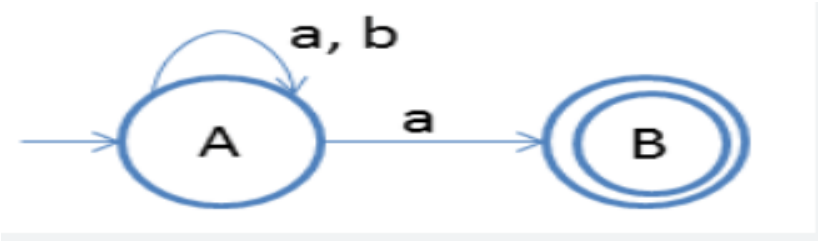
   ```
   Transition Function
   δ:  Q X (Σ U ε ) --> 2 ^ Q.
   ```

   A non-deterministic finite automaton (NDFA/NFA) are defined by a five-tuple: (Q, Σ, δ, q0, F), similar to DFAs.

where,
Q = is a finite set of states.
Σ = is a finite set of input symbols.
δ = is a transition function mapping from Q × Σ to 2Q.
q0 = is the initial state, q0 ∈ Q.
F = is a set of final states, F ⊆ Q.

Examples of NFA

Ex:1 Design a NFA which accepts a language over the alphabets Σ={a,b} that accepts the string ending with ‘a’.



The NFA that accepts strings ending with "ab" is given by

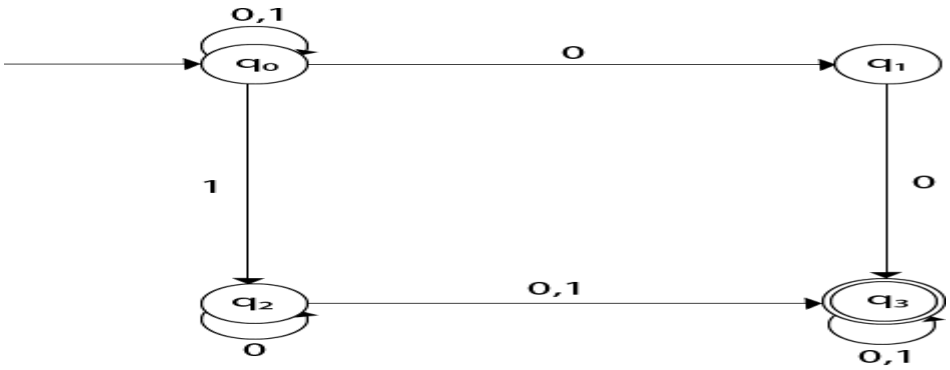
Example 2:

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:

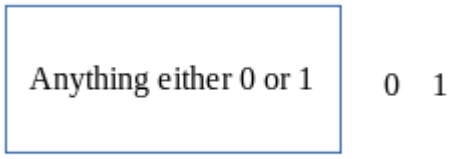
The transition diagram can be drawn by using the mapping function as given in the table.



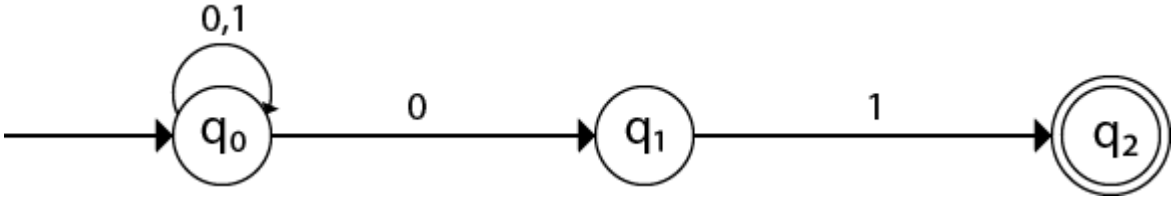
Here,

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}

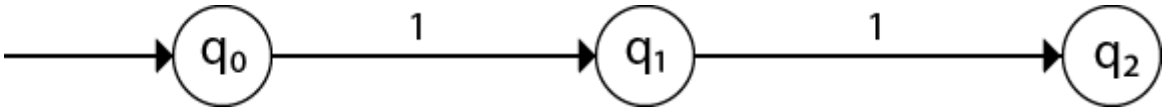
Example 3:
Design an NFA with ∑ = {0, 1} accepts all string ending with 01.
Solution:



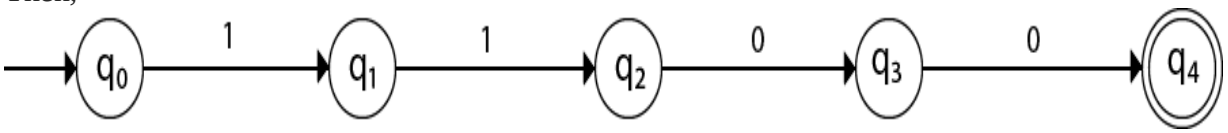
Hence, NFA would be:



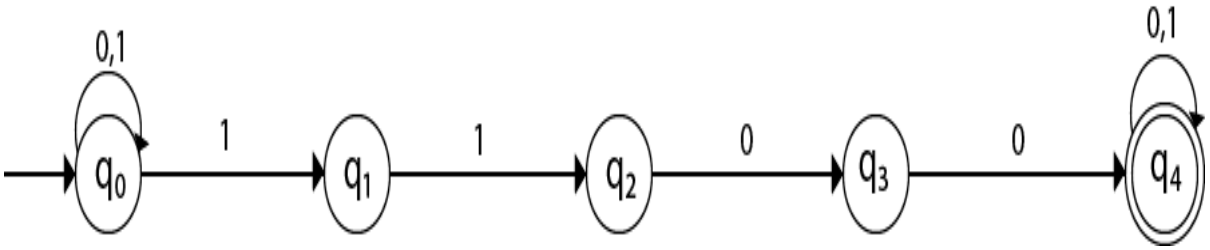
Example 4:
Design an NFA with ∑ = {0, 1} in which double '1' is followed by double '0'.
Solution:
The FA with double 1 is as follows:



It should be immediately followed by double 0.
Then,



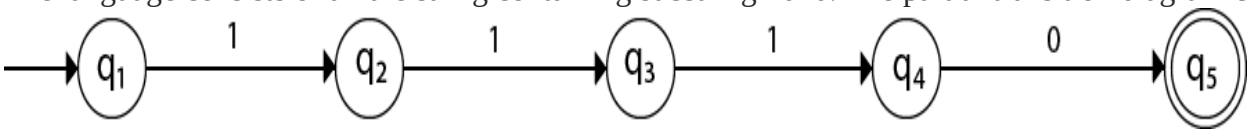
Now before double 1, there can be any string of 0 and 1. Similarly, after double 0, there can be any string of 0 and 1.
Hence the NFA becomes:



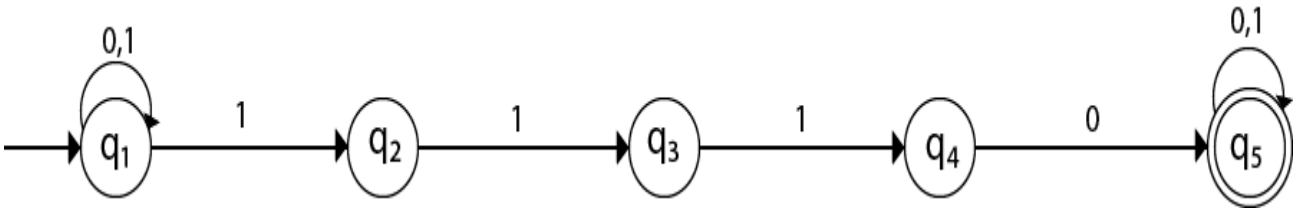
Now considering the string 01100011

1. q0 → q1 → q2 → q3 → q4 → q4 → q4 → q4

Example 5:
Design an NFA in which all the string contain a substring 1110.
Solution:
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,

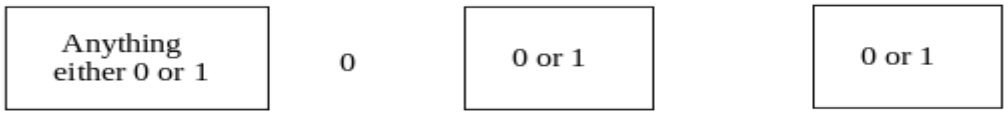
1. δ(q1, 111010) = δ(q1, 1100)
2. = δ(q1, 100)
3. = δ(q2, 00)

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

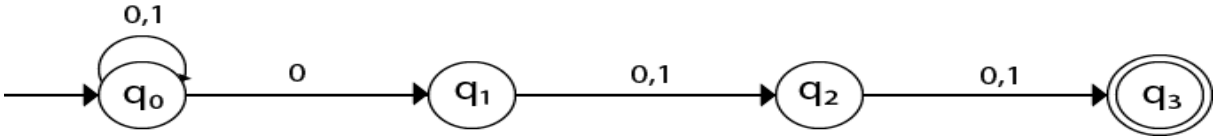
1. δ(q1, 111010) = δ(q2, 1100)
2. = δ(q3, 100)
3. = δ(q4, 00)
4. = δ(q5, 0)
5. = δ(q5, ε)

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

Example6:
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.

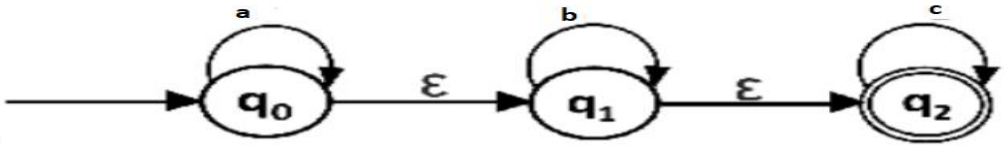
ε-Transitions

Another extension of Finite Automata is allowed to make a transition simultaneous without receiving an input symbol i.e, transition on **ε** (empty string).

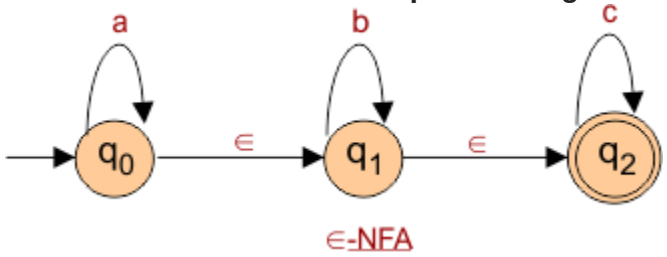
Some times its difficult to construct a direct NFA / DFA

Example construct **ε** -NFA to accept strings having a's followed by b's & followed by c's.

So we use **ε** strings as input like

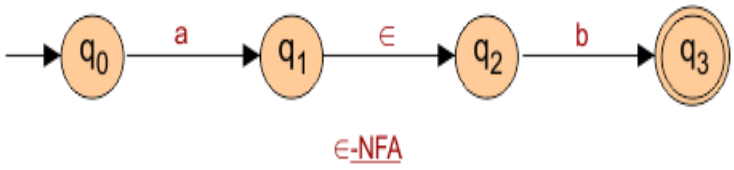


Draw a ε -NFA which can accept the string "a or b or c"



∈-NFA

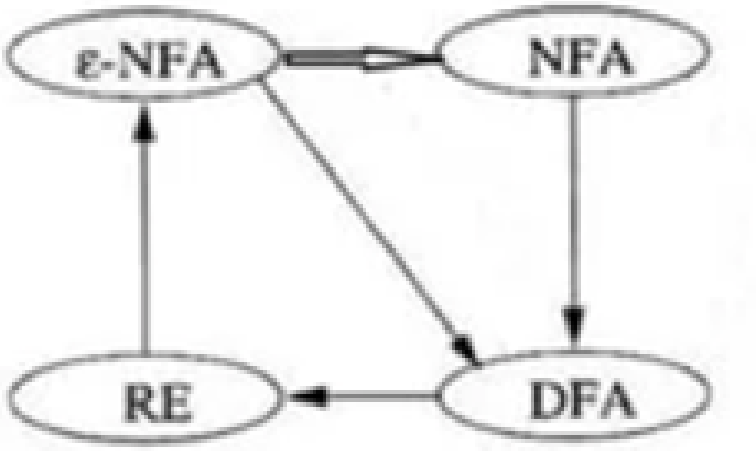
Draw a ε -NFA which accept the string "ab".



∈-NFA

Therefore, for many problems its easy to construct NFA with **ε** and then convert it into DFA.

Allowed conversions are



NFA WITH $\in$ - MOVES

Finite automata With $\in$ - Transitions

This is same as NFA except we are using a special input symbol called epsilon ($\in$). Using this symbol path we can jump to one state to other state without reading any input symbol.

This also analytically indicated as 5 - tuple notation.

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

$Q \rightarrow$ set of states in design

$\Sigma \rightarrow$ input alphabet

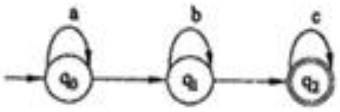
$q_0 \rightarrow$ initial state

$F \rightarrow$ final states ($\subseteq Q$)

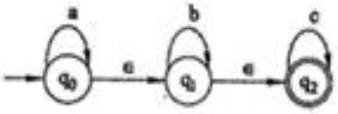
$\delta \rightarrow$ mapping function indicates $Q \times (\Sigma \cup \{\in\}) \rightarrow 2^Q$

Example : Draw a transition diagram of NFA which include transitions on the empty input $\in$ and accepts a language consisting of any number a's followed by any number of b's and which in turn followed by any number of c's.

Solution : It requires three states q_0, q_1 and q_2 and they accept any number of a's, b's and c's respectively. Assign q_2 as final state.



To reach from q_0 to q_1 and q_1 *to* q_2 no input will be given i. e., they treat $\in$ as their input and do the transition.



Normally these $\in$'s do not appear explicitly in the string.
The transition function for the NFA is shown below :

	a	b	c	$\in$
$\rightarrow q_0$	$\{q_0\}$	ϕ	ϕ	$\{q_1\}$
q_1	ϕ	$\{q_1\}$	ϕ	$\{q_2\}$
(q_2)	ϕ	ϕ	$\{q_2\}$	ϕ

For example consider the string $\omega = ab\,c$

String $\omega = ab\,c$ (i. e. , string in actual form is $a \in b \in c$ i. e., included along with epsilons).

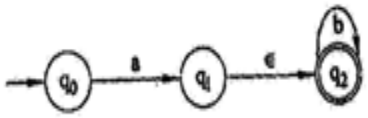
$$\begin{aligned} \delta(q_0, abc) &\vdash \delta(q_0, bc) \\ &\vdash \delta(q_0, \in bc) \\ &\vdash \delta(q_1, bc) \\ &\vdash \delta(q_1, \in c) \\ &\vdash \delta(q_2, c) \quad \vdash q_2 \in F \end{aligned}$$

The path is shown below :

$$q_0 \overset{a}{} q_0 \overset{\in}{} q_1 \overset{b}{} q_1 \overset{\in}{} q_2 \overset{c}{} q_2$$

with arcs labeled $a, \in, b, \in, c$

Example : The following NFA with $\in$ transitions accepts input strings with (a's and b's) single a or a followed by any number of b's.



NOTE:- Therefore, for many problems because of difficulty in constructing DFA directly its easy to construct NFA or NFA with ε and then finally convert it into DFA.

Difference Between DFA and NFA

S.No.	DFA	NFA
1	The full form of DFA is Deterministic Finite Automata.	The full form of NFA is Nondeterministic Finite Automata (NFA).
2	It is not competent in handling an Empty String transition.	It is competent to handle an empty string transition.
3	Transition Function δ: Q X (Σ U ε) --> 2 ^ Q.	Transition Function δ : Q X Σ --> Q
4	It can be defined as one machine.	Multiple machines execute computational tasks at the same time.
6	In DFA, empty string transitions are not noticed.	It allows empty string transition.
7	It is tough to construct a DFA.	It is easy to construct a NFA.
8	It needs more space.	It needs less space.
9	The complete time needed for managing any input string in DFA is shorter than NFA.	The complete time needed for managing any input string in NFA is larger than DFA.
10	All DFA are considered as NFA.	All NFA are not considered as DFA.

Conversion from NFA to 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

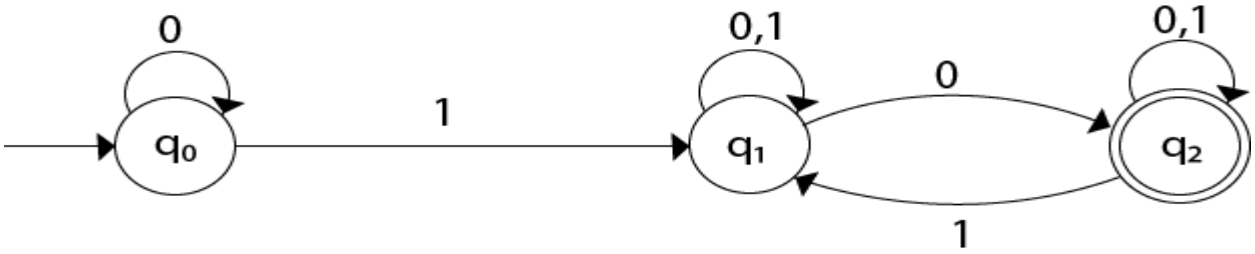
Step 2: Add q0 of NFA to Q'. Then find the transitions from this start state.

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

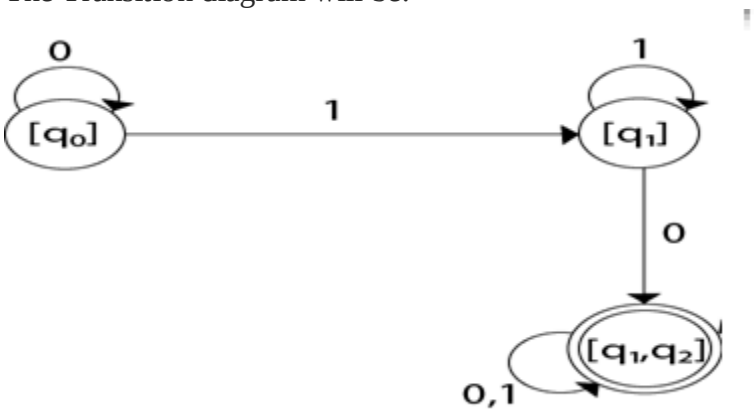
Now we will obtain δ' transition on [q1, q2].

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]
*[q1, q2]	[q1, q2]	[q1, q2]

The Transition diagram will be:



The state q2 can be eliminated because q2 is an unreachable state.
Ex:2 Convert following NFA to DFA

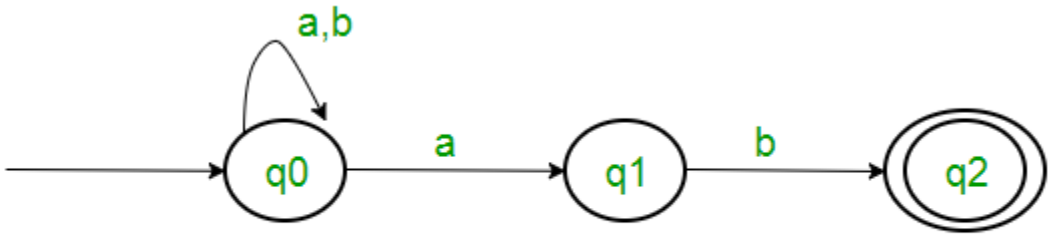


Figure 1

State	a	B
q0	{q0,q1}	q0
{q0,q1}	{q0,q1}	{q0,q2}
{q0,q2}	{q0,q1}	q0

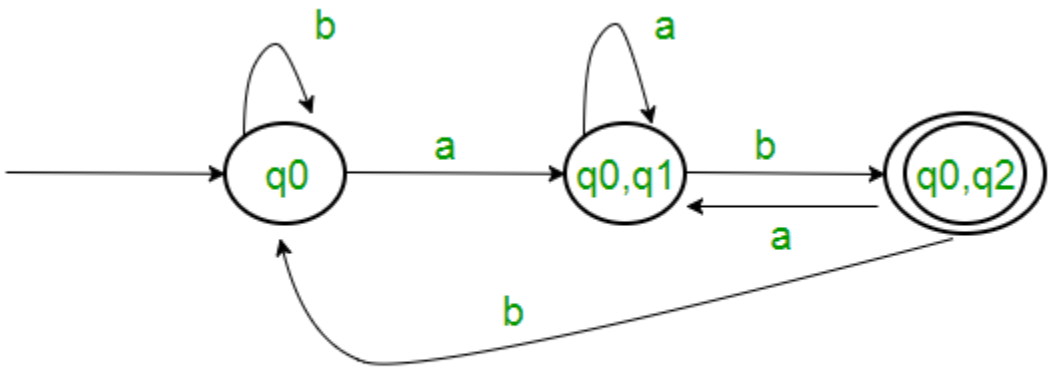
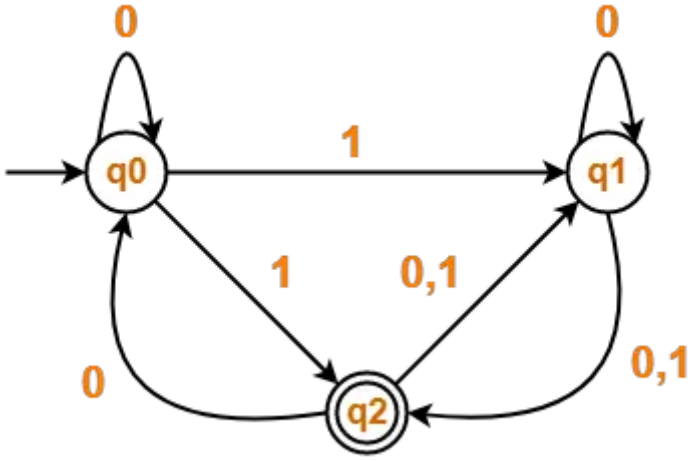
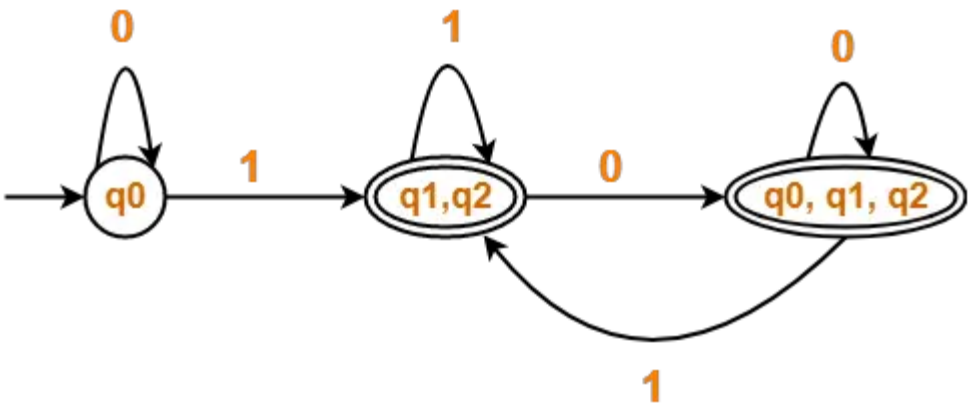


Figure 2

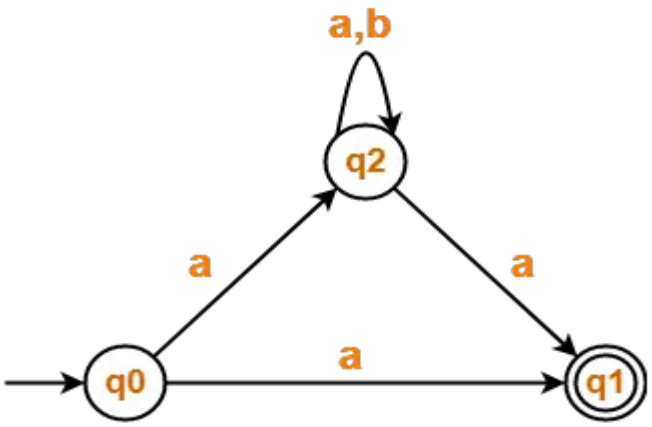
Ex:3 convert following NFA to DFA



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

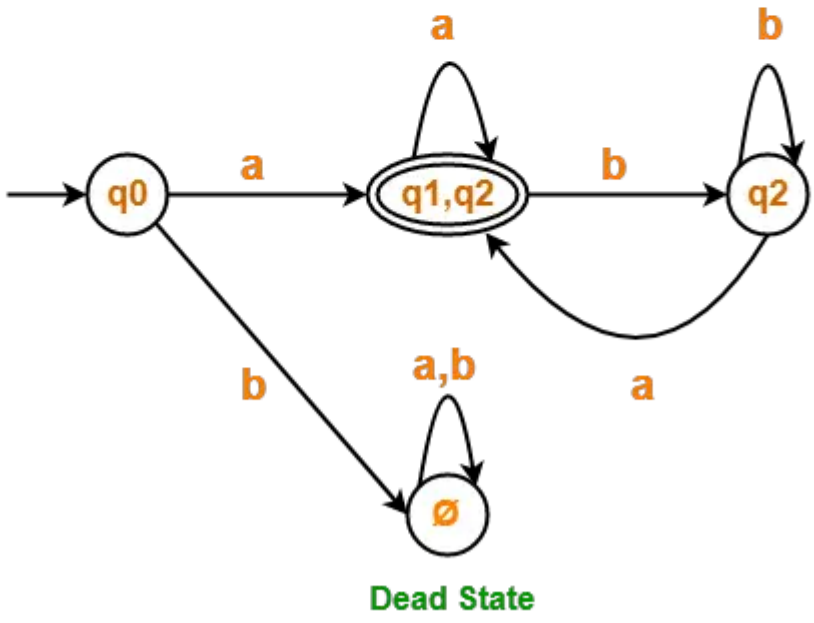


Ex:4 convert following NFA to DFA



State / Alphabet	a	b
→q0	*{q1, q2}	Ø
*{q1, q2}	*{q1, q2}	q2
q2	*{q1, q2}	q2
Ø	Ø	Ø

Now, Deterministic Finite Automata (DFA) may be drawn as-

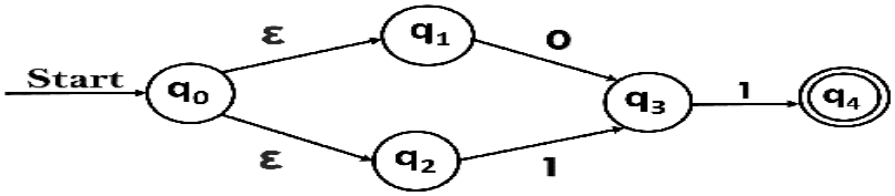


Deterministic Finite Automata (DFA)

Conversion from NFA with ε to NFA

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 NFA:
Step 1: Find ε-closure of all the states
Step 2: Obtain δ‘ transition for each state on each output.
Step 3: Using the resultant states build the transition table .
Step 4: If ε-closure(qi) has final state then qi will be final state in NFA.

Example :- convert the following ε- NFA to NFA



ε-closure {q0} = {q0, q1, q2}
ε-closure {q1} = {q1}
ε-closure {q2} = {q2}
ε-closure {q3} = {q3}
ε-closure {q4} = {q4}

Now, obtain δ‘ for each state on each input
δ'(q0, 0) = ε-closure {δ(ε-closure(q0), 0) }
= ε-closure {δ((q0, q1, q2), 0) }
= ε-closure {δ(q0, 0) ∪ δ(q1, 0) ∪ δ(q2, 0) }
= ε-closure {q3}
= {q3}

δ'(q0, 1) = ……=$\{q_3\}$
δ'(q1, 0) = ……=$\{q_3\}$
δ'(q1, 1) = ……= ϕ

δ'(q2, 0) = ……= ϕ
δ'(q2, 1) = ……=$\{q_3\}$

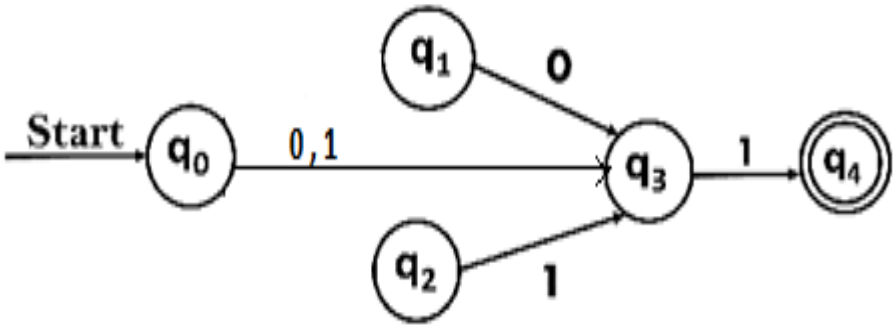
δ'(q3, 0) = ……=ϕ
δ'(q3, 1) = ……=$\{q_4\}$
δ'(q4, 0) = ……=ϕ
δ'(q4, 1) = ……=ϕ

Now draw the transition table

State	0	1
→q0	q3	q3
q1	q3	φ
q2	φ	q3
q3	φ	q4
q4	φ	φ

As in ε-NFA the ε-closure(q4) has final state q4

So in resultant NFA the final state will be q4.



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 ε 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.

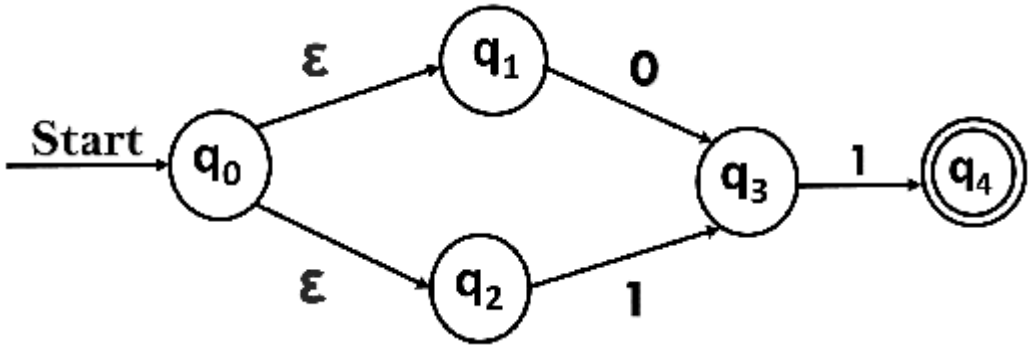
Step 3: If we found a new state, take it as current state and repeat step 2.

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}

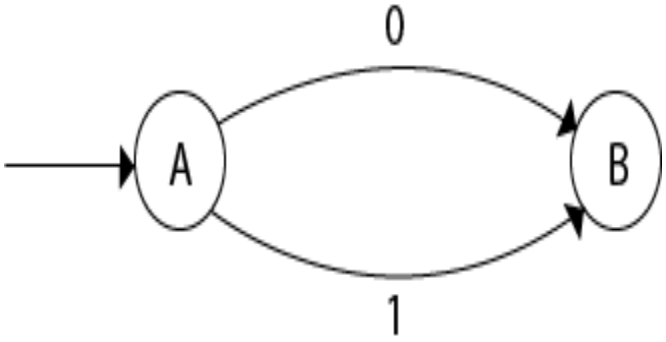
Now, let ε-closure {q0} = {q0, q1, q2} be state A.

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



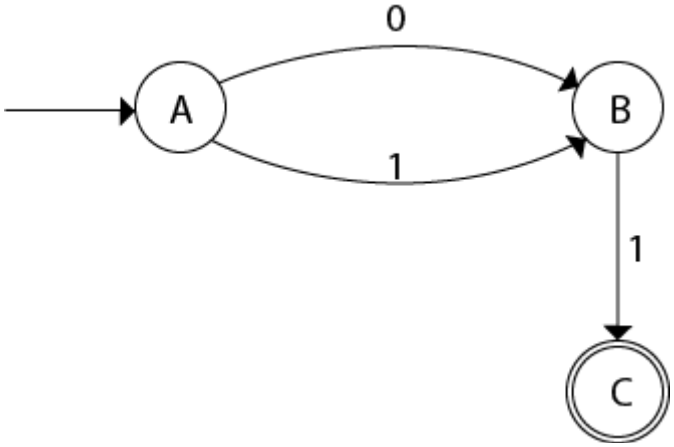
Now,

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**

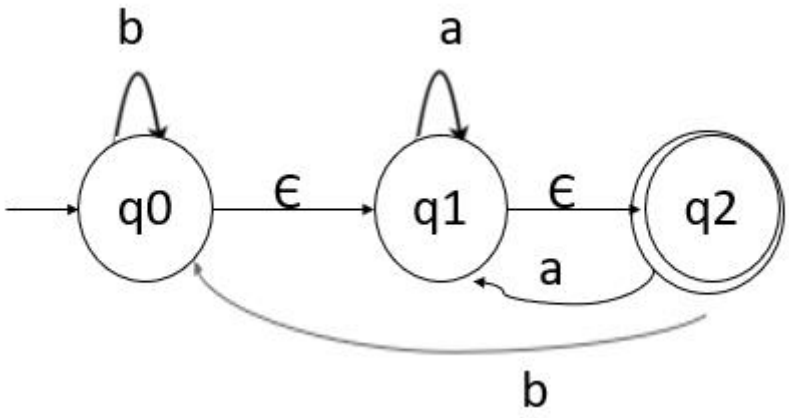
For state C:

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,



EX:-2 Convert the following NFA with epsilon to equivalent DFA



To convert this NFA with epsilon, we will first find the ε-closures, as given below −

- ε-closure(q0)={q0,q1,q2}
- ε-closure(q1)={q1,q2}
- ε-closure(q2)={q2}

Let us start from **ε-closure of start state**, as mentioned below −

When, ε-closure(q0)={q0,q1,q2}, we will call this state as A.

Now, let us find transition on A with every input symbol, as shown below −

δ'(A, a) = ε-closure(δ(A,a))
= ε-closure(δ(q0,q1,q2), a))
= ε-closure(δ(q0, a) ∪ δ(q1,a) U δ(q2,a))
= ε-closure(ΦUq1 ∪q2)
= ε-closure(q1)
= {q1, q2} let us call it as state B

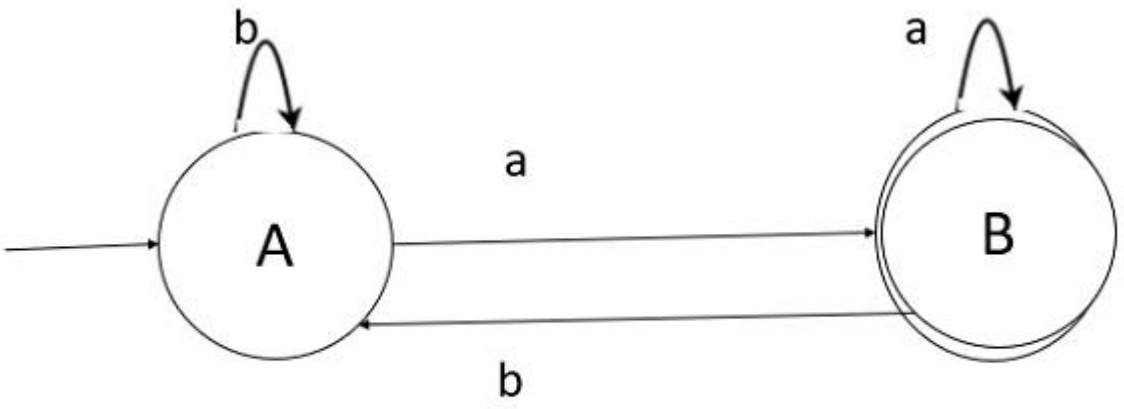
δ'(A, b) = ε-closure(δ(A,b))
= ε-closure(δ(q0,q1,q2), b))
= ε-closure(δ(q0, b) ∪ δ(q1,b) U δ(q2,b))
= ε-closure(q0 U Φ∪q0)
= ε-closure(q0)
= {q0,q1, q2} its nothing but state A

δ'(B, a) = ε-closure(δ(B,a))
= ε-closure(δ(q1,q2), a))
= ε-closure(δ(q1,a) U δ(q2,a))
= ε-closure(q1 ∪q2)
= ε-closure(q1)
= {q1, q2} its nothing but state B

δ'(B, b) = ε-closure(δ(B,b))
= ε-closure(δ(q1,q2), b))
= ε-closure(δ(q1,b) U δ(q2,b))
= ε-closure(Φ∪q0)
= ε-closure(q0)
= {q0,q1, q2} its nothing but state A

Hence, the transition table for the generated DFA is as follows −

States\inputs	a	b
A	B	A
B	B	A



$$F' = \{A, B\}$$

Finite Automata- Accepting the Union, Intersection, Difference of Two Languages.

An **alphabet** (Σ) or is a finite set of input symbols.
Ex: ∑ = {0, 1} is an alphabet of binary digits

A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet.
Ex:- for alphabet ∑ = {0, 1} w = 010101 is a string.

Length of a string is denoted as **|w|** and is defined as the number of positions for the symbol in the string.
Ex:- for string w = 010101, |w| =6

empty string ϵ is the string with zero occurrence of symbols.

Phi ∅ is no string or null

A **language** is a set of strings from some alphabet (finite or infinite).
Operations on Languages:-

1. union
2. Intersection
3. Difference
4. Concatenation
5. Kleene closure (∑*)
6. Positive closure (∑+)

∑* contains an empty string ϵ, ∑* = ∑+ U {ϵ }
Ex:- for ∑={0,1} then ∑* = {ϵ, 0, 1, 01, 10, 00, 11, 10101,…}

∑+ denotes non- empty string., ∑+ = ∑* - {ϵ }
Ex:- for ∑={0,1} then ∑+ = { 0, 1, 01, 10, 00, 11, 10101,…}

∑1 = {0, 1}
∑2 = {00, 01, 10, 11}
If language L = {x ∈ {0, 1} } and S = {x ∈ {a, b, c}} then concatenation L.S={0a,1a,0b,1b,0c,1c}

Union, Intersection, Difference of Two Languages

Suppose L1 and L2 are both languages over Σ. If x ∈ Σ* , then knowing whether x ∈ L1 and whether x ∈ L2 is enough to determine whether x ∈ L1 ∪ L2. This means that if we have one algorithm to accept L1 and another to accept L2, we can easily formulate an algorithm to accept L1 ∪ L2. So if we actually have finite automata accepting L1 and L2, then there is a finite automaton accepting L1 ∪ L2, and that the same approach also gives us FAs accepting L1 ∩ L2 and L1 − L2.

Theorem

Suppose M1 = (Q1, Σ, q1, A1, δ1) and M2 = (Q2, Σ, q2, A2, δ2) are finite automata accepting L1 and L2, respectively.

Let **M** be the FA (Q, Σ, q0, A, δ), where **Q = Q1 × Q2** and **q0 = (q1, q2)** and the transition function δ is defined by the formula **δ((p, q), σ) = (δ1(p, σ), δ2(q, σ))** for every p ∈ Q1, every q ∈ Q2, and every σ ∈ Σ. Then

1. If A = {(p, q) | p ∈ A1 or q ∈ A2}, M accepts the language L1 ∪ L2.
2. If A = {(p, q) | p ∈ A1 and q ∈ A2}, M accepts the language L1 ∩ L2.
3. If A = {(p, q) | p ∈ A1 and q /∈ A2}, M accepts the language L1 − L2

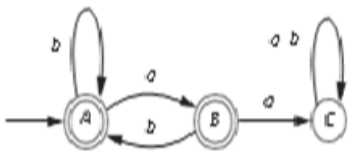
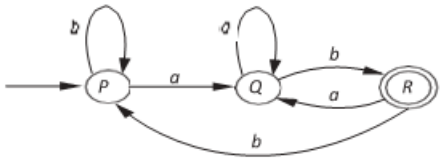
Example1:-

Let L1 and L2 be the languages

L1 = {x ∈ {a, b} ∗ | aa is not a substring of x}

L2 = {x ∈ {a, b} ∗ | x ends with ab}

Finite automata M1 and M2 accepting these languages are obtained as shown

	
FA for machine M1 accepting language L1	FA for machine M2 accepting language L2
Q1={A,B,C} q0=A A1={A,B} δ (A, a) = {B} δ(A,b) = {A} δ (B,a) = {C} δ(B,b) = {A} δ (C, a) = {C} δ (C, b) = {C}	Q2={P,Q,R} q0=P A2={R} δ(P,a) = {Q} δ(P,b) = {P} δ(Q,a) = {Q} δ(Q,b) = {R} δ(R,a) = {Q} δ(R,b) = {P}

Then according to the given Theorem ,

M be the FA (Q, Σ, q0, A, δ),

where **Q = Q1 × Q2** i.e, {(AP), (AQ),(AR),(BP),(BQ),BR),(CP),(CQ),(CR)}

q0 = (q1, q2) i.e, (A,P)

the transition function δ is defined by the formula **δ((p, q), σ) = (δ1(p, σ), δ2(q, σ))**

18 transition functions for M are produced as follows

δ((A,P), a) = (δ((A, a), (P, a)) ={BQ}

δ((A,P), b) = (δ((A, b), (P, b)) ={AP}

δ((A,Q) a) = (δ((A, a), (Q, a)) ={BQ}

δ((A,Q) b) = (δ((A, b), (Q, b)) ={AR}

δ((A,R) a) = (δ((A, a), (R, a)) ={BQ}

δ((A,R) b) = (δ((A, b), (R, b)) ={AP}

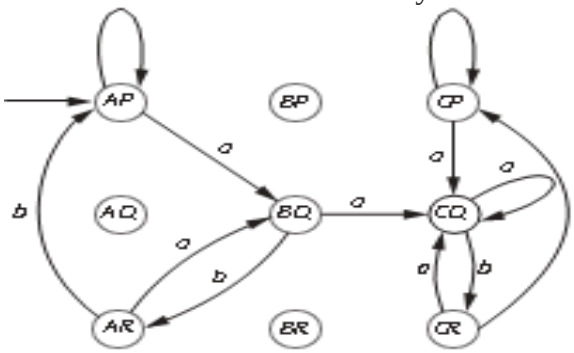
δ((B,P), a) = (δ((B, a), (P, a)) ={CQ}

δ((B,P), b) = (δ((B, b), (P, b)) ={AP}

δ((B,Q), a) = (δ((B, a), (Q, a)) ={CQ}

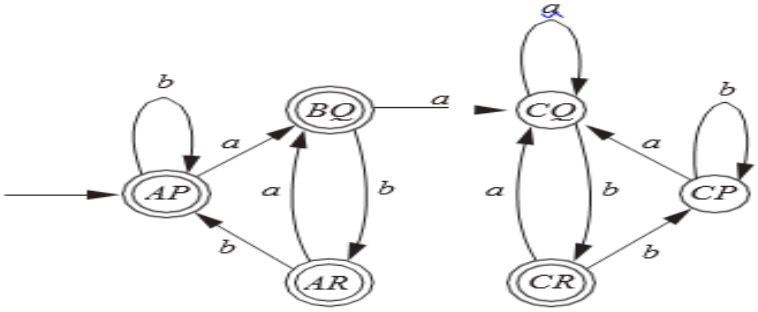
δ((B,Q), b) = (δ((B, b), (Q, b)) ={AR}
δ((B,R), a) = (δ((B, a), (R, a)) ={CQ}
δ((B,R), b) = (δ((B, b), (R, b)) ={AP}
δ((C,P), a) = (δ((C, a), (P, a)) ={CQ}
δ((C,P), b) = (δ((C, b), (P, b)) ={CP}
δ((C,Q), a) = (δ((C, a), (Q, a)) ={CQ}
δ((C,Q), b) = (δ((C, b), (Q, b)) ={CR}
δ((C,R), a) = (δ((C, a), (R, a)) ={CQ}
δ((C,R), b) = (δ((C, b), (R, b)) ={CP}

Rather than drawing all eighteen transitions, we start at the initial state (A, P), draw the two transitions to (B, Q) and (A, P) using the definition of δ in the theorem, and continue in this way, at each step drawing transitions from a state that has already been reached by some other transition. At some point, we have six states such that every transition from one of these six goes to one of these six. Since none of the remaining three states is reachable from any of the first six, we can leave them out.



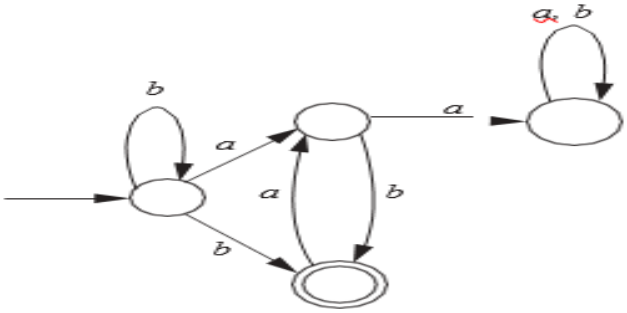
For L1 ∪ L2

If we want our finite automaton to accept L1 ∪ L2, then we designate as accepting states the ordered pairs among the remaining six that involve at least one of the accepting states A, B, and R.



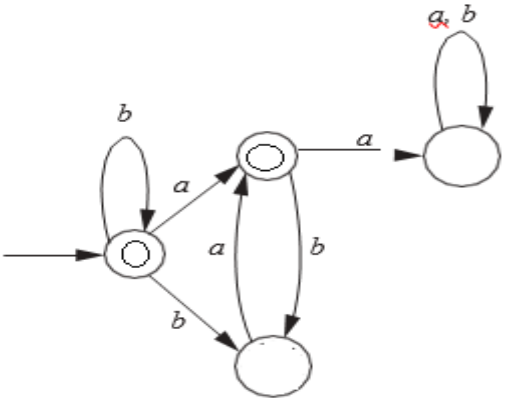
For L1 ∩ L2

If instead we want to accept L1 ∩ L2, then the only accepting state is (A, R), since (B, R) was one of the three omitted.



For L1 - L2,

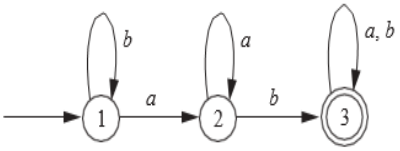
The FA we would get for L1 − L2 is similar and also has just four states, but in that case two are accepting states.



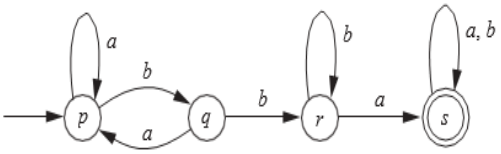
Example2:-

FAs M1 and M2 accepting L1 = {x ∈ {a, b}∗ | x contains the substring ab} and L2 = {x ∈ {a, b}∗ | x contains the substring bba}, respectively

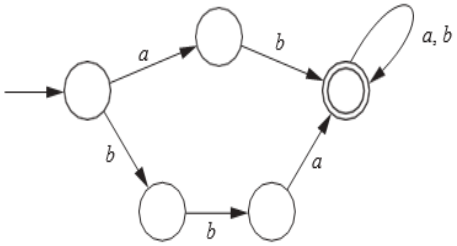
FA for M1 accepting Language L1



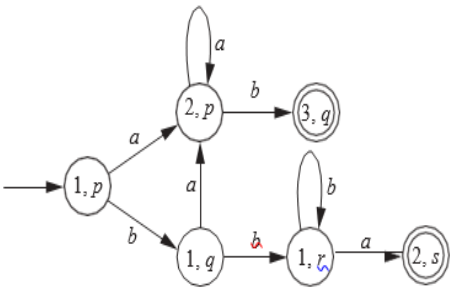
FA for M2 accepting Language L2

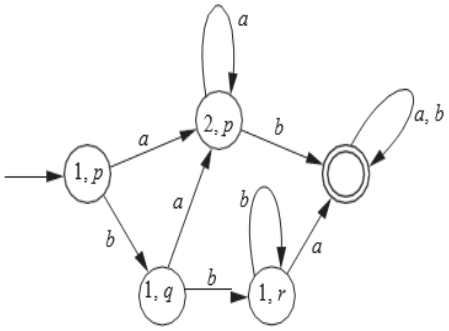


FA for M



FA for M accepting Languages L1 U L2





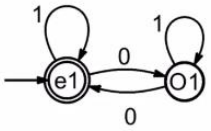
FA for M accepting Languages L1 ∩ L2

Example3:-

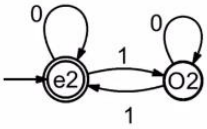
Problem:

The set of strings containing even number of 0's or even number of 1's over Σ = { 0, 1 }

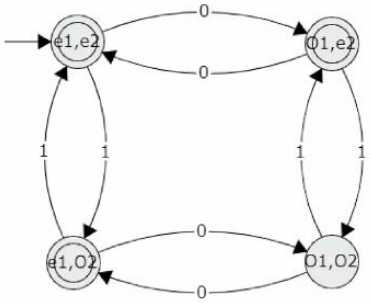
Part 1 :
Set of strings containing even number of 0's and any number of 1's



Part 2:
Set of strings containing even number of 1's and any number of 0's



union

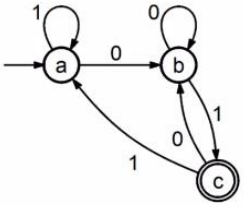


Example4:-

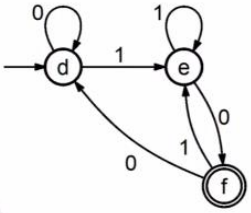
Problem:

The set of strings either ending with 01 or 10 over Σ = { 0, 1 }

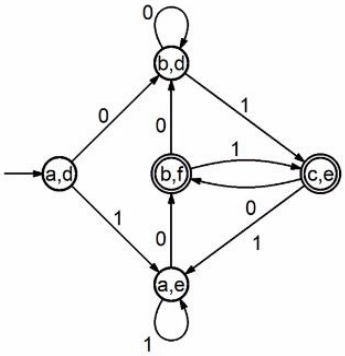
Part 1 :
Set of strings ending with 01



Part 2:
Set of strings ending with 10



union



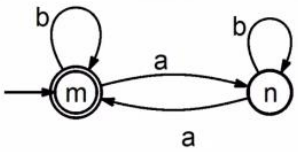
Example5:-

Problem:

L(M) = { w | w has an even number of a's and each a is followed by at least one b }

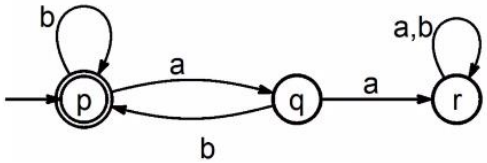
Part 1 :

L(M1) = { w | w has an even number of a's}

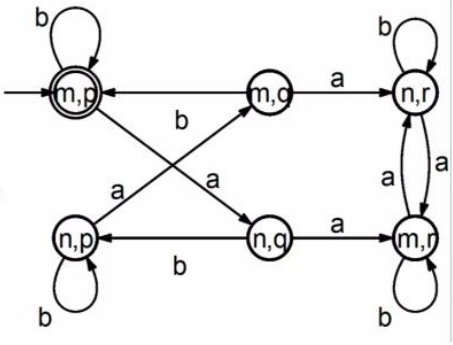


Part 2:

L(M2) = { w | each a in w is followed by at least one b}



intersection



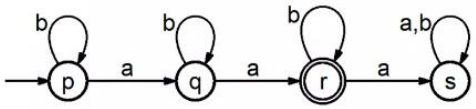
Example6:-

Problem:

L(M) = { w | w has exactly two a's and at least two b's }

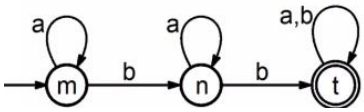
Part 1 :

L(M1) = { w | w has exactly two a's }

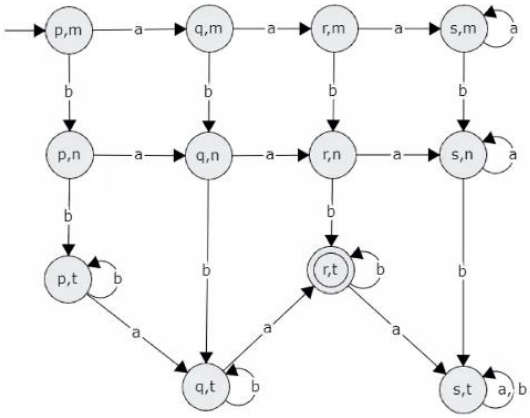


Part 2:

L(M2) = { w | w has at least two b's }



intersection



Regular Languages and Regular Expressions

Introduction

There are 3 specifications of tokens:

1) Strings
2) Language
3) Regular expression

An **alphabet** or **character** class is a finite set of symbols.
A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet.
A **language** is a set of strings from some alphabet (finite or infinite).

In language theory, the terms "sentence" and "word" are often used as synonyms for "string."
The length of a string s, usually written |s|, is the number of occurrences of symbols in s.
For example, banana is a string of length six. The empty string, denoted ε, is the string of length zero.

Operations on strings

The following string-related terms are commonly used:

1. A **prefix** of string s is any string obtained by removing zero or more symbols from the end of strings.
   For example, ban is a prefix of banana.

2. A **suffix** of string s is any string obtained by removing zero or more symbols from the beginning For example, nana is a suffix of banana.

3. A **substring** of s is obtained by deleting any prefix and any suffix from s. For example, nan is a substring of banana.

4. The **proper prefixes, suffixes, and substrings** of a string s are those prefixes, suffixes, and substrings, respectively of s that are not ε or not equal to s itself.

5. A **subsequence** of s is any string formed by deleting zero or more not necessarily consecutive positions of s
   For example, baan is a subsequence of banana.

Operations on languages:

The following are the operations that can be applied to languages:

1. Union
2. Concatenation
3. Kleene closure
4. Positive closure

The following example shows the operations on strings:

Union	: L U S={0,1,a,b,c}
Concatenation	: L.S={0a,1a,0b,1b,0c,1c}
Kleene closure	: L*={ ε,0,1,00….}
Positive closure	: L+={0,1,00….}

Regular languages are formal languages **that regular expressions can describe and can also be recognized by finite automata.** They are used to define sets of strings, such as sequences of characters or words, that follow specific patterns.

They are important in computer science and theoretical computer science because they form a foundation for understanding the theory of computation and the design of compilers and other software tools.

Formally, a regular language can be defined as the collection of all strings that are recognized by a finite automata (FA).

An FA is a 5-tuple (Q, Σ, δ, q0, F), where:

1. Q stands for a finite number of states
2. Σ stands for a finite alphabet, representing the input symbols
3. δ stands for the transition function which maps Q x Σ to Q
4. q0 stands for the initial state. It is one of the elements of Q
5. F stands for the set of final states. F is also a subset of Q.

FAs and regular expressions specify patterns or rules that define a language, such as sequences of characters that must or must not appear in the strings. The words in a regular language must follow the rules specified by the finite automaton or regular expression to be part of the language.

Examples of Regular Languages

Some common examples of regular languages include:

- Binary strings that represent even numbers
- Set of strings that contain exactly two *a*'s
- The set of all binary numbers that are divisible by 3
- The set of all strings that contain the substring "01"

Characteristics of Regular Languages

Regular languages have several useful properties. Let's discuss some of these properties.

1. Closure Properties

Regular languages are closed under union, concatenation, and Kleene star (zero or more repetitions). This means that if two regular languages are combined using one of these operations, the resulting language will also be regular.

- **Union:** Let L1 and L2 be regular languages, then L1 U L2 is a regular language
- **Concatenation:** Let L1 and L2 be regular languages, then L1.L2 is a regular language
- **Kleene Star:** Let L be a regular language, then L* is a regular language

2. Regular Expressions

Regular expressions are a compact and convenient way to define regular languages. They use a set of special characters and operators to represent different types of strings and sets of strings.

Let's consider the language L defined by all strings that consist of an even number of 0's. One way to define this language is by using a regular expression, as follows:

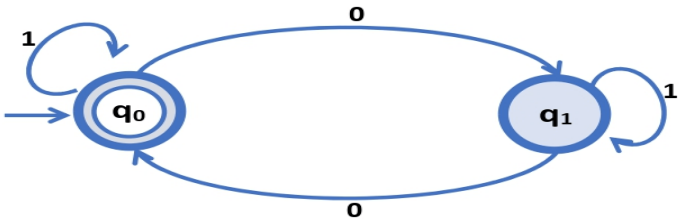
1*(01*01*)*

In this expression, the Kleene star operation "1*" *matches any number of 1's, while the inner expression "*(01*01*)*" ensures the occurrence of an even number of 0's and any number of 1's. . The outer Kleene star operation ensures that the inner expression is repeated any number of times.

3. Equivalence With Finite Automata

A regular language can be recognized by a finite automata, which is a simple machine model consisting of states, transitions, and an initial and final state. Conversely, **every regular language can be expressed using finite automata.**

In the example we considered in section 3.2, an equivalent nondeterministic finite automaton (NFA) is shown next using a finite automata diagram (also known as a state diagram):



The NFA will only accept strings that end in the state q0, corresponding to an even number of 0's in the string.

Practical Applications of Regular Languages

There are many ways regular languages are used in computer science and related fields. A few examples include:

- **Pattern matching:** They are often used in text editors, word processors, and programming languages for searching and manipulating strings that match a given pattern
- **Lexical analysis:** Regular languages are used in the lexical analysis phase of compiler design to identify and tokenize keywords, identifiers, and other elements of a programming language
- **Input validation:** Regular languages are used in programming to validate user input by checking if it matches a given pattern
- **Network protocols:** Regular languages are used to define the syntax of messages in network protocols such as HTTP, FTP, and SMTP
- **DNA sequence analysis:** Regular languages are used to analyze DNA sequences in bioinformatics

Limitations of Regular Languages

Limitations of regular languages include:

- **Less powerful formal language:** Regular languages are a limited class of formal languages and are less powerful than other classes of languages, such as CFLs and context-sensitive languages
- **Unboundedness:** Regular languages are limited to patterns that have a fixed length or can be described by a fixed number of repeating units
- **Expressiveness:** Regular languages are not powerful enough to describe all computable functions or to model all kinds of data structures

REGULAR EXPRESSIONS

The languages accepted by FA are regular languages and these languages are easily described by simple expressions called regular expressions. We have some algebraic notations to represent the regular expressions.

Regular expressions are means to represent certain sets of strings in some algebraic manner and regular expressions describe the language accepted by FA.

If Σ is an alphabet then regular expression(s) over this can be described by following rules.

1. Any symbol from $\Sigma, \in$ *and* ϕ are regular expressions.
2. If r_1 and r_2 are two regular expressions then *union* of these represented as $r_1 \cup r_2$ or $r_1 + r_2$ is also a regular expression
3. If r_1 and r_2 are two regular expressions then *concatenation* of these represented as $r_1 r_2$ is also a regular expression.
4. The Kleene closure of a regular expression r is denoted by r^* is also a regular expression.
5. If r is a regular expression then (r) is also a regular expression.
6. The regular expressions obtained by applying rules 1 to 5 once or more than once are also regular expressions.

Examples :

(1) If $\Sigma = \{a, b\}$, then

(a) a is a regular expression (Using rule 1)
(b) b is a regular expression (Using rule 1)
(c) $a + b$ is a regular expression (Using rule 2)
(d) b^* is a regular expression (Using rule 4)
(e) ab is a regular expression (Using rule 3)
(f) $ab + b^*$ is a regular expression (Using rule 6)

(2) Find regular expression for the following

(a) A language consists of all the words over $\{a, b\}$ ending in b.

(b) A language consists of all the words over $\{a, b\}$ ending in bb.

(c) A language consists of all the words over $\{a, b\}$ starting with a and ending in b.

(d) A language consists of all the words over $\{a, b\}$ having bb as a substring.

(e) A language consists of all the words over $\{a, b\}$ ending in aab.

Solution : Let $\Sigma = \{a, b\}$, and

All the words over $\Sigma = \{\in, a, b, aa, bb, ab, ba, aaa, \ldots\} = \Sigma^*$ *or* $(a + b)^*$ *or* $(a \cup b)^*$

(a) Regular expression for the given language is $(a + b) * b$

(b) Regular expression for the given language is $(a + b) * bb$

(c) Regular expression for the given language is $a (a + b) * b$

(d) Regular expression for the given language is $(a + b) * aa$ or $aa (a + b) *$ or $(a + b) * bb (a + b) *$

(e) Regular expression for the given language is $(a + b) * aab$

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

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

TABLE: Meaning of regular expressions

Example 4 : Obtain a regular expression to accept strings of a's and b's ending with 'b' and has no substring aa.

Solution :

Note : The statement "strings of a's and b's ending with 'b' and has no substring aa" can be restated as "string made up of either b or ab". Note that if we state something like this, the substring aa will never occur in the string and the string ends with 'b'. So, the regular expression can be of the form

$$(b + ab)^*$$

But, because of * closure, even null string is also included. But, the string should end with 'b'. So, instead of * closure, we can use positive closure '+'. So, the regular expression to accept strings of a's and b's ending with 'b' and has no substring aa can be written as

$$(b+ab)^+$$

The above regular expression can also be written as

$$(b+ab)\,(b+ab)^*$$

Note : Using the set notation this regular expression can be written as

$$L(r)=\{(b+ab)^n | n\geq 1\}$$

Example 5 : Obtain a regular expression to accept strings of 0's and 1's having no two consecutive zeros.

Solution :

The first observation from the statement is that whenever a 0 occurs it should be followed by 1. But, there is no restriction on the number of 1's. So , it is a string consisting of any combination of 1's and 01's. So, the partial regular expression for this can be of the form

$$(1+01)^*$$

No doubt that the above expression is correct. But, suppose the string ends with a 0. What to do? For this, the string obtained from above regular expression may end with 0 or may end with $\in$ (i. e., may not end with 0). So, the above regular expression can be written as

$$(1+01)^*(0+\in)$$

Example 6 : Obtain a regular expression to accept strings of a's and b's of length ≤ 10.

Solution :

The regular expression for this can be written as

$$\in + a + b + aa + ab + ba + bb + \ldots\ldots + bbbbbbbbba + bbbbbbbbbb$$

But, using ..... in a regular expression is not recommended and so we can write the above expression as

$$(\in + a + b)^{10}$$

Example 7 : Obtain a regular expression to accept strings of a's and b's starting with 'a' and ending with 'b'.

Solution :

Strings of a's and b's of arbitrary length can be written as $(a+b)^*$

But, this should start with 'a' and end with 'b'. So, the regular expression can be written as

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

Hierarchy of Evaluation of Regular Expressions

We follow the following order when we evaluate a regular expression.

1. Parenthesis
2. Kleene closure
3. Concatenation
4. Union

Example 1: Consider the regular expression $(a+b)^*aab$ and describe the all words represented by this.

Solution :

$(a+b)^*aab$ = {All words over $\{a,b\}$}aab (Evaluating $(a+b)^*$ first)

= $\{\in, a, b, aa, bb, ab, ba, aaa, \ldots\}\, aab$

= {All words over $\{a, b\}$ ending in aab}

Example 2: Consider the regular expression $(a^* + b^*)^*$ and explain it.

Solution : We evaluate a^* and b^* first then $(a^* + b^*)^*$.

$(a^* + b^*)^*$ = (All the words over $\{a\}$ + all the words over $\{b\}$)*

= $(\{\in, a, aa, \ldots\}$ or $\{\in, b, bb, \ldots\})^*$

= $(\{\in, a, b, aa, bb, \ldots\})^*$

= $\{\in, a, b, aa, bb, ab, ba, aaa, bbb, abb, baa, aabb, \ldots\}$

= {All the words over $\{a, b\}$}

= $(a+b)^*$

So, $(a^* + b^*)^* = (a+b)^*$

RELATIONSHIP BETWEEN FA AND RE

There is a close relationship between a finite automata and the regular expression we can show this relation in below figure.

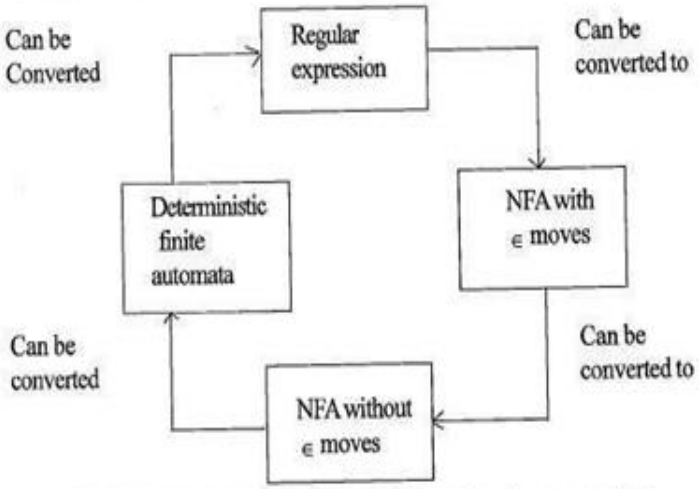


FIGURE : Relationship between FA and regular expression

The above figure shows that it is convenient to convert the regular expression to NFA with $\in$ moves. Let us see the theorem based on this conversion.

Conversion from Regular Expression to NFA

CONSTRUCTING FA FOR A GIVEN REs

Theorem : If r be a regular expression then there exists a NFA with $\in$ - moves, which accepts $L(r)$.

Proof : First we will discuss the construction of NFA M with $\in$ - moves for regular expression r and then we prove that $L(M) = L(r)$.

Let r be the regular expression over the alphabet Σ.

Construction of NFA with $\in$ - moves

Case 1 :

(i) $r = \phi$

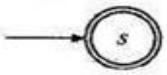
NFA $M = (\{s, f\}, \{\ \}, \delta, s, \{f\})$ as shown in Figure1 (a)

(No path from initial state s to reach the final state f.)

Figure 1 (a)

(ii) $r = \in$

NFA $M = (\{s\}, \{\ \}, \delta, s, \{s\})$ as shown in Figure 1 (b)

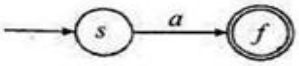


(The initial state s is the final state)

Figure 1 (b)

(iii) $r = a$, for all $a \in \Sigma$,

NFA $M = (\{s, f\}, \Sigma, \delta, s, \{f\})$



(One path is there from initial state s to reach the final state f with label a.)

Figure 1 (c)

Case 2 : $|r| \geq 1$

Let r_1 and r_2 be the two regular expressions over Σ_1, Σ_2 and N_1 and N_2 are two NFA for r_1 and r_2 respectively as shown in Figure 2 (a).

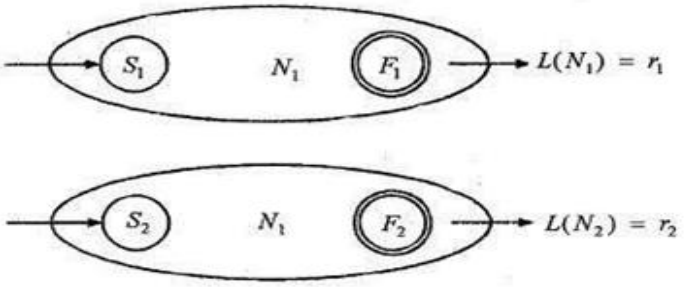


Figure 2 (a) NFA for regular expression r_1 and r_2

Rule 1 : For constructing NFA M for $\mathbf{r} = \mathbf{r}_1 + \mathbf{r}_2$ or $\mathbf{r}_1 \cup \mathbf{r}_2$

Let s and f are the starting state and final state respectively of M.

Transition diagram of M is shown in Figure 2 (b).

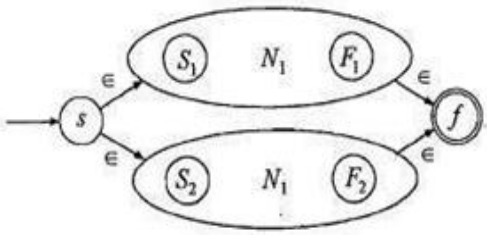


Figure 2 (b) NFA for regular expression $r_1 + r_2$

$$L(M) = \in L(N_1) \in \text{ or } \in L(N_2) \in$$
$$= L(N_1) \text{ or } L(N_2) = r_1 \text{ or } r_2$$

So, $r = r_1 + r_2$

$M = (Q, \Sigma_1 \cup \Sigma_2, \delta, s, \{f\})$, where Q contains all the states of N_1 and N_2.

Rule 2 : For regular expression $\mathbf{r} = \mathbf{r}_1\mathbf{r}_2$, NFA M is shown in Figure2 (c).

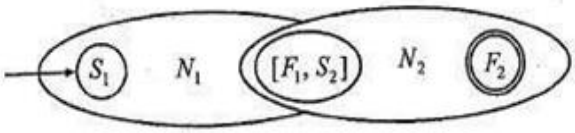


Figure 2 (c) NFA for regular expression r_1r_2

The final state (s) of N_1 is merged with initial state of N_2 into one state $[F_1S_2]$ as shown above in Figure2 (c).

$$L(M) = L(N_1) \text{ followed } L(N_2)$$
$$= L(N_1)\ L(N_2) = r_1r_2$$

So, $r = r_1r_2$

$M = (Q, \Sigma_1 \cup \Sigma_2, \delta, S_1, \{F_2\})$, where Q contains all the states of N_1 and N_2 such that final state(s) of N_1 is merged with initial state of N_2.

Rule 3 : For regular expression $\mathbf{r} = \mathbf{r_1^*}$, NFA M is shown in Figure2 (d)

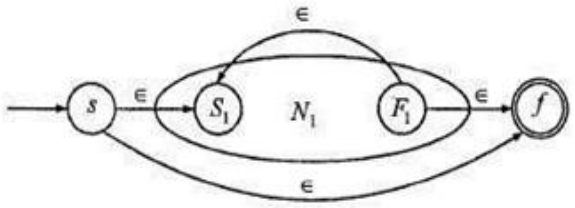


Figure 2 (d) NFA for regular expression for r_1^*

$$L(M) = \{\in, L(N_1), L(N_1)\ L(N_1), L(N_1)L(N_1)L(N_1), \ldots\}$$
$$= L(N_1)^*$$
$$= r_1^*$$

$M = (\{s, f\} \cup Q_1, \Sigma_1, \delta, s, \{f\})$, where Q_1 is the set of states of N_1.

Rule 4 : For construction of NFA M for $r = r_1^+$, M is shown in Figure 2 (e).

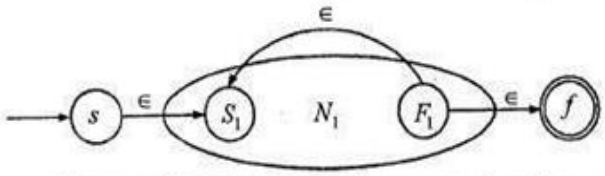


Figure 2(e) NFA for regular expression for r_1^+

$$L(M) = \{L(N_1), L(N_1)L(N_1), L(N_1)L(N_1)L(N_1), \ldots\}$$
$$= L(N_1)^+ = r_1^+$$

$M = (\{s, f\} \cup Q_1, \Sigma_1, \delta, s, \{f\})$, where Q_1 is the set of states of N_1.

Example 1 : Construct NFA for the regular expression a + ba *.

Solution : The regular expression

$r = a + ba*$ can be broken into r_1 and r_2 as

$r_1 = a$

$r_2 = ba*$

Let us draw the NFA for r_1, which is very simple.

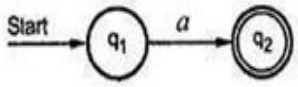


FIGURE 1: For r_1

Now, we will go for $r_2 = ba*$, this can be broken into r_3 and r_4 where $r_3 = b$ and $r_4 = a*$. Now the case for concatenation will be applied. The NFA will look like this r_3 will be shown in figure2.

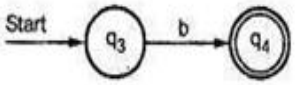


FIGURE 2: For r_3

and r_4 will be shown as

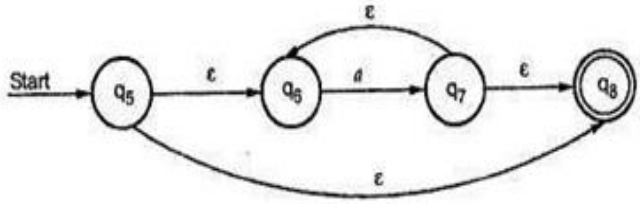


FIGURE 3 : For r_4

The r_2 will be $r_2 = r_3.r_4$

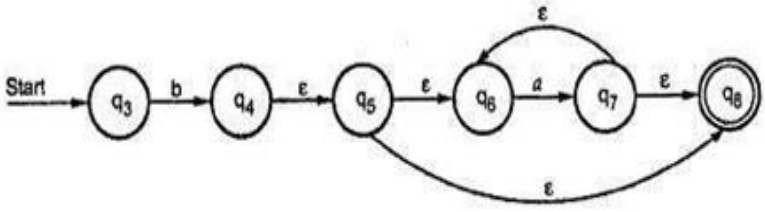


FIGURE 4 : For r_2

Now, we will draw NFA for $r = r_1 + r_2$ i. e. $a + ba^*$

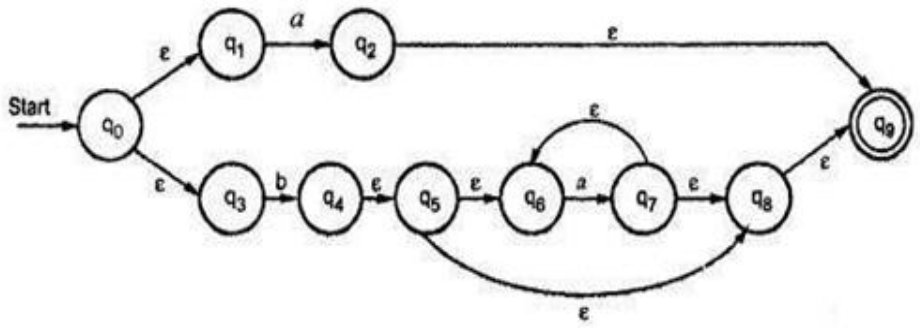


FIGURE 5 : NFA for $r = r_1 + r_2$ i. e. $a + ba^*$

Example 2 : Construct NFA with $\in$ moves for the regular expression $(0+1)^*$.

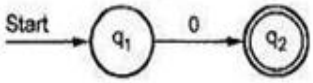
Solution : The NFA will be constructed step by step by breaking regular expression into small regular expressions.

$$r_3 = (r_1 + r_2)$$

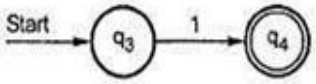
$$r = r_3^*$$

where $r_1 = 0,\ r_2 = 1$

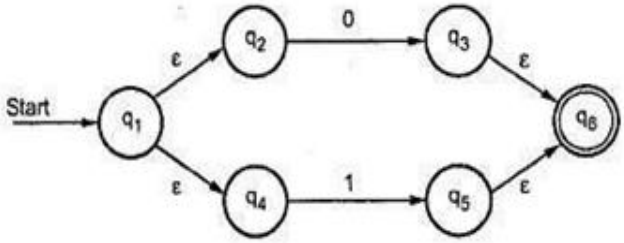
NFA for r_1 will be



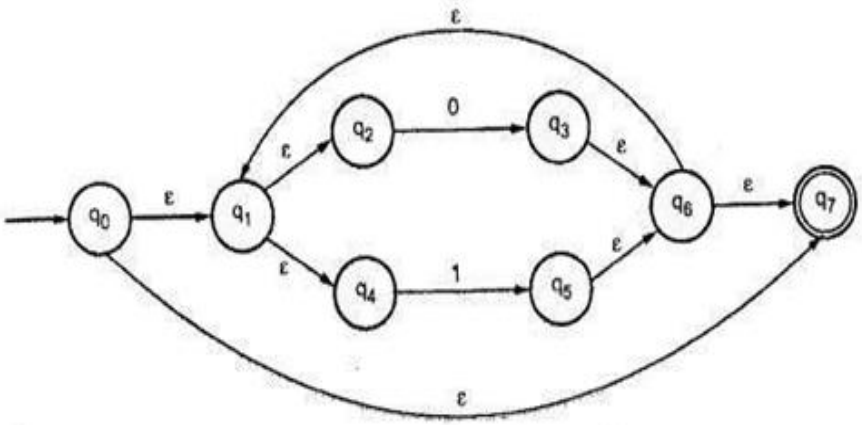
NFA for r_2 will be



NFA for r_3 will be



And finally



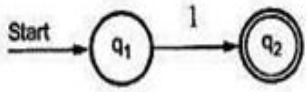
Example 3 : Construct NFA for the language having odd number of one's over the set $\Sigma = \{1\}$.

Solution : In this problem language L is given, we have to first convert it to regular expression. The r. e. for this L is written as r.e. = 1 (11)*.

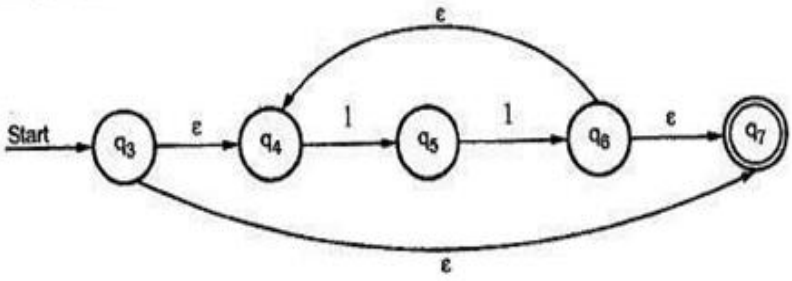
The r is now written as

$$r = r_1 \; r_2$$

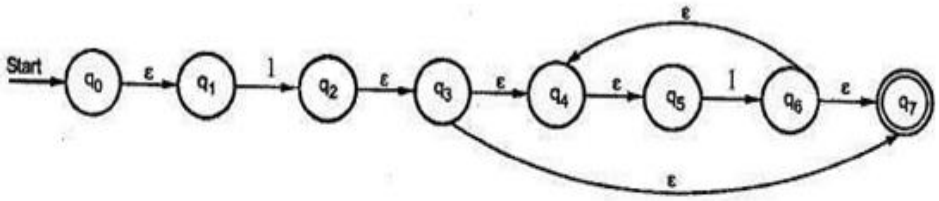
NFA for $r_1 = 1$ is



NFA for $r_2 = (11)^*$



The final NFA is



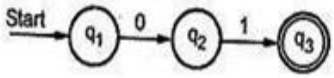
Example 4 : Construct NFA for the r. e. $(01+2^*)0$.

Solution : Let us design NFA for the regular expression by dividing the expression into smaller units

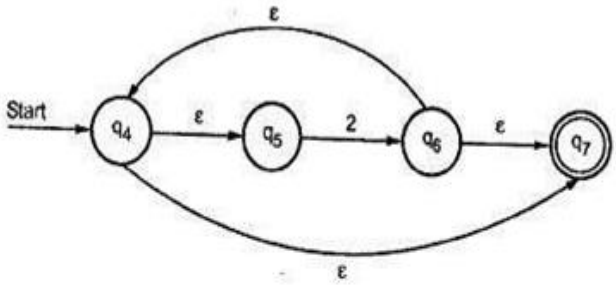
$$r=(r_1+r_2)r_3$$

where $r_1=01$, $r_2=2^*$ and $r_3=0$

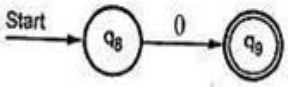
The NFA for r_1 will be



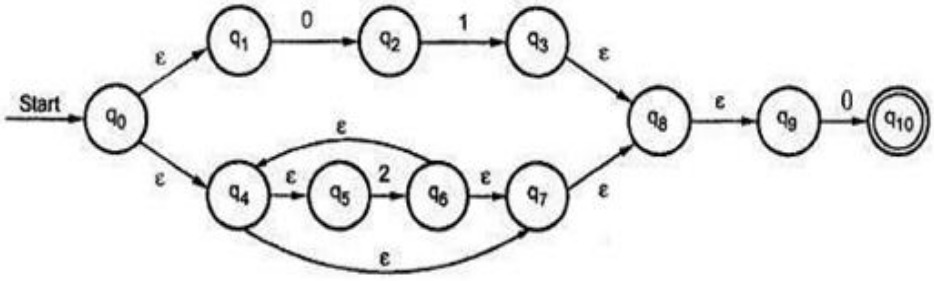
The NFA for r_2 will be



The NFA for r_3 will be



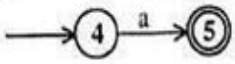
The final NFA will be



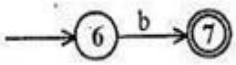
Example 5 : Obtain an NFA which accepts strings of a's and b's starting with the string ab.

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

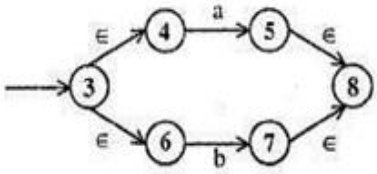
Step 1 : The machine to accept 'a' is shown below.



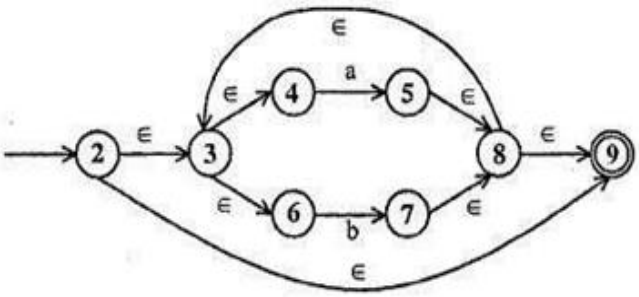
Step 2 : The machine to accept 'b' is shown below.



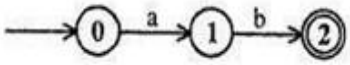
Step 3 : The machine to accept (a + b) is shown below.



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



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



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

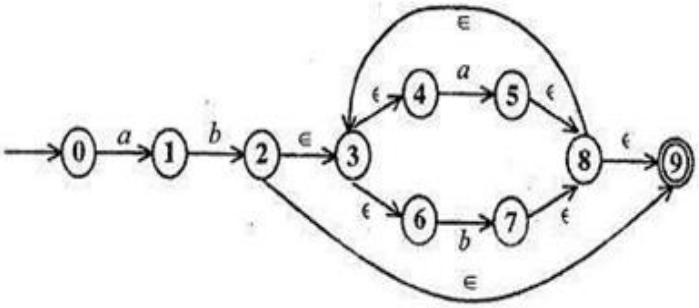
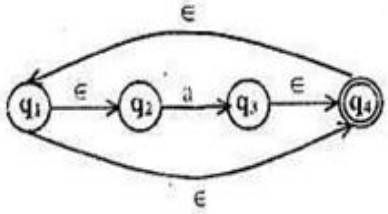


FIGURE : To accept the language (ab (a+b)*)

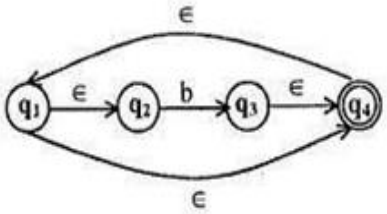
Example 6: Obtain an NFA for the regular expression $a^* + b^* + c^*$

Solution :

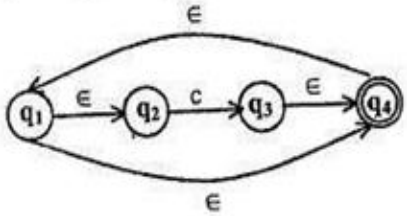
The machine corresponding the regular expression a* can be written as



The machine corresponding the regular expression b* can be written as



The machine corresponding the regular expression c* can be written as



The machine corresponding the regular expression $a^* + b^* + c^*$ is shown in below figure.

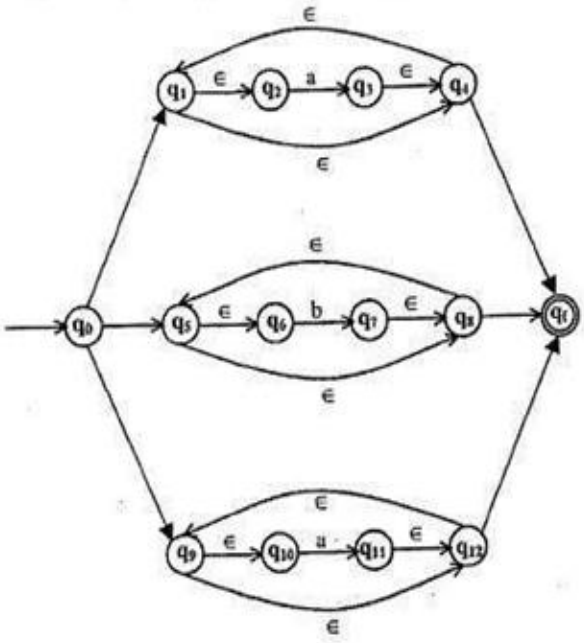
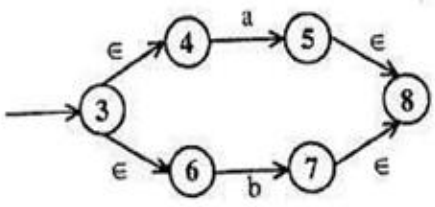


FIGURE: To accept the language $(a^* + b^* + c^*)$

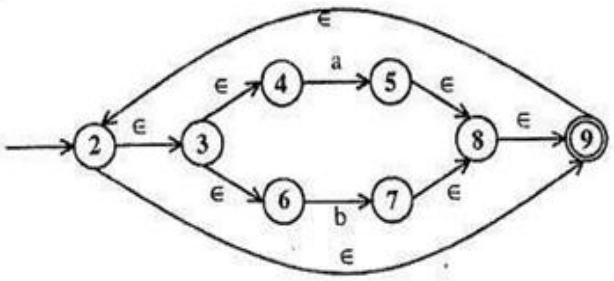
Example 7 : Obtain an NFA for the regular expression $(a+b)^* aa(a+b)^*$

Solution :

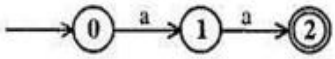
Step 1 : The machine to accept (a + b) is shown below.



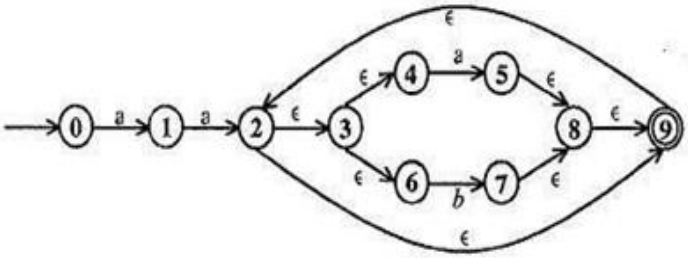
Step 2 : The machine to accept (a + b) * is shown below.



Step 3 : The machine to accept aa is shown below.



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



Step 5 : The machine to accept (a + b) * aa (a + b)* is shown in below figure.

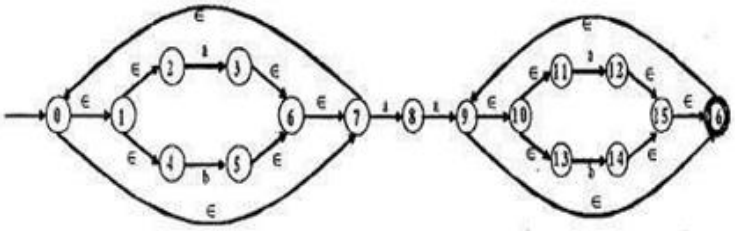
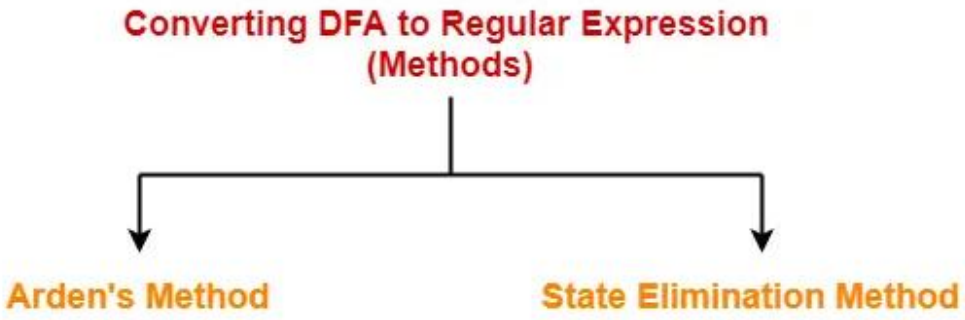


FIGURE : NFA to accept (a + b)* aa (a + b)*

CONVERTING DFA TO REGULAR EXPRESSIONS

The two popular methods for converting a given DFA to its regular expression are-



Arden's Theorem-

Arden's Theorem is popularly used to convert a given DFA to its regular expression.

It states that-
Let P and Q be two regular expressions over ∑.
If P does not contain a null string ∈, then-

R = Q + RP has a unique solution i.e. R = QP*

To use Arden's Theorem, following conditions must be satisfied-

- The transition diagram must not have any ∈ transitions.
- There must be only a single initial state.

Steps-

To convert a given DFA to its regular expression using Arden's Theorem, following steps are followed-

Step-01:

- Form a equation for each state considering the transitions which comes towards that state.
- Add '∈' in the equation of initial state.

Step-02:

Bring final state in the form R = Q + RP to get the required regular expression.

Note-01:

Arden's Theorem can be used to find a regular expression for both DFA and NFA.

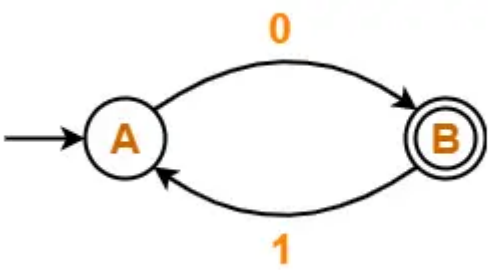
Note-02:

If there exists multiple final states, then-

- Write a regular expression for each final state separately.
- Add all the regular expressions to get the final regular expression.

Problem-01:

Find regular expression for the following DFA using Arden's Theorem-



Solution-

Step-01:

Form a equation for each state-

A = ∈ + B.1 ……(1)

B = A.0 ……(2)

Step-02:

Bring final state in the form R = Q + RP.

Using (1) in (2), we get-

B = (∈ + B.1).0

B = ∈.0 + B.1.0

B = 0 + B.(1.0) ……(3)

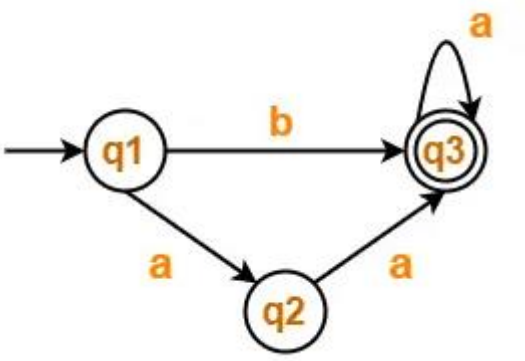
Using Arden's Theorem in (3), we get-

B = 0.(1.0)*

Thus, Regular Expression for the given DFA = 0(10)*

Problem-02:

Find regular expression for the following DFA using Arden's Theorem-



Solution-

Step-01:

Form a equation for each state-

q1 = ∈ ……(1)
q2 = q1.a ……(2)
q3 = q1.b + q2.a + q3.a …….(3)

Step-02:

Bring final state in the form R = Q + RP.

Using (1) in (2), we get-
q2 = ∈.a
q2 = a …….(4)

Using (1) and (4) in (3), we get-

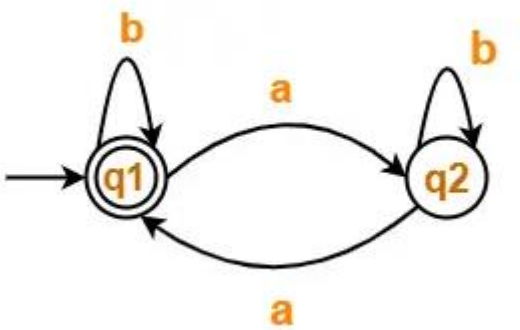
q3 = q1.b + q2.a + q3.a
q3 = ∈.b + a.a + q3.a
q3 = (b + a.a) + q3.a …….(5)

Using Arden's Theorem in (5), we get-
q3 = (b + a.a)a*

Thus, Regular Expression for the given DFA = (b + aa)a*

Problem-03:

Find regular expression for the following DFA using Arden's Theorem-



Solution-

Step-01:

Form a equation for each state-

q1 = ∈ + q1.b + q2.a ……(1)
q2 = q1.a + q2.b ……(2)

Step-02:

Bring final state in the form R = Q + RP.

Using Arden's Theorem in (2), we get-
$q2 = q1.a.b^*$ …….(3)

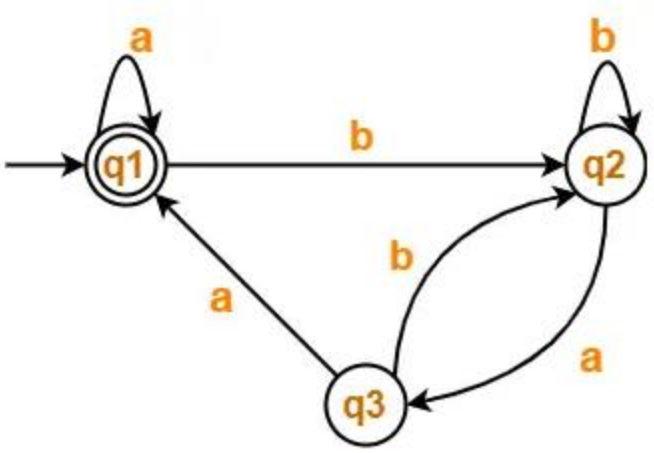
Using (3) in (1), we get-
$q1 = \in + q1.b + q1.a.b^*.a$
$q1 = \in + q1.(b + a.b^*.a)$ …….(4)

Using Arden's Theorem in (4), we get-
$q1 = \in.(b + a.b^*.a)^*$
$q1 = (b + a.b^*.a)^*$

Thus, Regular Expression for the given DFA = $(b + a.b^*.a)^*$

Problem-04:

Find regular expression for the following DFA using Arden's Theorem-



Solution-

Step-01:

Form a equation for each state-
$q1 = \in + q1.a + q3.a$ ……(1)
$q2 = q1.b + q2.b + q3.b$ ……(2)
$q3 = q2.a$ …….(3)

Step-02:

Bring final state in the form R = Q + RP.

Using (3) in (2), we get-
$q2 = q1.b + q2.b + q2.a.b$
$q2 = q1.b + q2.(b + a.b)$ …….(4)

Using Arden's Theorem in (4), we get-
$q2 = q1.b.(b + a.b)^*$ …….(5)

Using (5) in (3), we get-
$q3 = q1.b.(b + a.b)^*.a$ …….(6)

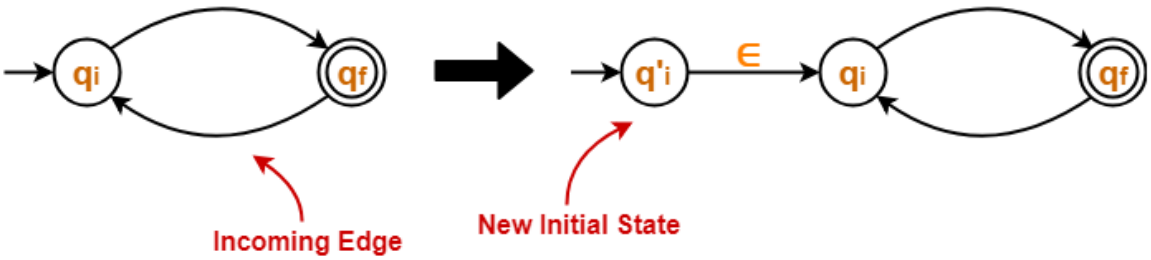
Using (6) in (1), we get-
$q1 = \in + q1.a + q1.b.(b + a.b)^*.a.a$
$q1 = \in + q1.(a + b.(b + a.b)^*.a.a)$ …….(7)

Using Arden's Theorem in (7), we get-
$q1 = \in.(a + b.(b + a.b)^*.a.a)^*$
$q1 = (a + b.(b + a.b)^*.a.a)^*$

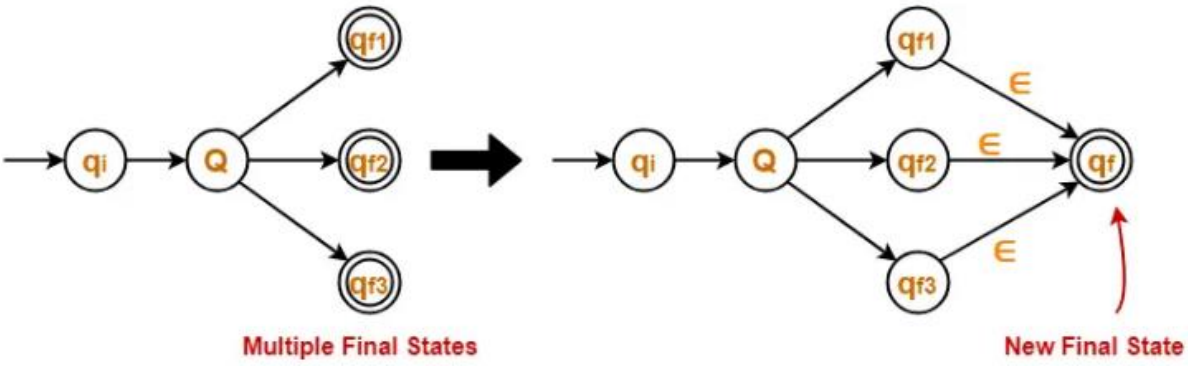
Thus, Regular Expression for the given DFA = (a + b(b + ab)*aa)*

State Elimination Method

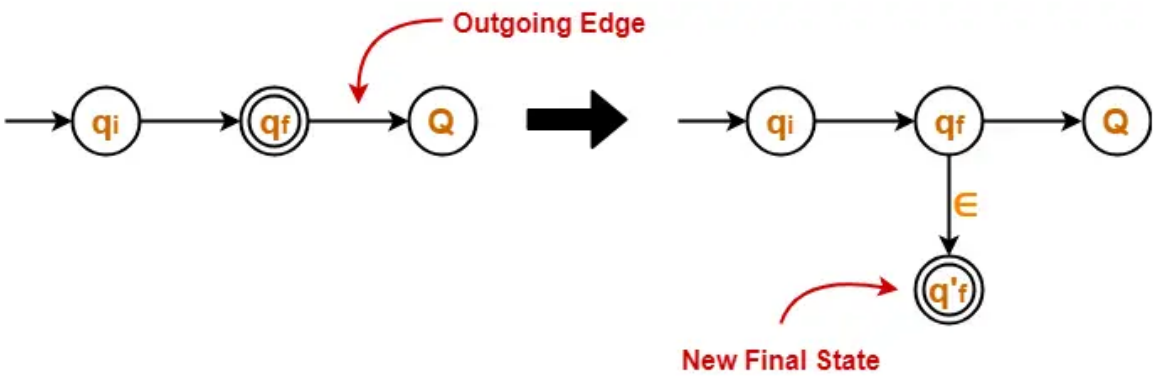
Let's consider the state elimination method to convert FA to RE.
Rules
The rules for state elimination method are as follows −
Rule 1
The initial state of DFA must not have any incoming edge.
If there is any incoming edge to the initial edge, then create a new initial state having no incoming edge to it.



Rule 2
There must exist only one final state in DFA.
If there exist multiple final states, then convert all the final states into non-final states and create a new single final state.



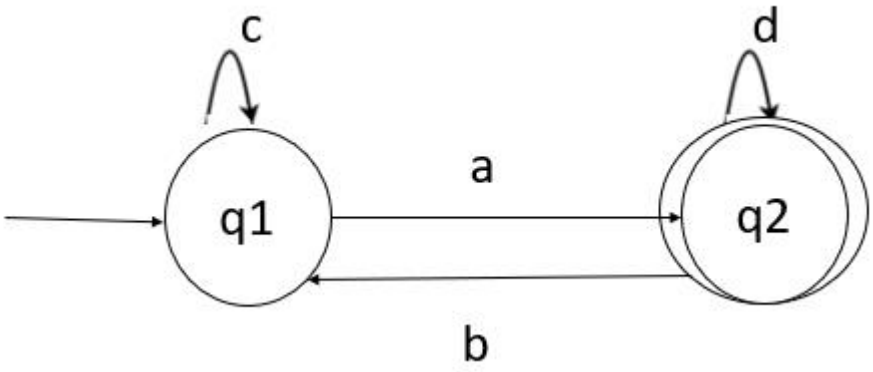
Rule 3
The final state of DFA must not have any outgoing edge.
If this exists, then create a new final state having no outgoing edge from it.



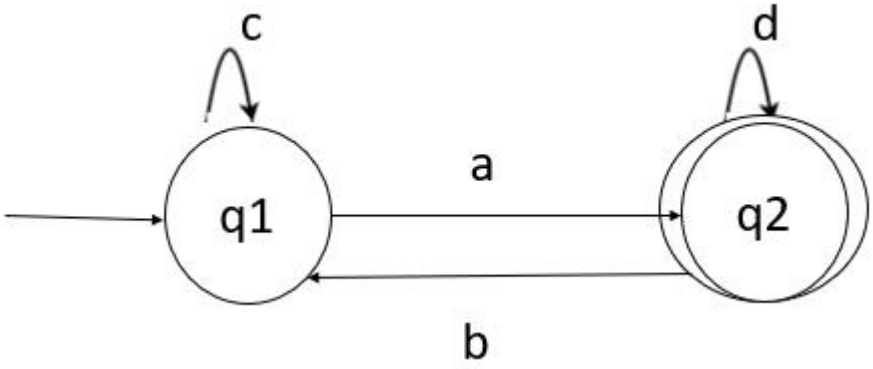
Rule 4
Eliminate all intermediate states one by one.
Now, apply these rules to convert the FA to RE easily.

NOTE:- The state elimination method can be applied to any finite automata.(NFA, ∈-NFA, DFA etc)

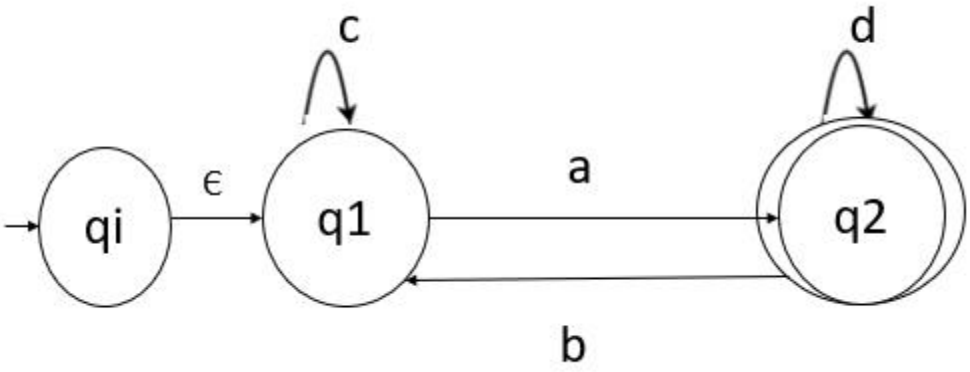
Example:-
Convert the given finite automata (FA) into regular expression (RE).



The given FA is as follows −

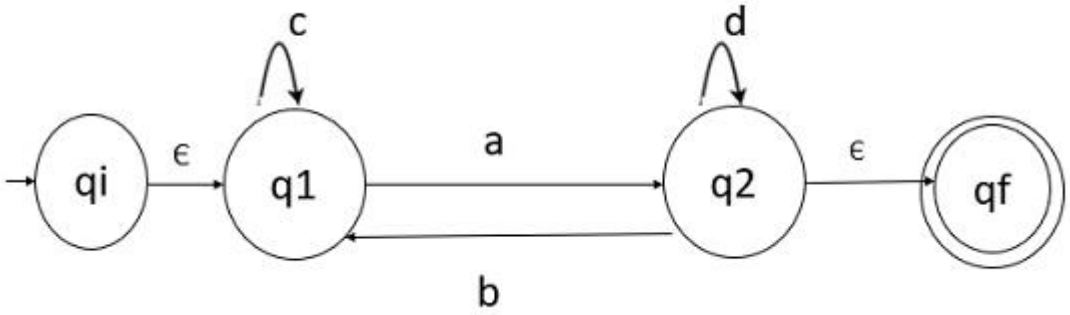


Step 1
Initial state q1 has an incoming edge so create a new initial state qi.



Step 2

Final state q2 has an outgoing edge. So, create a new final state qf.



Step 3

Start eliminating intermediate states

First eliminate q1

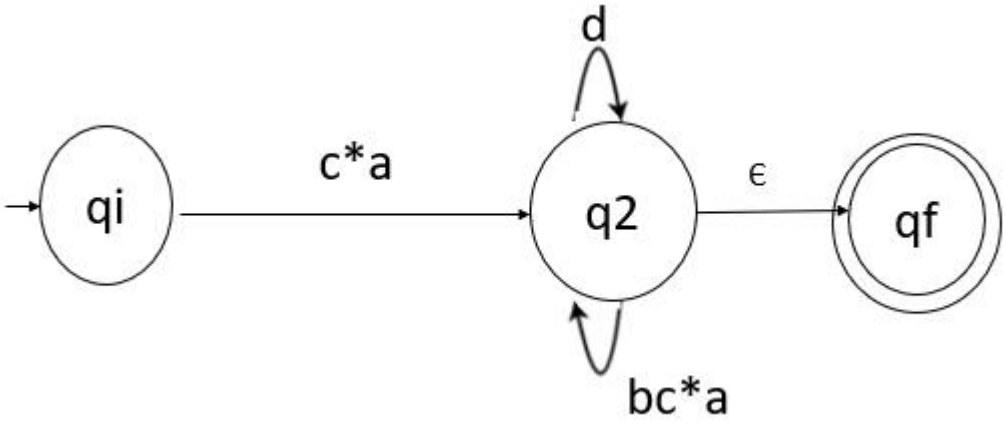
There is a path going from qi to q2 via q1. So, after eliminating q1 we can connect a direct path from qi to q2 having cost.

εc*a=c*a

There is a loop on q2 using state qi. So, after eliminating q1 we put a direct loop to q2 having cost.

b.c*.a=bc*a

After eliminating q1, the FA looks like following −



Second eliminate q2

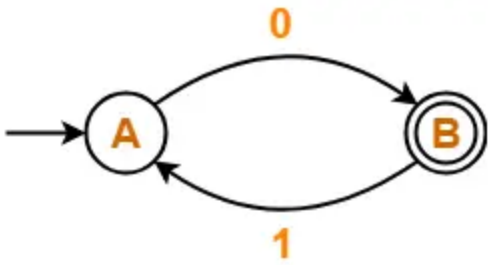
There is a direct path from qi to qf so, we can directly eliminate q2 having cost −

C*a(d+bc*a)* ε = c*a(d+bc*a)*

Which is our final regular expression for given finite automata

Problem-01:

Find regular expression for the following DFA-

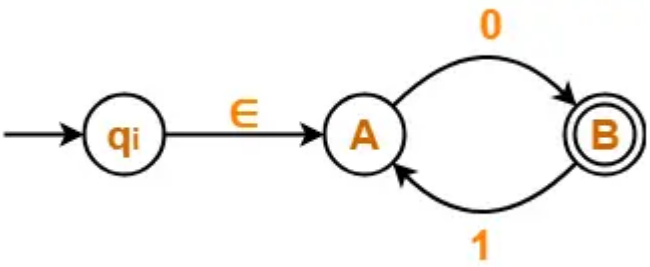


Solution-

Step-01:

Initial state A has an incoming edge.
So, we create a new initial state q_i.

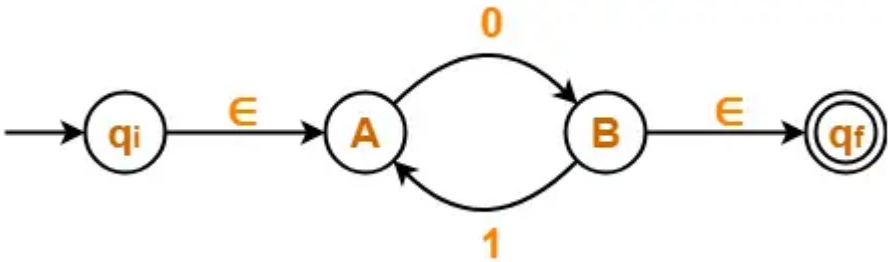
The resulting DFA is-



Step-02:

Final state B has an outgoing edge.
So, we create a new final state q_f

The resulting DFA is-



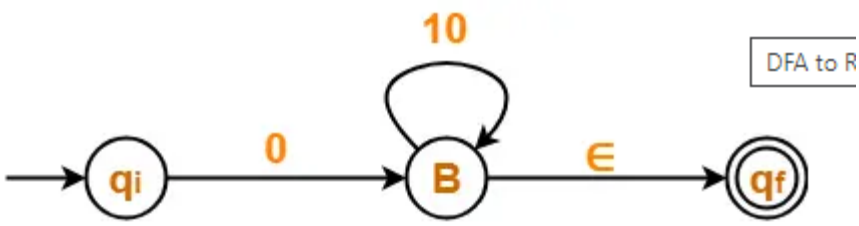
Step-03:

Now, we start eliminating the intermediate states.

First, let us eliminate state A.

There is a path going from state q_i to state B via state A.
So, after eliminating state A, we put a direct path from state q_i to state B having cost $\in.0 = 0$
There is a loop on state B using state A.
So, after eliminating state A, we put a direct loop on state B having cost 1.0 = 10.

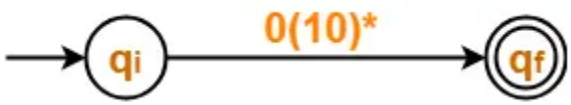
Eliminating state A, we get-



Step-04:

Now, let us eliminate state B.
There is a path going from state q_i to state q_f via state B.
So, after eliminating state B, we put a direct path from state q_i to state q_f having cost $0.(10)^*.\in = 0(10)^*$

Eliminating state B, we get-



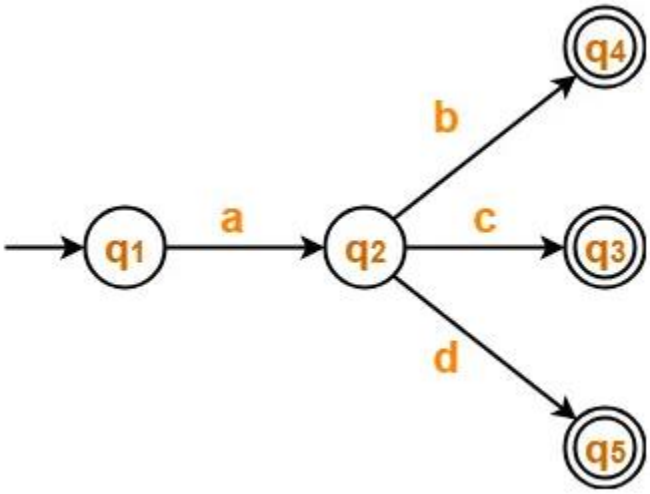
Regular Expression = 0(10)*

NOTE-

In the above question,
If we first eliminate state B and then state A, then regular expression would be = (01)*0.
This is also the same and correct.

Problem-02:

Find regular expression for the following DFA-

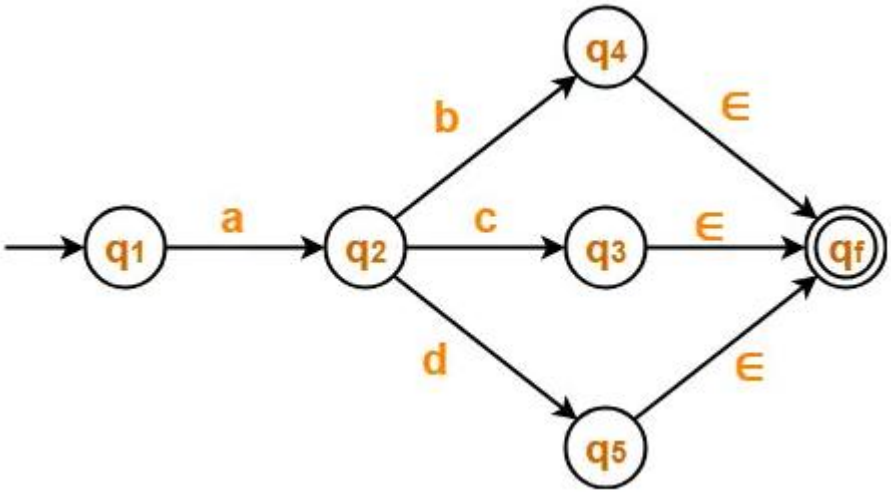


Solution-

Step-01:

There exist multiple final states.
So, we convert them into a single final state.

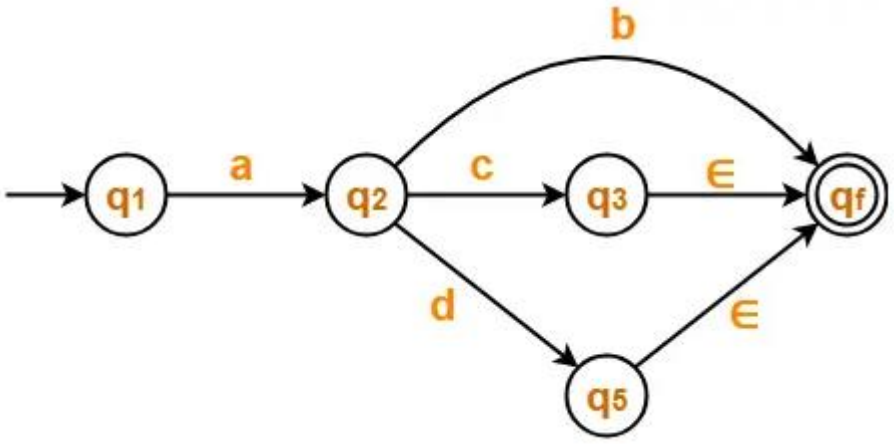
The resulting DFA is-



Step-02:

Now, we start eliminating the intermediate states.

First, let us eliminate state q_4.
There is a path going from state q_2 to state q_f via state q_4.
So, after eliminating state q_4 , we put a direct path from state q_2 to state q_f having cost $b.\in = b$.

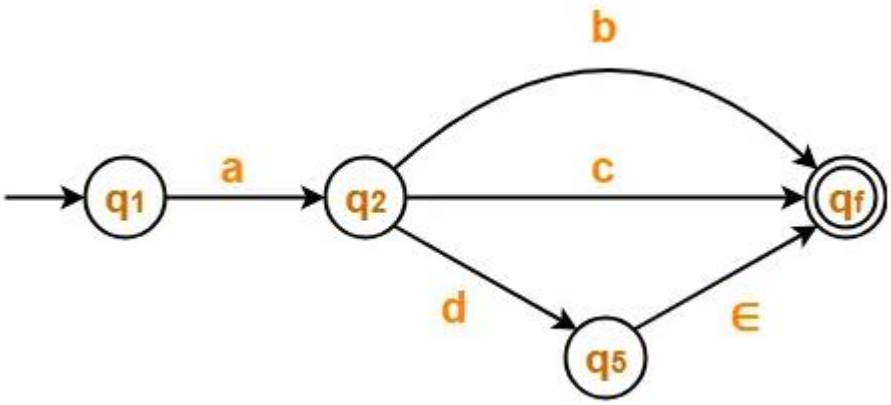


Step-03:

Now, let us eliminate state q_3.

There is a path going from state q_2 to state q_f via state q_3.

So, after eliminating state q_3 , we put a direct path from state q_2 to state q_f having cost $c.\in = c$.

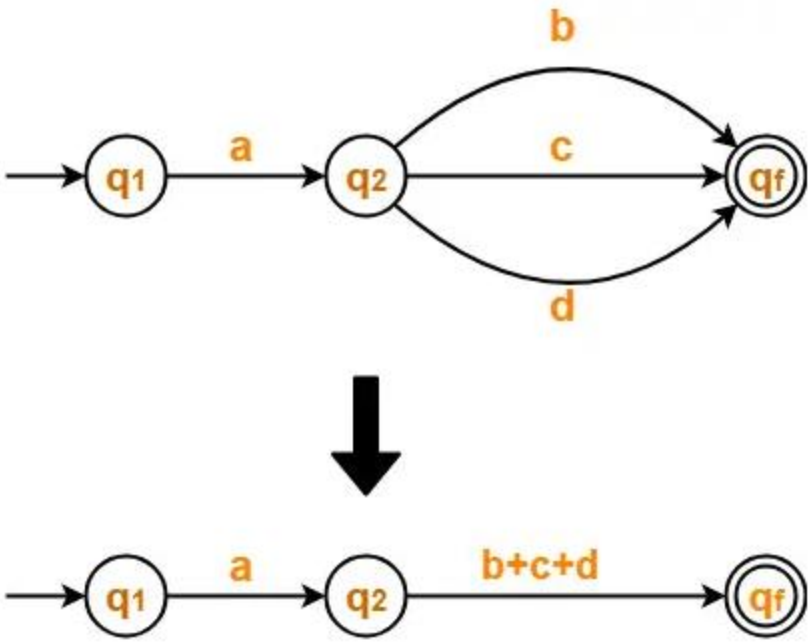


Step-04:

Now, let us eliminate state q_5.

There is a path going from state q_2 to state q_f via state q_5.

So, after eliminating state q_5 , we put a direct path from state q_2 to state q_f having cost $d.\in = d$.

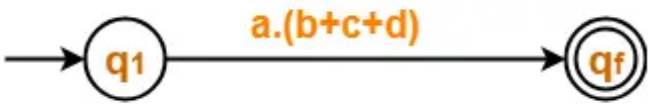


Step-05:

Now, let us eliminate state q_2.

There is a path going from state q_1 to state q_f via state q_2.

So, after eliminating state q_2 , we put a direct path from state q_1 to state q_f having cost a.(b+c+d).

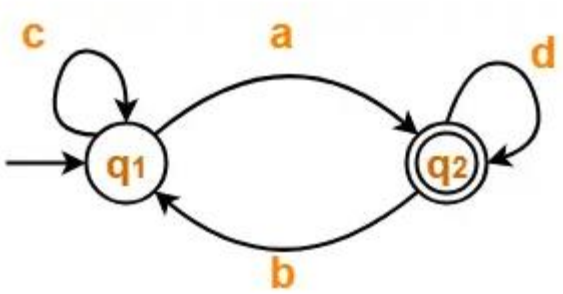


From here,

Regular Expression = a(b+c+d)

Problem-03:

Find regular expression for the following DFA-

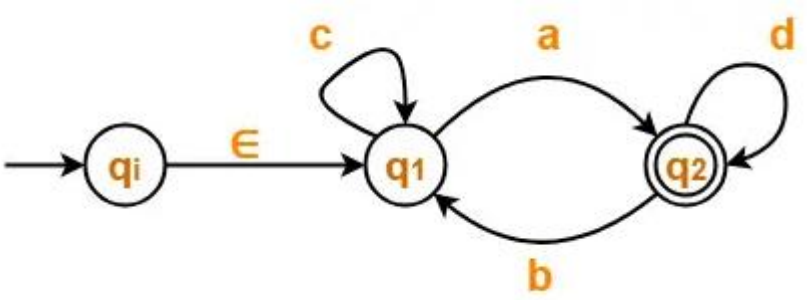


Solution-

Step-01:

Initial state q_1 has an incoming edge.
So, we create a new initial state q_i.

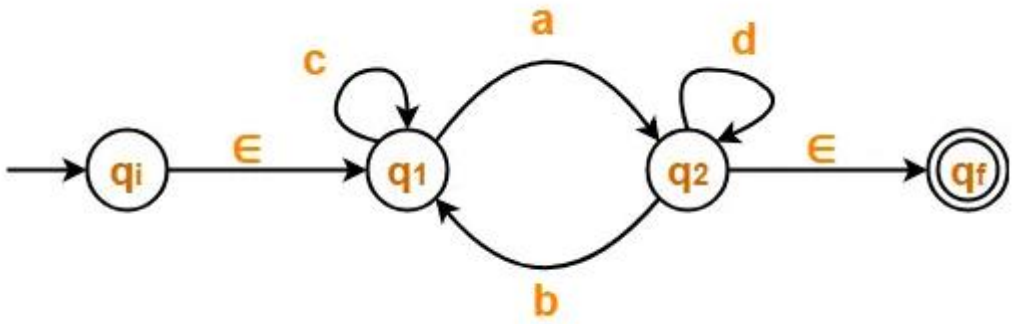
The resulting DFA is-



Step-02:

Final state q_2 has an outgoing edge.
So, we create a new final state q_f.

The resulting DFA is-

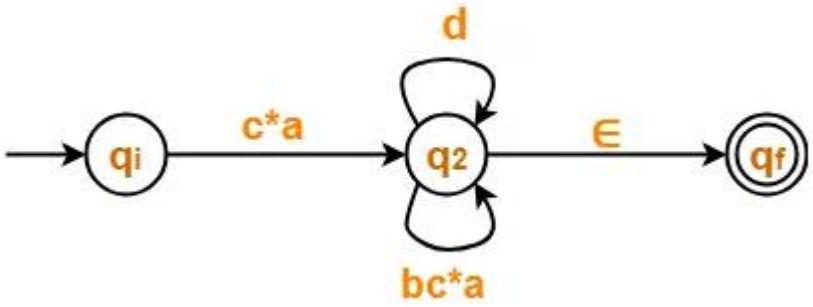


Step-03:

Now, we start eliminating the intermediate states.

First, let us eliminate state q_1.
There is a path going from state q_i to state q_2 via state q_1 .
So, after eliminating state q_1, we put a direct path from state q_i to state q_2 having cost $\in$.c*.a = c*a
There is a loop on state q_2 using state q_1 .
So, after eliminating state q_1 , we put a direct loop on state q_2 having cost b.c*.a = bc*a

Eliminating state q_1, we get-



Step-04:

Now, let us eliminate state q_2.
There is a path going from state q_i to state q_f via state q_2 .
So, after eliminating state q_2, we put a direct path from state q_i to state q_f having cost $c^*a(d+bc^*a)^*\in = c^*a(d+bc^*a)^*$

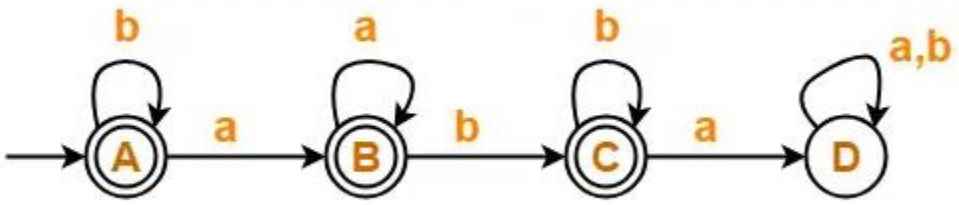
Eliminating state q_2, we get-

From here,

Regular Expression = c*a(d+bc*a)*

Problem-04:

Find regular expression for the following DFA-

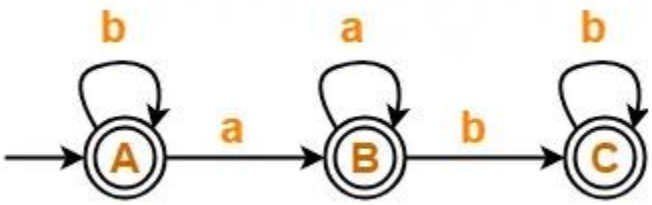


Solution-

Step-01:

State D is a dead state as it does not reach to any final state.
So, we eliminate state D and its associated edges.

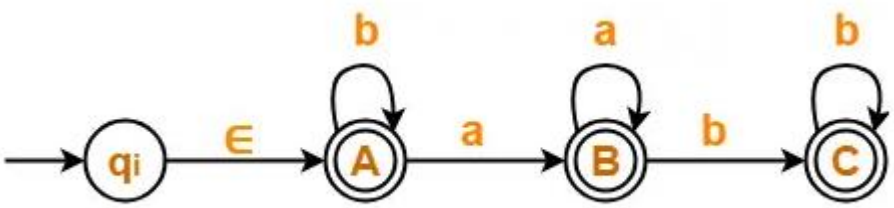
The resulting DFA is-



Step-02:

Initial state A has an incoming edge (self loop).
So, we create a new initial state q_i.

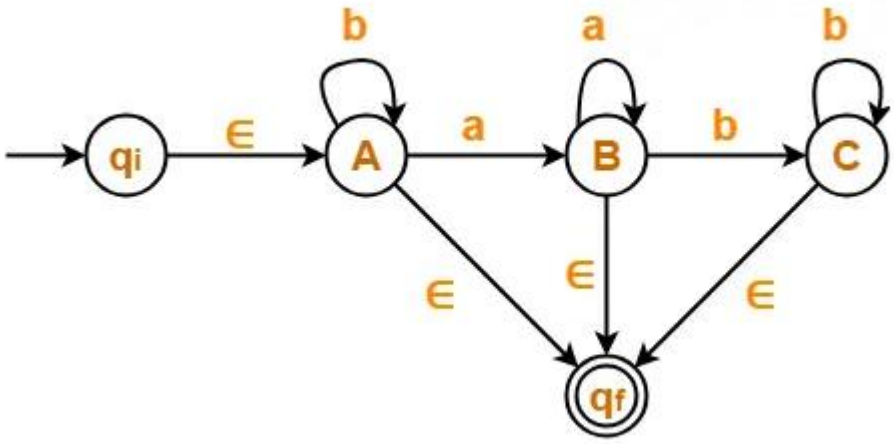
The resulting DFA is-



Step-03:

There exist multiple final states.
So, we convert them into a single final state.

The resulting DFA is-

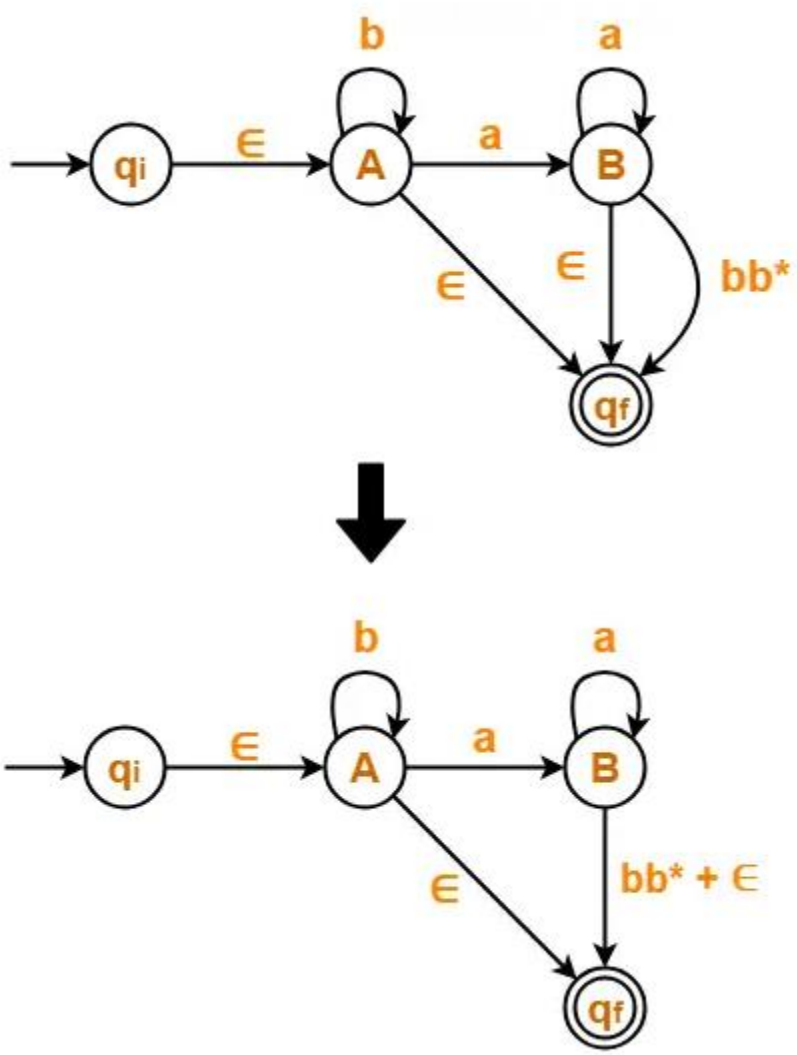


Step-04:

Now, we start eliminating the intermediate states.

First, let us eliminate state C.
There is a path going from state B to state q_f via state C.
So, after eliminating state C, we put a direct path from state B to state q_f having cost b.b*.∈ = bb*

Eliminating state C, we get-



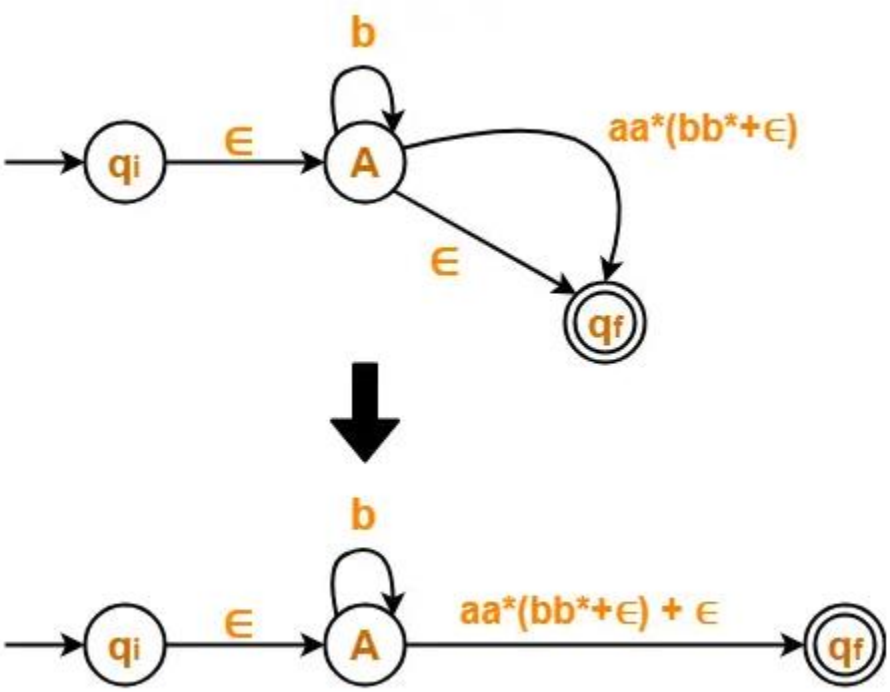
Step-05:

Now, let us eliminate state B.

There is a path going from state A to state q_f via state B.

So, after eliminating state B, we put a direct path from state A to state q_f having cost a.a*.(bb*+∈) = aa*(bb*+∈)

Eliminating state B, we get-



Step-06:

Now, let us eliminate state A.

There is a path going from state q_i to state q_f via state A.

So, after eliminating state A, we put a direct path from state q_i to state q_f having cost ∈.b*.(aa*(bb*+∈)+∈) = b*(aa*(bb*+∈)+∈)

Eliminating state A, we get-

From here,

Regular Expression = b*(aa*(bb*+∈)+∈)

We know, bb* + ∈ = b*

So, we can also write-

Regular Expression = b*(aa*b*+∈)

Context free grammar

Regular Languages can be expressed in

1. Regular Expressions
2. Finite Automata
3. Construction of Natural Language
4. Grammar

- Grammar is a set of rules(production rules) used to define the language. These rules can be rewritten to generate the desired string.
- Grammar is the effective way of representing regular languages.

Ex:- Language representing $1^+ = \{1,11,111\ldots\ldots\}$

to generate this given string we can write the production rule as

S-> 1

S->1S production rules

Non-terminal (represented in capitals)---- ex:- S

Terminals (represented in small)—ex:-1

Model	Language Recognition	Grammar used	Memory management	Computing Power
Pushdown Automata	Context-free Languages	Context-free grammar	stack	Used for medium computing Ex:- compilers for programming languages, Online Transaction Processing system, Tower Of Hanoi (Recursive Solution)

CFG is a formal grammar which is used to generate all possible patterns of strings in a given language.

Context **F**ree **G**rammars (CFG) can be classified on the basis of following two properties:

1) Based on number of strings it generates.

A. If CFG is generating finite number of strings, then CFG is **Non-Recursive**

Ex:- S->Aa

A->b|c

The language generated by the above grammar is :{ba, ca}, which is finite.

A. If CFG can generate infinite number of strings then the grammar is said **Recursive** grammar

Ex:- S->SaS

S->b

The language generated by the above grammar is :{b, bab, babab,…}, which is infinite

2) Based on number of derivation trees.

A. If there is only 1 derivation tree then the CFG is **unambiguous**.

B. If there are more than 1 left most derivation tree or right most derivation or parse tree , then the CFG is **ambiguous**

NOTE:-During Compilation, the parser uses the grammar of the language to make a parse tree(or derivation tree) out of the source code. The grammar used must be unambiguous. An ambiguous grammar must not be used for parsing.

CONTEXT FREE GRAMMARS

A grammar $G = (V, T, P, S)$ is said to be a CFG if the productions of G are of the form :

$$A \rightarrow \alpha, \text{ where } \alpha \in (V \cup T)^*$$

The right hand side of a CFG is not restricted and it may be null or a combination of variables and terminals. The possible length of right hand sentential form ranges from 0 to ∞ i.e., $0 \leq |\alpha| \leq \infty$.

As we know that a CFG has no context neither left nor right. This is why, it is known as CONTEXT - FREE. *Many programming languages have recursive structure that can be defined by CFG's.*

Example 1 : Consider the grammar $G = (V, T, P, S)$ having productions :
$S \rightarrow aSa \mid bSb \mid \in$. Check the productions and find the language generated.

Solution :

Let $P_1 : S \rightarrow aSa$ (RHS is terminal variable terminal)
$P_2 : S \rightarrow bSb$ (RHS is terminal variable terminal)
$P_3 : S \rightarrow \in$ (RHS is null string)

Since, all productions are of the form $A \rightarrow \alpha$, where $\alpha \in (V \cup T)^*$, hence G is a CFG.

Language Generated :

$S \Rightarrow aSa \text{ or } bSb$

$\Rightarrow a^nSa^n \text{ or } b^nSb^n$ (Using n step derivation)

$\Rightarrow a^nb^mSb^ma^n \text{ or } b^na^mSa^mb^n$ (Using m step derivation)

$\Rightarrow a^nb^mb^ma^n \text{ or } b^na^ma^mb^n$ (Using $S \rightarrow \in$)

So, $L(G) = \{ww^R : w \in (a + b)^*\}$

Example 2 : Let G = (V, T, P, S) where V = { S , C }, T = { a , b }
P = { S $\rightarrow$ aCa
C $\rightarrow$ aCa | b
} S is the start symbol
What is the language generated by this grammar ?

Solution : Consider the derivation
$S \Rightarrow aCa \Rightarrow$ aba (By applying the 1st and 3rd production)
So, the string aba $\in L(G)$
Consider the derivation

$S \Rightarrow aCa$	By applying	$S \rightarrow aCa$
$\Rightarrow aaCaa$	By applying	$C \rightarrow aCa$
$\Rightarrow aaaCaaa$	By applying	$C \rightarrow aCa$
.....		
.....		
$\Rightarrow a^nCa^n$	By applying	$C \rightarrow aCa$ n times
$\Rightarrow a^nba^n$	By applying	$C \rightarrow b$

So, the language L accepted by the grammar G is $L(G) = \{ a^n ba^n \mid n \geq 1 \}$
i. e., the language L derived from the grammar G is "The string consisting of n number of a's followed by a 'b' followed by n number of a's.

Example 3 : What is the language generated by the grammar
$S \rightarrow 0A \mid \in$
$A \rightarrow 1S$

Solution : The null string $\in$ can be obtained by applying the production $S \rightarrow \in$ and the derivation is shown below :

$S \Rightarrow \in$ (By applying $S \rightarrow \in$)

Consider the derivation

$S \Rightarrow 0A$ (By applying $S \rightarrow 0A$)

$\Rightarrow 01S$ (By applying $A \rightarrow 1S$)

$\Rightarrow 010A$ (By applying $S \rightarrow 0A$)

$\Rightarrow 0101S$ (By applying $A \rightarrow 1S$)

$\Rightarrow 0101$ (By applying $S \rightarrow \in$)

So, alternatively applying the productions $S \rightarrow 0A$ and $A \rightarrow 1S$ and finally applying the production $S \rightarrow \in$, we get string consisting of only of 01's. So, both null string i.e., $\in$ and string consisting 01's can be generated from this grammar. So, the language generated by this grammar is

$$L = \{w | w \in \{01\}^*\} \text{ or } L = \{(01)^n | n \geq 0\}$$

Example 5 : Draw a CFG to generate a language consisting of equal number of a's and b's.

Solution : Note that initial production can be of the form

$$S \rightarrow aB \mid bA$$

If the first symbol is **'a'**, the second symbol should be a non - terminal from which we can obtain either **'b'** or one more **'a'** followed by two **B's** denoted by **aBB** or a **'b'** followed by S denoted by **bS**.

Note that from all these symbols definitely we obtain equal number of a's and b's. The productions corresponding to these can be of the form

$$B \rightarrow b|aBB|bS$$

On similar lines we can write A - productions as

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

from which we obtain a 'b' followed by either

1. 'a' or
2. a 'b' followed by AA's denoted by bAA or
3. symbol 'a' followed by S denoted by aS

The context free grammar G = (V, T, P, S) where

$$V = \{S, A, B\}, \; T = \{a, b\}$$

$$P = \{ \; S \rightarrow aB \mid bA$$

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

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

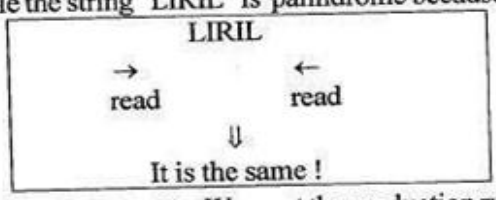
$\}$ S is the start symbol

generates the language consisting of equal number of a's and b's.

Example 6 : Construct CFG for the language L which has all the strings which are all palindromes over $T = \{a, b\}$

Solution : As we know the strings are palindrome if they posses same alphabets from forward as well as from backward.

For example the string "LIRIL" is palindrome because



Since the language L is over $T = \{a, b\}$. We want the production rules to be build a's and b's. As $\in$ can be the palindrome, a can be palindrome even b can be palindrome. So we can write the production rules as

$$G = (\{S\}, \{a, b\}, P, S)$$

P can be

$$S \rightarrow a\ S\ a$$
$$S \rightarrow b\ S\ b$$
$$S \rightarrow a$$
$$S \rightarrow b$$
$$S \rightarrow \in$$

The string **abaaba** can be derived as

$$S \rightarrow aSa$$
$$\rightarrow ab\ Sba$$
$$\rightarrow ab\ a\ Sa\ ba$$
$$\rightarrow ab\ a \in a\ ba$$
$$\rightarrow ab\ a\ a\ ba$$

which is a palindrome.

Example 7 : Obtain a CFG to generate integers .

Solution :

The sign of a number can be '+' or '-' or $\in$. The production for this can be written as

$$S \rightarrow +|-|\in$$

A number can be formed from any of the digits 0, 1, 2, ...9. The production to obtain these digits can be written as $D \rightarrow 0|1|2|...|9$

A number N can be recursively defined as follows .

1. A number N is a digit D (i. e., $N \rightarrow D$)
2. The number N followed by digit D is also a number (i. e., $N \rightarrow ND$)

The productions for this recursive definition can be written as

$$N \rightarrow D$$
$$N \rightarrow ND$$

An integer number I can be a number N or the sign S of a number followed by number N. The production for this can be written as $I \rightarrow N|SN$

So, the grammar G to obtain integer numbers can be written as $G = (V, T, P, S)$ where

V = { D, S, N, I } , T = { +, -, 0, 1, 2, ... 9 }

p = {

I → N | SN

N → D | ND

S → + | - | ∈

D → 0 | 1 | 2 | ..... | 9

}

S = I which is the start symbol

Example 8 : Obtain the grammar to generate the language

$L = \{ 0^m 1^m 2^n \mid m \geq 1 \text{ and } n \geq 0 \}$.

Solution : In the language $L = \{ 0^m 1^m 2^n \}$, if n = 0, the language L contains m number of 0's and m number of 1's. The grammar for this can be of the form

$$A \rightarrow 01 \mid 0A1$$

If n is greater than zero, the language L should contain m number of 0's followed by m number of 1's followed by one or more 2's i. e., the language generated from the non - terminal A should be followed by n number of 2's. So, the resulting productions can be written as

$$S \rightarrow A \mid S2$$

$$A \rightarrow 01 \mid 0A1$$

Thus, the grammar G to generate the language

$L = \{ 0^m 1^m 2^n \mid m \geq 1 \text{ and } n \geq 0 \}$

can be written as G = (V, T, P, S) where

V = { S, A } , T = { 0, 1, 2 }

P = {

$S \rightarrow A \mid S2$

$A \rightarrow 01 \mid 0A1$

} S is the start symbol

Example 9 : Obtain a grammar to generate the language $L = \{0^n 1^{n+1} \mid n \geq 0 \}$.

Solution :

Note : It is clear from the language that total number of 1's will be one more than the total number of 0's and all 0's precede all 1's. So, first let us generate the string $0^n 1^n$ and add the digit 1 at the end of this string.

The recursive definition to generate the string $0^n 1^n$ can be written as

$$A \rightarrow 0A1 \mid \in$$

If the production $A \rightarrow 0A1$ is applied n times we get the sentential form as shown below.

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow \quad \ldots\ldots 0^n A 1^n$$

Finally if we apply the production

$$A \rightarrow \in$$

the derivation starting from the start symbol A will be of the form

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 0^n A 1^n \Rightarrow 0^n 1^n$$

Thus, using these productions we get the string $0^n 1^n$. But, we should get the string $0^n 1^{n+1}$ i. e., an extra 1 should be placed at the end. This can be achieved by using the production

$$S \rightarrow A1$$

Note that from A we get string $0^n 1^n$ and 1 is appended at the end resulting in the string $0^n 1^{n+1}$.

So, the final grammar G to generate the language $L = \{ 0^n 1^{n+1} \mid n \geq 0 \}$ will be G = (V, T, P, S) where

V = { S, A } , T = { 0, 1 }

P = {

S → A 1

A → 0 A 1 | ∈

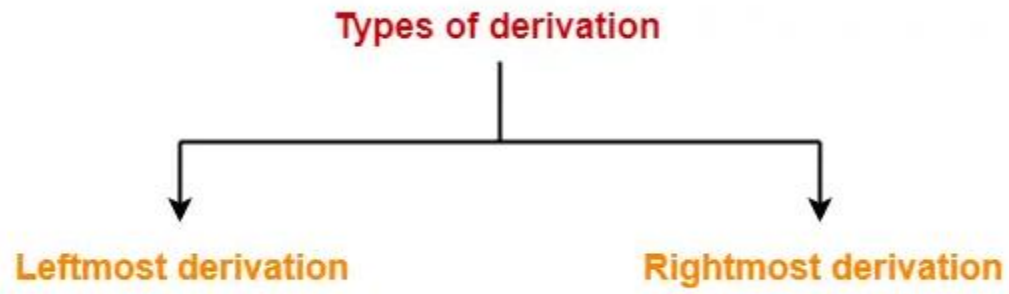
} S is the start symbol

DERIVATION TREE or PARSE TREE

The process of deriving a string is called as **derivation**.

Derivation tree or parse tree is a graphical representation for the derivation of the given production rules for a CFG.
IF A->α1α2… αn is given production in G
A parse tree contains the following properties:

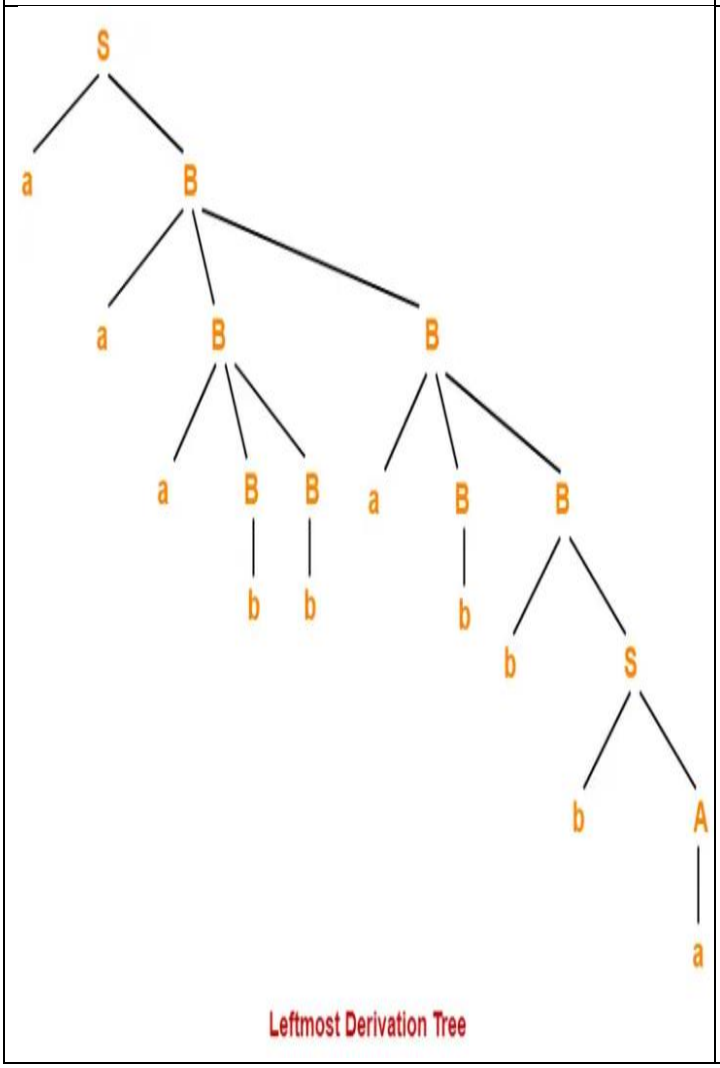
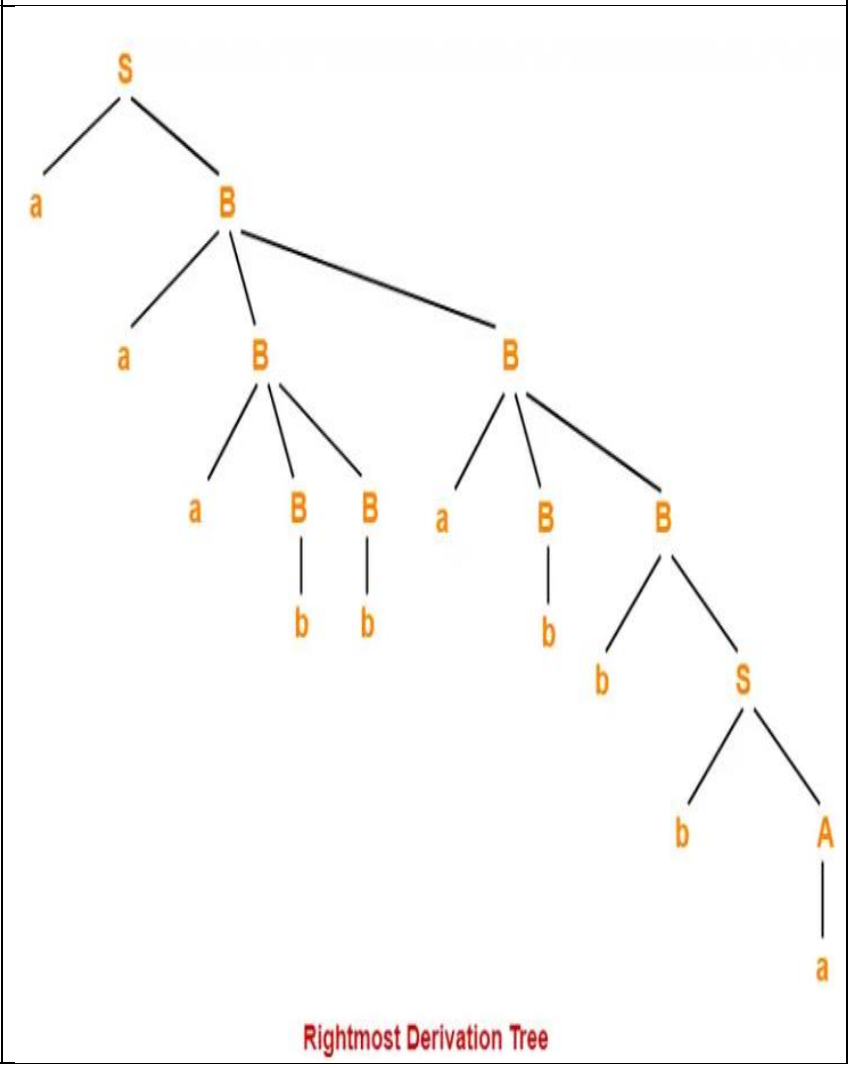
- A becomes the root node indicating start symbols.
- The derivation is read from left to right.
- The leaf node can be ε or terminal symbols.
- The interior nodes are always the non-terminal nodes/ variables.
- The collection of leaves from left to right yields the input string w.



The process of deriving a string by expanding the leftmost non-terminal at each step is called as **leftmost derivation**.
The process of deriving a string by expanding the rightmost non-terminal at each step is called as **rightmost derivation**.

EXAMPLE:-
Consider the following grammar-
S → aB / bA
S → aS / bAA / a
B → bS / aBB / b
Let us consider a string w = aaabbabbba
Now, let us derive the string w

Leftmost Derivation-		**Rightmost Derivation-**	
S → a**B**			
→ aa**B**B	(Using B → aBB)	S → a**B**	
→ aaa**B**BB	(Using B → aBB)	→ aaB**B**	(Using B → aBB)
→ aaab**B**B	(Using B → b)	→ aaBaB**B**	(Using B → aBB)
→ aaabb**B**	(Using B → b)	→ aaBaBb**S**	(Using B → bS)
→ aaabba**B**B	(Using B → aBB)	→ aaBaBbb**A**	(Using S → bA)
→ aaabbab**B**	(Using B → b)	→ aaBa**B**bba	(Using A → a)
→ aaabbabb**S**	(Using B → bS)	→ aa**B**abbba	(Using B → b)

→ aaabbabbb**A** (Using S → bA) → aaabbabbba (Using A → a)	→ aaaB**B**abbba (Using B → aBB) → aaa**B**babbba (Using B → b) → aaabbabbba (Using B → b)
 Leftmost Derivation Tree	 Rightmost Derivation Tree

Derivation tree or parse tree is a graphical representation for the derivation of the given production rules for a CFG.
IF A->α1α2… αn is given production in G
A parse tree contains the following properties:

- A becomes the root node indicating start symbols.
- The derivation is read from left to right.
- The leaf node can be ε or terminal symbols.
- The interior nodes are always the non-terminal nodes/ variables.
- The collection of leaves from left to right yields the input string w.

Example : Consider a CFG $S \rightarrow S + S \mid S * S \mid a \mid b$ and construct the derivation trees for all productions.

Solution :

For the production $S \rightarrow S + S$

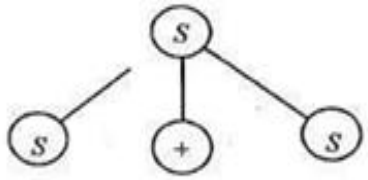


Figure (a)

For the production $S \rightarrow S * S$

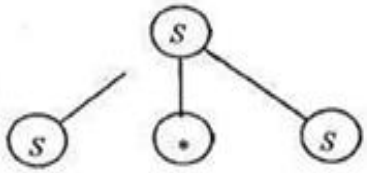


Figure (b)

For the production $S \rightarrow a$

Figure (c)

For the production $S \rightarrow b$

Figure (d)

DERIVATION TREES

Let $G = (V, T, P, S)$ is a CFG. Each production of G is represented with a tree satisfying the following conditions:

1. If $A \rightarrow \alpha_1\alpha_2\alpha_3 \ldots \alpha_n$ is a production in G, then A becomes the parent of nodes labeled $\alpha_1, \alpha_2, \alpha_3, \ldots \alpha_n$, and
2. The collection of children from left to right yields $\alpha_1\alpha_2\alpha_3 \ldots \alpha_n$

Example : Consider a CFG $S \rightarrow S + S \mid S * S \mid a \mid b$ and construct the derivation trees for all productions.

Solution :

For the production $S \rightarrow S + S$

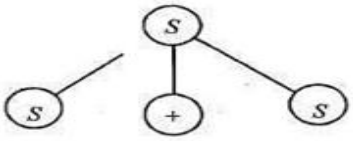


Figure (a)

For the production $S \rightarrow S * S$

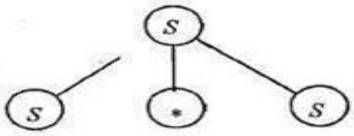


Figure (b)

For the production $S \rightarrow a$

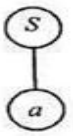


Figure (c)

For the production $S \rightarrow b$

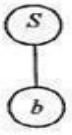


Figure (d)

If $w \in L(G)$ then it is represented by a tree called **derivation tree or parse tree** satisfying the following conditions :

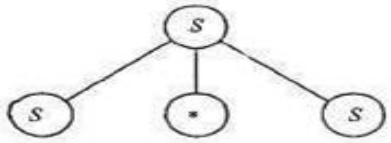
1. The root has label S (the starting symbol),
2. The all internal vertices (or nodes) are labeled with variables,
3. The leaves or terminal nodes are labeled with $\in$ or terminal symbols,
4. If $A \rightarrow \alpha_1\alpha_2\alpha_3 \ldots \alpha_n$ is a production in G, then A becomes the parent of nodes labeled $\alpha_1, \alpha_2, \alpha_3, \ldots \alpha_n$, and
5. The collection of leaves from left to right yields the string w.

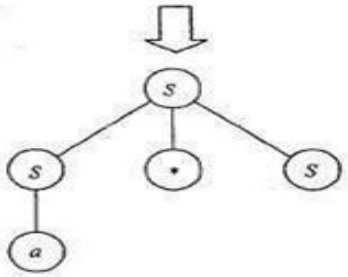
Example 1 : Consider the grammar $S \rightarrow S + S | S * S | a | b$. Construct derivation tree for string $w = a * b + a$.

Solution : The derivation tree or parse tree is shown in below figure.

Leftmost derivation for $w = a * b + a$:

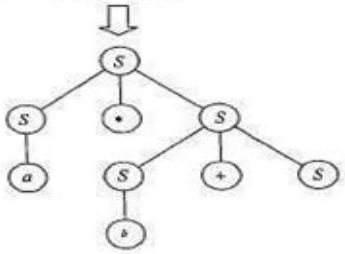
$S \Rightarrow S * S$ (Using $S \rightarrow S * S$)





$\Rightarrow a * S$ (The first left hand symbol is a, so using $S \rightarrow a$)

$\Rightarrow a * b + S$ (Second symbol from the left is b, so using $S \rightarrow b$)



$\Rightarrow a * b + a$ (The last symbol from the left is a, so using $S \rightarrow a$)

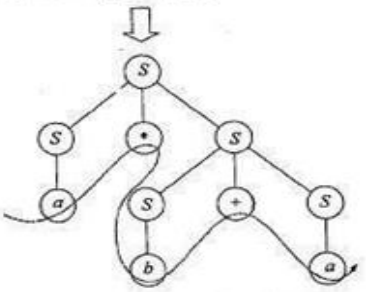


Figure : Parse tree for $a * b + a$

Example 2 : Consider a grammar G having productions $S \rightarrow aAS|a$, $A \rightarrow SbA|SS|ba$.

Show that $S \overset{*}{\Rightarrow} aabbaa$ and construct a derivation tree whose yield is aabbaa.

Solution :

$S \Rightarrow aAS$
$\Rightarrow aSbAS$
$\Rightarrow$ aabAS
$\Rightarrow aabbaS$
$\Rightarrow aabbaa$

Hence, $S \overset{*}{\Rightarrow} aabbaa$
Parse tree is shown in figure .

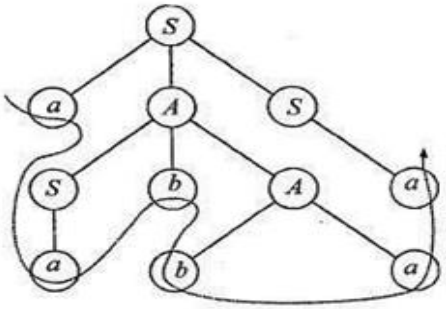


Figure : Parse tree yielding $aabbaa$

Example 3 : Consider the grammar G whose productions are

$S \rightarrow 0B|1A$, $A \rightarrow 0|0S|1AA$, $B \rightarrow 1|1S|0BB$. Find construct derivation tree.

Derivation tree :

Derivation tree is shown in below figure .

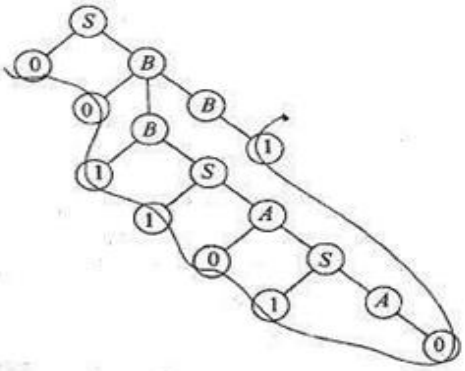


Figure : Derivation tree for 00110101

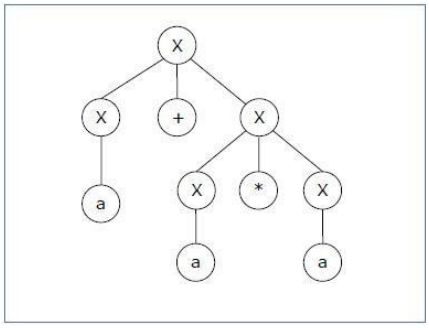
Ambiguity

A grammar is said to ambiguous if for any string generated by it, it produces more than one-
Parse tree
Or derivation tree
Or syntax tree
Or leftmost derivation
Or rightmost derivation

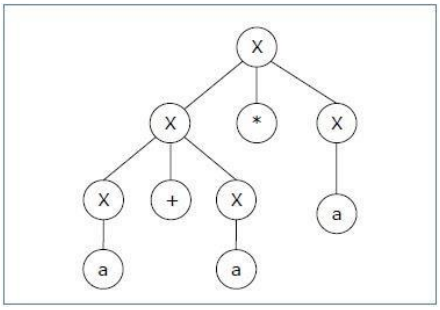
NOTE:- If the grammar has ambiguity, then it isn't useful for compiler construction.

Example:-Check whether the grammar G with production rules −
X → X+X | X*X |X| a is ambiguous or not.
Solution

Let's find out the derivation tree for the string "a+a*a". It has two leftmost derivations.
Derivation 1 − X → X+X → a +X → a+ X*X → a+a*X → a+a*a
Parse tree 1 −



Derivation 2 − X → X*X → X+X*X → a+ X*X → a+a*X → a+a*a
Parse tree 2 −

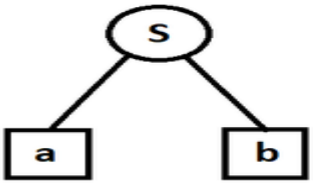


Since there are two parse trees for a single string "a+a*a", the grammar **G** is ambiguous.

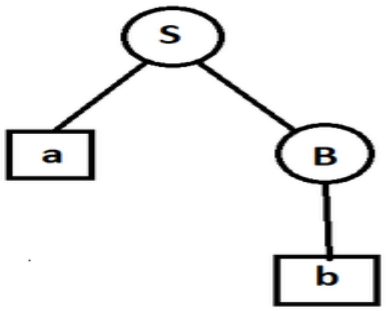
Example -

S -> aB | ab
A -> AB | a
B -> Abb | b

In the above example, although both left and right recursion are not present, but if we see string { ab }, we can make more than one parse tree to generate the string.



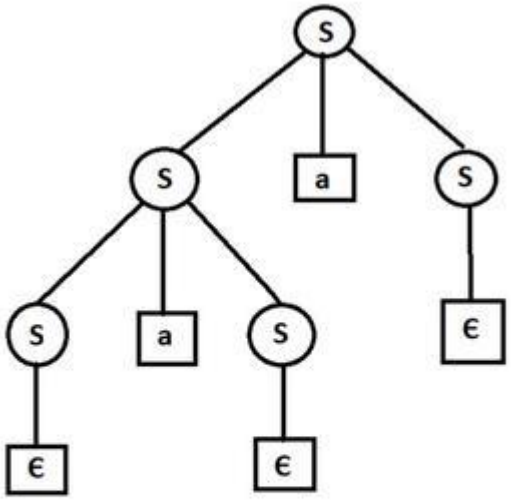
Parse Tree 1



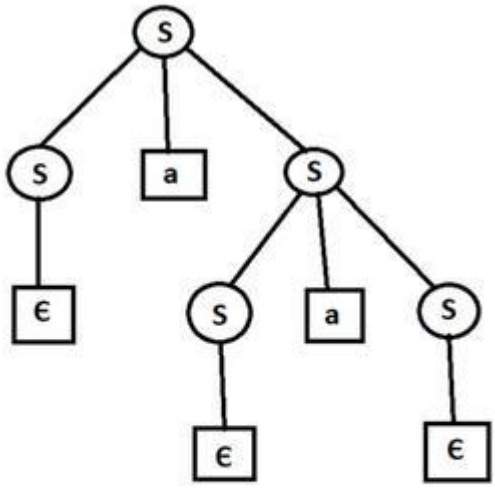
Parse Tree 2

Example - S -> SaS|Є

In this grammar we can see both left and right recurtion. So the grammar is ambiguous.
We can make more than one parse tree/derivation tree for input string (let's say {aa})



Parse Tree 1



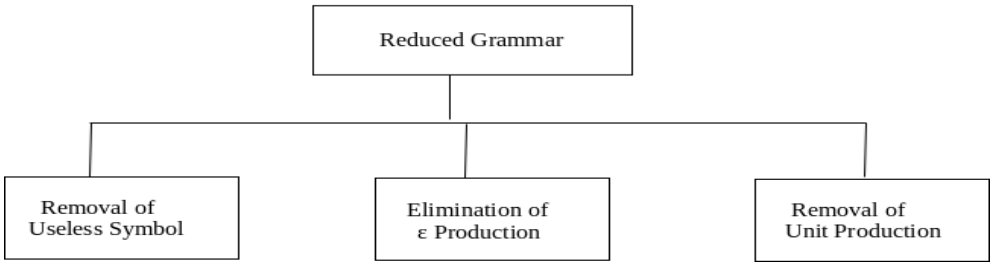
Parse Tree 2

Simplified forms

All the grammar are not always optimized that means the grammar may consist of some extra symbols(non-terminal). Having extra symbols, unnecessary increase the length of grammar. Simplification of grammar means reduction of grammar by removing useless symbols.

The properties of reduced grammar are given below:

1. Each variable (i.e. non-terminal) and each terminal of G appears in the derivation of some word in L.
2. There should not be any production as X → Y where X and Y are non-terminal.
3. If ε is not in the language L then there need not to be the production X → ε.



Removal of Useless Symbols

A symbol can be useless if it does not appear on the right-hand side of the production rule and does not take part in the derivation of any string. That symbol is known as a useless symbol. Similarly, a variable can be useless if it does not take part in the derivation of any string. That variable is known as a useless variable.

For Example:

1. T → aaB | abA | aaT
2. A → aA
3. B → ab | b
4. C → ad

In the above example, the variable 'C' will never occur in the derivation of any string, so the production C → ad is useless. So we will eliminate it, and the other productions are written in such a way that variable C can never reach from the starting variable 'T'.

Production A → aA is also useless because there is no way to terminate it. If it never terminates, then it can never produce a string. Hence this production can never take part in any derivation.

To remove this useless production A → aA, we will first find all the variables which will never lead to a terminal string such as variable 'A'. Then we will remove all the productions in which the variable 'B' occurs.

Example:-

S -> abS | abA | abB
A -> cd
B -> aB
C -> dc

In the example above , production ‘C -> dc’ is useless because the variable ‘C’ will never occur in derivation of any string. The other productions are written in such a way that variable ‘C’ can never reached from the starting variable ‘S’.

Production ‘B ->aB’ is also useless because there is no way it will ever terminate . If it never terminates , then it can never produce a string. Hence the production can never take part in any derivation.

To remove useless productions , we first find all the variables which will never lead to a terminal string such as variable ‘B’. We then remove all the productions in which variable ‘B’ occurs.
So the modified grammar becomes –

S -> abS | abA
A -> cd
C -> dc

Elimination of ε Production

The productions of type S → ε are called ε productions. These type of productions can only be removed from those grammars that do not generate ε.

Step 1: First find out all nullable non-terminal variable which derives ε.

Step 2: For each production A → ε, replace each occurrence of A with ε

Step 3: Now combine the result of step 2 with the original production and remove ε productions.
Example:
Remove the production from the following CFG by preserving the meaning of it.

1. S → XYX
2. X → 0X | ε
3. Y → 1Y | ε

Solution:

Now, while removing ε production, we are deleting the rule X → ε and Y → ε. To preserve the meaning of CFG we are actually placing ε at the right-hand side whenever X and Y have appeared.

Let us take
S → XYX
If the first X at right-hand side is ε. Then
S → YX
Similarly if the last X in R.H.S. = ε. Then
S → XY

If both X are replaced by ε
S → Y

If Y = ε then on XYX
S → XX

If Y = ε on XY or YX
S → X

Therefore ,
S → XY | YX | XX | X | Y

Now let us consider

X → 0X

If we place ε at right-hand side for X then,
X → 0
X → 0X | 0

Similarly Y → 1Y | 1

Collectively we can rewrite the CFG with removed ε production as
S → XY | YX | XX | X | Y
X → 0X | 0
Y → 1Y | 1

Example:-

S -> ABCd (1)
A -> BC (2)
B -> bB | ε (3)
C -> cC | ε (4)
Lets first find all the nullable variables. Variables 'B' and 'C' are clearly nullable because they contain 'λ' on the RHS of their production. Variable 'A' is also nullable because in (2) , both variables on the RHS are also nullable. So variables 'A' , 'B' and 'C' are nullable variables.
Lets create the new grammar. We start with the first production. Add the first production as it is. Then we create all the possible combinations that can be formed by replacing the nullable variables with ε. Therefore line (1) now becomes 'S -> ABCd | ABd | ACd | BCd | Ad | Bd |Cd | d '.We apply the same rule to line (2) but we do not add 'A -> ε' even though it is a possible combination. We remove all the productions of type 'V -> ε'. The new grammar now becomes –

S -> ABCd | ABd | ACd | BCd | Ad | Bd |Cd | d
A -> BC | B | C
B -> bB | b
C -> cC | c

Removing Unit Productions

The unit productions are the productions in which one non-terminal gives another non-terminal. Use the following steps to remove unit production:

Step 1: To remove X → Y, add production X → a to the grammar rule whenever Y → a occurs in the grammar.

Step 2: Now delete X → Y from the grammar.

Step 3: Repeat step 1 and step 2 until all unit productions are removed.
For example:

S → 0A | 1B | C
A → 0S | 00
B → 1 | A
C → 01

Solution:

S → C is a unit production. But while removing S → C we have to consider what C gives. So, we can add a rule to S.

S → 0A | 1B | 01

Similarly, B → A is also a unit production so we can modify it as
B → 1 | 0S | 00
Thus finally we can write CFG without unit production as

S → 0A | 1B | 01
A → 0S | 00
B → 1 | 0S | 00
C → 01

Example 1:

Consider the CFG,

$S \rightarrow A|bb$

$A \rightarrow B|b$

$B \rightarrow S|a$

where S is the start symbol.

Eliminate unit productions from this grammar.

Here unit productions are,

$S \rightarrow A$

$A \rightarrow B$

$B \rightarrow S$

Here,

$S \rightarrow A$ generates $S \rightarrow b$

$S \rightarrow A \rightarrow B$ generates $S \rightarrow a$

$A \rightarrow B$ generates $A \rightarrow a$

$A \rightarrow B \rightarrow S$ generates $A \rightarrow bb$

$B \rightarrow S$ generates $B \rightarrow bb$

$B \rightarrow S \rightarrow A$ generates $B \rightarrow b$

The new CFG is,

$S \rightarrow b|a|bb$

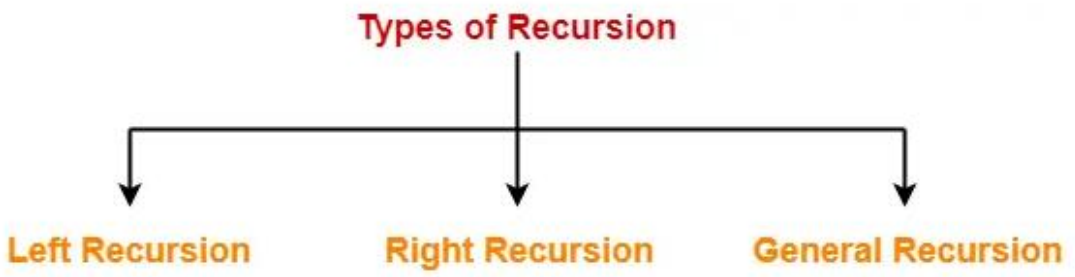
$A \rightarrow a|bb|b$

$B \rightarrow bb|b|a$

which contains no unit productions.

Recursion-

Recursion can be classified into following three types-

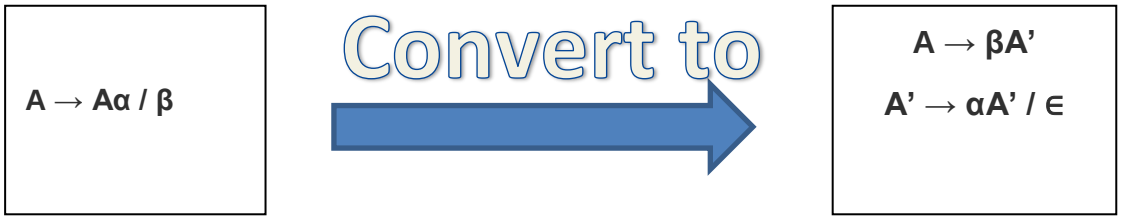


1. Left Recursion-

- A production of grammar is said to have **left recursion** if the leftmost variable of its RHS is same as variable of its LHS.
- A grammar containing a production having left recursion is called as Left Recursive Grammar.

  S → Sa / ∈ (**Left Recursive Grammar**)

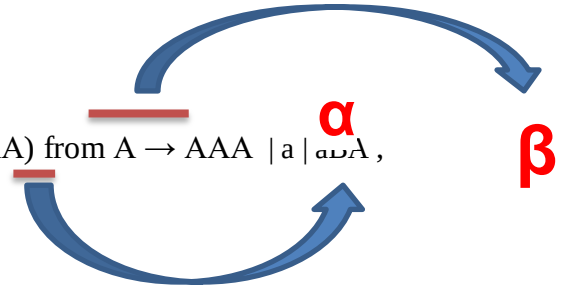
  Left recursion is considered to be a problematic situation for Top down parsers.
  Therefore, left recursion has to be eliminated from the grammar.



(**Left Recursive Grammar**) **(Right Recursive Grammar)**

S → aB | AA
A → a | aBA | AAA
B → b
X → a

To remove left recursion (A → AAA) from A → AAA | a | aBA ,



We get : S → aB | AA
A → aC | aBAC
C → AAC | ε
B → b
X → a

2. Right Recursion-

- A production of grammar is said to have **right recursion** if the rightmost variable of its RHS is same as variable of its LHS.
- A grammar containing a production having right recursion is called as Right Recursive Grammar.

$$S \rightarrow aS / \in$$

(**Right Recursive Grammar**)

Right recursion does not create any problem for the Top down parsers.
Therefore, there is no need of eliminating right recursion from the grammar.

3. General Recursion-

- The recursion which is neither left recursion nor right recursion is called as general recursion.

Example-

$$S \rightarrow aSb / \in$$

NORMAL FORMS

As we have seen the grammar can be simplified by reducing the ϵ production, removing useless symbols, unit productions. There is also a need to have grammar in some specific form. As you have seen in CFG at the right hand of the production there are any number of terminal or non - terminal symbols in any combination. We need to normalize such a grammar. That means we want the grammar in some specific format. That means there should be fixed number of terminals and non - terminals, in the context free grammar.

In a CFG, there is no restriction on the right hand side of a production. The restrictions are imposed on the right hand side of productions in a CFG resulting in normal forms. The different normal forms are :

1. Chomsky Normal Form (CNF)
2. Greiback Normal Form (GNF)

5.6.1 Chomsky Normal Form (CNF)

Chomsky normal form can be defined as follows.

Non - terminal $\rightarrow$ Non - terminal.Non - terminal
Non - terminal $\rightarrow$ terminal

The given CFG should be converted in the above format then we can say that the grammar is in CNF. Before converting the grammar into CNF it should be in reduced form. That means remove all the useless symbols, ϵ productions and unit productions from it. Thus this reduced grammar can be then converted to CNF.

Definition :
Let G = (V, T, P, S) be a CFG. The grammar G is said to be in CNF if all productions are of the form

$$A \rightarrow BC$$
or
$$A \rightarrow a$$

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

Note that if a grammar is in CNF, the right hand side of the production should contain two symbols or one symbol. If there are two symbols on the right hand side those two symbols must be non - terminals and if there is only one symbol, that symbol must be a terminal.

Theorem 5.6.1 : Let G = (V, T, P, S) be a CFG which generates context free language without ϵ. We can find an equivalent context free grammar $G_1 = (V_1, T, P_1, S)$ in CNF such that $L(G)=L(G_1)$ i. e., all productions in G_1 are of the form

$$A \rightarrow BC$$
or
$$A \rightarrow a$$

Proof : Let the grammar G has no $\in$ - productions and unit productions. The grammar G_1 can be obtained using the following steps.

Step 1 : Consider the productions of the form

$$A \to X_1 X_2 X_3 \ldots\ldots\ldots X_n$$

where $n \geq 2$ and each $X_i \in (V \cup T)$ i. e., consider the productions having more than two symbols on the right hand side of the production. If X is a terminal say a, then replace this terminal by a corresponding non terminal B_a and introduce the production

$$B_a \to a$$

The non - terminals on the right hand side of the production are retained. The resulting productions are added to P_1. The resulting context free grammar $G_1 = (V_1, T, P_1, S)$ where each production in P_1 is of the form

$$A \to A_1 A_2 \ldots\ldots\ldots A_n$$

or

$$A \to a$$

generates the same language as accepted by grammar G. So, $L(G) = L(G_1)$.

Step 2 : Restrict the number of variables on the right hand side of the production. Add all the productions of G_1 which are in CNF to P_1. Consider a production of the form

$$A \to A_1 A_2 \ldots\ldots\ldots A_n$$

where $n \geq 3$ (Note that if n = 2, the production is already in CNF and n can not be equal to 1. Because if n = 1, there is only one symbol and it is a terminal which again is in CNF). The A - production can be written as

$$\begin{aligned} A &\to A_1 D_1 \\ D_1 &\to A_2 D_2 \\ D_2 &\to A_3 D_3 \\ &\vdots \\ D_{n-2} &\to A_{n-1} D_{n-1} \end{aligned}$$

These productions are added to P_1 and new variables are added to V_1. The grammar thus obtained is in CNF. The resulting grammar $G_1 = (V_1, T, P_1, S)$ generates the same language as accepted by G i. e. $L(G) = L(G_1)$.

Example 1 : Consider the grammar

S → 0A | 1 B
A → 0AA | 1 S | 1
B → 1 BB | 0 S | 0 Obtain the grammar in CNF :

Solution :

Step 1 : All productions which are in CNF are added to P_1. The productions which are in standard form and added to P_1 are :

A → 1
B → 0 ..... (1)

Consider the productions, which are not in CNF. Replace the terminal a on right hand side of the production by a non - terminal A and introduce the production A → a. This step has to be carried out for each production which are not in CNF.

The table below shows the action taken indicating which terminal is replaced by the corresponding non - terminal and what is the new production introduced. The last column shows the resulting productions.

Given Productions	Action	Resulting productions
S → 0A |1B	Replace 0 by B_0 and introduce the production $B_0 \to 0$ Replace 1 by B_1 and introduce the production $B_1 \to 1$	$S \to B_0 A \mid B_1 B$ $B_0 \to 0$ $B_1 \to 1$
A → 0AA/1S	Replace 0 by B_0 and introduce the production $B_0 \to 0$ Replace 1 by B_1 and introduce the production $B_1 \to 1$	$A \to B_0 AA / B_1 S$ $B_0 \to 0$ $B_1 \to 1$
B → 1BB/0S	Replace 0 by B_0 and introduce the production $B_0 \to 0$ Replace 1 by B_1 and introduce the production $B_1 \to 1$	$B \to B_1 BB / B_0 S$ $B_1 \to 1$ $B_0 \to 0$

The grammar $G_1 = (V_1, T, P_1, S)$ can be obtained by combining the productions obtained from the last column in the table and the productions shown in (1).

V_1 = { S, A, B, B_0, B_1 }

T_1 = { 0, 1 }

P_1 = {

S → $B_0A \mid B_1B$

A → $B_0AA \mid B_1S \mid 1$

B → $B_1BB \mid B_0S \mid 0$

B_0 → 0

B_1 → 1

} S is the start symbol

Step 2 :

Restricting the number of variables on the right hand side of the production to 2. The productions obtained after step 1 are :

$S \rightarrow B_0A \mid B_1B$

$A \rightarrow B_0AA \mid B_1S \mid 1$

$B \rightarrow B_1BB \mid B_0S \mid 0$

$B_0 \rightarrow 0$

$B_1 \rightarrow 1$

In the above productions, the productions which are in CNF are

$S \rightarrow B_0A \mid B_1B$

$A \rightarrow B_1S \mid 1$

$B \rightarrow B_0S \mid 0$

$B_0 \rightarrow 0$

$B_1 \rightarrow 1$ ..... (2)

and add these productions to P_1. The productions which are not in CNF are

$A \rightarrow B_0AA$

$B \rightarrow B_1BB$

The following table shows how these productions are changed to CNF so that only two variables are present on the right hand side of the production.

Given Productions	Action	Resulting productions
$A \to B_0AA$	Replace AA on R.H.S with variable D_1 and introduce the production $D_1 \to AA$	$A \to B_0D_1$ $D_1 \to AA$
$B \to B_1BB$	Replace BB on R. H. S with variable D_2 and introduce the production $D_2 \to BB$	$B \to B_1D_2$ $D_2 \to BB$ .....(3)

The final grammar which is in CNF can be obtained by combining the productions in (2) and (3).

The grammar $G_1 = (V_1, T, P_1, S)$ is in CNF where

V_1 = { S, A, B, B_0, B_1, D_1, D_2 }

T_1 = { 0, 1 }

P_1 = {

$S \to B_0A \mid B_1B$

$A \to B_1S \mid 1 \mid B_0D_1$

$B \to B_0S \mid 0 \mid B_1D_2$

$B_0 \to 0$

$B_1 \to 1$

$D_1 \to AA$

$D_2 \to BB$

} S is the start symbol

Example 2 : Find a grammar in CNF equivalent to the grammar :

$$S \to -S \mid [\, S \uparrow S \,] \mid a \mid b$$

Solution : Given, grammar is :

$$S \to -S \mid [\, S \uparrow S \,] \mid a \mid b \qquad \text{.....(A)}$$

where, terminals are :

$$-, [\,, \uparrow,], \ a \text{ and } b$$

In the given grammar (A) there is no any $\in$–production, no any unit - production and no any useless symbols.

Now, in the given grammar (A), following are the productions which is already in the form of CNF:

$S \to a$

$S \to b$

Also, in the given grammar (A), following are the productions which are not in the form of CNF:

$$S \to -S$$

$$S \to [S \uparrow S]$$

Thus: (a) Considering the production:

$$S \to -S$$

We can write this production as:

$$S \to V_1 S \quad \text{.... (1)}$$

$$V_1 \to - \quad \text{.... (2)}$$

where V_1 is a new variable.

(b) Now, considering the production:

$$S \to [S \uparrow S]$$

We can write this production as:

$$S \to V_2\, SV_3\, SV_4 \quad \text{.... (3)}$$

$$V_2 \to [\quad \text{.... (4)}$$

$$V_3 \to \uparrow \quad \text{.... (5)}$$

$$V_4 \to] \quad \text{.... (6)}$$

where V_2, V_3 and V_4 are new variables.

Thus, from (1), ..., (6), the result grammar becomes:

$$S \to V_1 S \mid V_2 S\, V_3\, SV_4 \mid a \mid b$$

$$V_1 \to -$$

$$V_2 \to [\quad \text{.....(B)}$$

$$V_3 \to \uparrow$$

$$V_4 \to]$$

Now, in the resultant grammar (B), following is the production which is not in the form of CNF:

$$S \to V_2\, SV_3\, SV_4$$

(c) Now, considering the production:

$$S \to V_2 SV_3 SV_4$$

We can write this production as:

$$S \to V_2 V_5 V_6 \quad \text{.....(7)}$$

$$V_5 \to SV_3 \quad \text{.....(8)}$$

$$V_6 \to SV_4 \quad \text{.....(9)}$$

Thus, from (7), (8) and (9), the resultant grammar becomes :

$$S \rightarrow V_1\ S \mid V_2 V_5 V_6 \mid a \mid b$$
$$V_1 \rightarrow -$$
$$V_2 \rightarrow [$$
$$V_5 \rightarrow SV_3 \qquad \text{.....(C)}$$
$$V_6 \rightarrow SV_4$$
$$V_3 \rightarrow \uparrow$$
$$V_4 \rightarrow]$$

Now, in the resultant grammar (C), following is the production which is not in the form of CNF:

$$S \rightarrow V_2 V_5 V_6$$

We can write this production as :

$$S \rightarrow V_2 V_7 \qquad \text{.....(10)}$$
$$V_7 \rightarrow V_5 V_6 \qquad \text{.....(11)}$$

Thus, from (10) and (11), the resultant grammar becomes :

$$S \rightarrow V_1 S \mid V_2 V_7 \mid a \mid b$$
$$V_1 \rightarrow -$$
$$V_2 \rightarrow [$$
$$V_7 \rightarrow V_5 V_6 \qquad \text{.....(D)}$$
$$V_5 \rightarrow SV_3$$
$$V_6 \rightarrow SV_4$$
$$V_3 \rightarrow \uparrow$$
$$V_4 \rightarrow]$$

Thus, the resultant grammar (D) is in the form of CNF, which is the required solution.

Greibach Normal form (GNF)

Greibach normal form can be defined as follows :

Non - terminal → one terminal. Any number of non - terminals

Example :

$S \rightarrow aA$ is in GNF
$S \rightarrow a$ is in GNF

But $S \to AA$ is not in GNF

$S \to Aa$ is not in GNF

Definition : A CFG $G = (V, T, P, S)$ is in Greibach normal form (GNF) if its all productions are of type $A \to a\alpha$, where $\alpha \in V^*$ (String of variables including null string) and $a \in T$. A grammar in GNF is the natural generalization of a regular grammar (right - linear).

Theorem 5.6.2 : Every CFL L without $\in$ is generated by grammar, where productions are of type $A \to a\alpha$, where $\alpha \in V^*$ and $a \in T$.

Proof : We use removal of left recursion (without null productions) as given below.

Let the variable A has left recursive productions given as follows :

$A \to A\alpha_1 | A\alpha_2 | A\alpha_3 | .... | A\alpha_n | \beta_1 | \beta_2 | \beta_3 | .... | B_m$, where $\beta_1, \beta_2, \beta_3, ... B_m$ do not begin with A, then we replace A - productions by the productions given below .

$$A \to \beta_1 A' | \beta_2 A' | .... | \beta_m A' | \beta_1 | \beta_2 | \beta_3 | ... | B_m, \text{ where}$$

$$A' \to \alpha_1 A' | \alpha_2 A' | \alpha_3 A' | .... | \alpha_n A' | \alpha_1 | \alpha_2 | \alpha_3 | ... | \alpha_n$$

Method for Converting a CFG into GNF :

We consider CFG $G = (V, T, P, S)$.

Step 1 : Rename all the variables of G as $A_1, A_2, A_3, ..., A_n$

Step 2 : Repeat Step 3 and Step 4 for $i = 1, 2, ..., n$

Step 3 : If $A_i \to a\alpha_1\alpha_2\alpha_3 ..... \alpha_m$, where $a \in T$, and α_j is a variable or a terminal symbol,

Repeat for $j = 1, 2, ..., m$

If α_j is a terminal then replace it by a variable A_{n+j} and add production $A_{n+j} \to \alpha_j$, and $n = n+1$. Consider the next A_i – production and go to step 3.

Step 4 : If $A_i \to \alpha_1\alpha_2\alpha_3 ..... \alpha_m$, where α_1 is a variable, then perform the following :

If α_1 is same as A_i, then remove the left recursion and go to Step 3.

Else replace α_1 by all RHS of α_1 -productions one by one. Consider the remaining A_i -productions, which are not in GNF and go to Step 3.

Step 5 : Exit

Advantages of GNF :

1. Avoids left recursion.
2. Always has terminal in leftmost position in RHS of each production.
3. Helps select production correctly.
4. Guarantees derivation length no longer than string length.

Example 1 : Consider the CFG $S \to S+S \mid S*S \mid a \mid b$ and find an equivalent grammar in GNF.

Solution: Let G_1 is the equivalent grammar in GNF.

Renaming the variable, we get

$P_1 : S_1 \to S_1 + S_1$ (Not in GNF)

$P_2 : S_1 \to S_1 * S_1$ (Not in GNF)

$P_3 : S_1 \to a$ (In GNF)

$P_4 : S_1 \to b$ (In GNF)

P_1 and P_2 are left recursive productions, so removing the left recursion, we get

$S_1 \to a S_2 \mid b S_2 \mid a \mid b$, where

$S_2 \to + S_1 S_2 \mid * S_1 S_2 \mid + S_1 \mid * S_1$

Now, all productions are in GNF.

Example 2 : Consider the grammar $G = (\{A_1, A_2, A_3\}, \{a,b\}, P, A_1)$, where P consists of following production rules.

$A_1 \to A_2 A_3, A_2 \to A_3 A_1 \mid b, A_3 \to A_1 A_2 \mid a$ Convert it into GNF.

Solution : (Renaming is not required)

Consider A_1 - productions :

$A_1 \to A_2 A_3$ (Not in GNF)

Replacing A_2 by its RHS, we get

$A_1 \to bA_3$ (In GNF)

$A_1 \to A_3 A_1 A_3$ (Not in GNF)

Now , consider $A_1 \to A_3 A_1 A_3$, and replacing A_3 by its RHS, we get

$A_1 \to aA_1 A_3$ (In GNF)

$A_1 \to A_1 A_2 A_1 A_3$ (Not in GNF)

So, A_1 - productions are $A_1 \to bA_3 \mid aA_1 A_3 \mid A_1 A_2 A_1 A_3$

Now, consider $A_1 \to A_1 A_2 A_1 A_3$ and removing left recursion, we get

$A_1 \to bA_3 A_4 \mid bA_3$ (In GNF)

$A_1 \to aA_1 A_3 A_4 \mid aA_1 A_3$ (In GNF), where

$A_4 \to A_2 A_1 A_3 A_4 \mid A_2 A_1 A_3$

(A_4 is a new variable and its production is not in GNF)

So, now all A_1 - productions are in GNF .

Consider the A_2 - productions :

$A_2 \to b$ (In GNF)

$A_2 \to A_3 A_1$ (Not in GNF)

Now, consider $A_2 \to A_3 A_1$ and replacing A_3 by its RHS, we get

$A_2 \to aA_1$ (In GNF)

$A_2 \to A_1 A_2 A_1$ (Not in GNF)

Now, consider $A_2 \to A_1 A_2 A_1$ and replacing A_1 by its RHS, we get

$A_2 \to bA_3 A_4 A_2 A_1$, (In GNF)

$A_2 \to bA_3 A_2 A_1$ (In GNF)

$A_2 \to aA_1 A_3 A_4 A_2 A_1$ (In GNF)

$A_2 \to aA_1 A_3 A_2 A_1$ (In GNF)

So , all A_2 - productions are in GNF.

Consider A_3 - productions :

$A_3 \to a$ (In GNF)

$A_3 \to A_1 A_2$ (Not in GNF)

Now, consider $A_3 \to A_1 A_2$ and replacing A_1 by its RHS, we get

$A_3 \to bA_3 A_4 A_2 \mid bA_3 A_2 \mid aA_1 A_3 A_4 A_2 \mid aA_1 A_3 A_2$ (In GNF)

Consider A_4 - productions :

$A_4 \to A_2 A_1 A_3 A_4 \mid A_2 A_1 A_3$ (Not in GNF)

Replacing A_2 by its RHS, we get

$A_4 \to bA_1 A_3 A_4 \mid bA_1 A_3 \mid aA_1 A_1 A_3 A_4 \mid aA_1 A_1 A_3$,

$A_4 \to bA_3 A_4 A_2 A_1 \; A_1 A_3 A_4$,

$A_4 \to bA_3 A_4 A_2 A_1 \; A_1 A_3$,

$A_4 \to bA_3 A_2 A_1 \; A_1 A_3 A_4$,

$A_4 \to bA_3 A_2 A_1 \; A_1 A_3$,

$A_4 \to aA_1 A_3 A_4 A_2 A_1 A_1 A_3 A_4$,

$A_4 \to aA_1 A_3 A_4 A_2 A_1 A_1 A_3$,

$A_4 \to aA_1 A_3 A_2 A_1 A_1 A_3 A_4$, and

$A_4 \to aA_1 A_3 A_2 A_1 A_1 A_3$

Now, all A_4 - productions are in GNF.

Productions in GNF are :

$A_1 \to aA_1A_3 \mid bA_3A_4 \mid bA_3 \mid aA_1A_3A_4 \mid aA_1A_3$

$A_2 \to b \mid aA_1 \mid bA_3A_4A_2A_1 \mid bA_3A_2A_1 \mid aA_1A_3A_4A_2A_1 \mid aA_1A_3A_2A_1$,

$A_3 \to a \mid bA_3A_4A_2 \mid bA_3A_2 \mid aA_1A_3A_4A_2 \mid aA_1A_3A_2$,

$A_4 \to bA_1A_3A_4 \mid bA_1A_3 \mid aA_1A_1A_3A_4 \mid aA_1A_1A_3 \mid bA_3A_4A_2A_1A_1A_3A_4$,

$A_4 \to bA_3A_2A_1A_1A_3A_4 \mid aA_1A_3A_4A_2A_1A_1A_3A_4 \mid aA_1A_3A_2A_1A_1A_3A_4$,

$A_4 \to bA_3A_4A_2A_1A_1A_3 \mid bA_3A_2A_1A_1A_3 \mid aA_1A_3A_4A_2A_1A_1A_3 \mid aA_1A_3A_2A_1A_1A_3$

Example 3 : Find equivalent grammar in GNF.

(a) S → aB | b A, A → aS | b AA | a, B → bS | a BB | b
(b) S → abSb | a | a Ab, A → bS | a AAb
(c) S → AA | 0 , A → SS | 1

Solution :

(a) Renaming S, A and B by A_1, A_2, and A_3 respectively, we get the following productions.

$A_1 \to aA_3 \mid bA_2, A_2 \to aA_1 \mid bA_2A_2 \mid a, A_3 \to bA_1 \mid aA_3A_3 \mid b$

Since, all productions are in GNF, so there is no need of any modification.

(b) Renaming S, and A by A_1 and A_2 respectively, we get the following productions.

$A_1 \to abA_1b \mid a \mid aA_2b, A_2 \to bA_1 \mid aA_2A_2b$

Consider the A_1 - productions one by one.

$A_1 \to abA_1b$ (Not in GNF)

Replacing all the RHS terminals except the first by new variables, we get

$A_1 \to aA_3A_1A_3$ where $A_3 \to b$ (In GNF)

Considering the next A_1 - production :

$A_1 \to a$ (In GNF)

Considering the next A_1 - production :

$A_1 \to aA_2b$ (Not in GNF)

Replacing b by variable A_3 (since, we have already defined $A_3 \to b$), we get

$A_1 \to aA_2A_3$ (In GNF)

Consider the A_2 - production :

$A_2 \to bA_1$ (In GNF)

Considering the next A_2 - production :

$A_2 \to aA_2A_2b$ (Not in GNF)

Replacing b by variable A_3 (since, we have already defined $A_3 \to b$), we get

$A_2 \to aA_2A_2A_3$ (In GNF)

Now, all productions given following are in GNF.

$A_1 \to aA_3A_1A_3 \mid a \mid aA_2A_3$, $A_2 \to bA_1 \mid aA_2A_2A_3$, and $A_3 \to b$

(c) Renaming S, and A by A_1, and A_2 respectively, we get the following productions

$A_1 \to A_2A_2 \mid 0$, $A_2 \to A_1A_1 \mid 1$

Consider the A_1 - productions one by one.

$A_1 \to A_2A_2$ (Not in GNF)

Replacing leftmost A_2 by A_1A_1 and 1, we get

$A_1 \to A_1A_1A_2 \mid 1A_2$

Considering the production $A_1 \to A_1A_1A_2$, this is not in GNF and has left recursion. Considering the all A_1 - productions $A_1 \to A_1A_1A_2 \mid 1A_2 \mid 0$ and removing left recursion from the production $A_1 \to A_1A_1A_2$, we get $A_1 \to 1A_2A_3 \mid 0A_3$ (In GNF),

Where $A_3 \to A_1A_2A_3 \mid A_1A_2$

Considering A_2 - production $A_2 \to A_1A_1$ and replacing left most A_1 by $1A_2A_3$ and $0A_3$, we get

$A_2 \to 1A_2A_3A_1 \mid 0A_3A_1$ (In GNF)

Considering A_3 - productions $A_3 \to A_1A_2A_3 \mid A_1A_2$ and replacing A_1 by $1A_2A_3$ and $0A_3$, we get

$A_3 \to 1A_2A_3A_2 \mid 0A_3A_2 \mid 1A_2A_3A_2A_3 \mid 0A_3A_2A_3$ (In GNF)

Now, the productions in GNF are following.

$A_1 \to 1A_2A_3 \mid 0A_3$, $A_2 \to 1A_2A_3A_1 \mid 0A_3A_1 \mid 1$,

and $A_3 \to 1A_2A_3A_2 \mid 0A_3A_2 \mid 1A_2A_3A_2A_3 \mid 0A_3A_2A_3$

Chomsky's Normal Form (CNF):

CNF stands for Chomsky normal form. A CFG(context free grammar) is in

Definition 5: A CFG G = (V, T, P, S) is in Chomsky normal form if for each product of G is of the forms

$A \rightarrow BC$, or $A \rightarrow a$, where A, B and C are in V, and a is in T.

For example:
G1 = {S → AB, S → c, A → a, B → b}
G2 = {S → aA, A → a, B → c}

The production rules of Grammar G1 satisfy the rules specified for CNF, so the grammar G1 is in CNF. However, the production rule of Grammar G2 does not satisfy the rules specified for CNF as S → aZ contains terminal followed by non-terminal. So the grammar G2 is not in CNF.

Steps for converting CFG into CNF

Step 1: Eliminate start symbol from the RHS. If the start symbol T is at the right-hand side of any production, create a new production as:
S1 → S
Where S1 is the new start symbol.

Step 2: In the grammar, remove the null, unit and useless productions. You can refer to the Simplification of CFG.

Step 3: Eliminate terminals from the RHS of the production if they exist with other non-terminals or terminals. For example, production S → aA can be decomposed as:

S → RA
R → a
Step 4: Eliminate RHS with more than two non-terminals. For example, S → ASB can be decomposed as: S → RS
R → AS

Example: Convert the given CFG to CNF. Consider the given grammar G1:
S → a | aA | B
A → aBB | ε
B → Aa | b
Solution:
Step 1: We will create a new production S1 → S, as the start symbol S appears on the RHS. The grammar will be:
S1 → S
S → a | aA | B
A → aBB | ε
B → Aa | b
Step 2: As grammar G1 contains A → ε null production, its removal from the grammar yields: S1 → S
S → a | aA | B
A → aBB
B → Aa | b | a
Now, as grammar G1 contains Unit production S → B, its removal yield:
S1 → S
S → a | aA | Aa | b
A → aBB
B → Aa | b | a
Also remove the unit production S1 → S, its removal from the grammar yields:
S0 → a | aA | Aa | b
S → a | aA | Aa | b
A → aBB
B → Aa | b | a
Step 3: In the production rule S0 → aA | Aa, S → aA | Aa, A → aBB and B → Aa, terminal a exists on RHS with non-terminals. So we will replace terminal a with X:
S0 → a | XA | AX | b
S → a | XA | AX | b
A → XBB
B → AX | b | a
X → a
Step 4: In the production rule A → XBB, RHS has more than two symbols, removing it from grammar yield: S0 → a | XA | AX | b
S → a | XA | AX | b
A → RB
B → AX | b | a
X → a
R → XB Hence, for the given grammar, this is the required CNF.

Greibach Normal Form (GNF):

GNF stands for Greibach normal form.

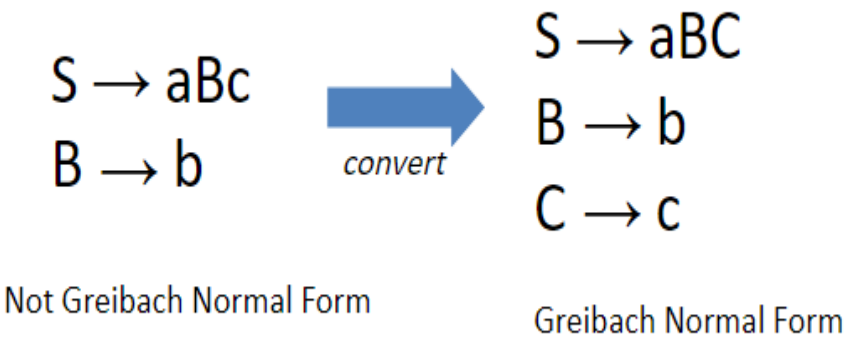
Definition 6: A CFG G = (V, T, P, S) is in Greiback normal form if for each product of G is of the form

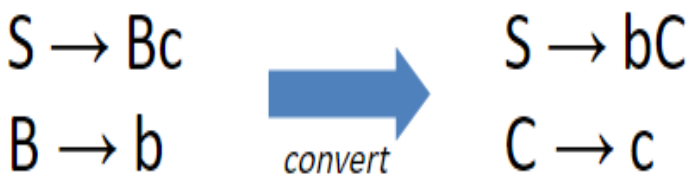
$A \rightarrow a\alpha$, where $A \in V$, $a \in T$ and $\alpha \in V^*$.

For example:

G1 = {S → aAB | aB, A → aA| a, B → bB | b}
G2 = {S → aAB | aB, A → aA | ε, B → bB | ε}

The production rules of Grammar G1 satisfy the rules specified for GNF, so the grammar G1 is in GNF. However, the production rule of Grammar G2 does not satisfy the rules specified for GNF as A → ε and B → ε contains ε(only start symbol can generate ε). So the grammar G2 is not in GNF.





Not Greibach Normal Form

Greibach Normal Form

Not in Greibach Normal Form
$L(G) = a^n b^n$

Greibach Normal Form
$L(G) = a^n b^n$

Steps to convert CFG to GNF

1. Simplify the CFG (*i.e., eliminate null production, unit production and useless symbols*) and convert to Chomsky Normal Form (CNF).
2. Convert the rules to ascending order. Rename the Non Terminals as A_1, A_2 …. with $S = A_1$.
3. For each production of the form $A_i \rightarrow A_j\alpha$, apply the following:
   (a) if $j>i$ – Leave the production as it is.
   (b) if $j=i$ – Apply elimination of left recursion rule.
   (c) if $j<i$ – Apply substitution rule.
4. For each production of the form $A_i \rightarrow A_j\alpha$, where $j>i$, apply the substitution lemma if A_j is in GNF, to bring A_i to GNF.

Example : Find equivalent grammar in GNF.

$$S \rightarrow aB \mid bA, A \rightarrow aS \mid bAA \mid a, B \rightarrow bS \mid aBB \mid b$$

Solution :

Renaming S, A and B by A_1, A_2, and A_3 respectively, we get the following productions.

$$A_1 \rightarrow aA_3 \mid bA_2, A_2 \rightarrow aA_1 \mid bA_2A_2 \mid a, A_3 \rightarrow bA_1 \mid aA_3A_3 \mid b$$

Since, all productions are in GNF, so there is no need of any modification.

Example 1 : Consider the CFG $S \rightarrow S+S \mid S*S \mid a \mid b$ and find an equivalent grammar in GNF.

Solution: Let G_1 is the equivalent grammar in GNF.

Renaming the variable, we get

$P_1 : S_1 \rightarrow S_1 + S_1$ (Not in GNF)

$P_2 : S_1 \rightarrow S_1 * S_1$ (Not in GNF)

$P_3 : S_1 \rightarrow a$ (In GNF)

$P_4 : S_1 \rightarrow b$ (In GNF)

P_1 and P_2 are left recursive productions, so removing the left recursion, we get

$S_1 \rightarrow aS_2 \mid bS_2 \mid a \mid b$, where

$S_2 \rightarrow +S_1S_2 \mid *S_1S_2 \mid +S_1 \mid *S_1$

Now, all productions are in GNF.

Example :-

A->BC
B->CA | b
C-> AB | a
Let A=A1, B=A2, C=A3 then

1)A1->A2A3 // j> i ,leave it as it is
2)A2->A3A1 | b // j> i ,leave it as it is
3)A3-> **A1**A2 |a // j< i do substitution A1->A2A3
A3-> **A2**A3A2 |a // j< i do substitution A2->A3A1 | b
A3-> A3A1A3A2 | bA3A2 |a // j = i , remove LR

A3 ->bA3A2A4 | aA4
A4 ->A1A3A2A4 | A1A3A2 | ε
// Now move by ε- production we get
A3 ->bA3A2A4 | aA4 | bA3A2 | a ----------A3-in GNF
A4 ->A1A3A2A4 | A1A3A2 --------not in GNF

A4 ->A1A3A2A4 | A1A3A2 ---------not in GNF
//now substitute A1->A2A3
A4 ->A2A3A3A2A4 | A2A3A3A2
//now substitute A2->A3A1
A4 ->A3A1A3A3A2A4 | A3A1A3A3A2
//now substitute A3 ->bA3A2A4 | aA4 | bA3A2 | a
A4 -> bA3A2A4A1A3A3A2A4 |
aA4A1A3A3A2A4 |
bA3A2A1A3A3A2A4 |
aA1A3A3A2A4 |
bA3A2A4A1A3A3A2 |
aA4A1A3A3A2 |
bA3A2A1A3A3A2 |
aA1A3A3A2 |

------A4----------now A4 is in GNF

Similarly by convert the A1 & A2 in GNF format

Resulting
A1->bA3A2A4A1A3 | aA4A1A3 |bA3A2A1A3 | aA1A3 ----------A1----------now A1 is in GNF

A2-> bA3A2B4A1| A4A1 | bA3A2A1 | aA1 | b----------A2----------now A2 is in GNF

104. Convert the following CFG to GNF

$S \rightarrow AA|a$

$A \rightarrow SS|b$

Solution:

Step 1: Simplify the CFG and convert to CNF

– Given grammar is in CNF form.

Step 2: Rename the variables as $S=A_1$, $A=A_2$

$A_1 \rightarrow A_2A_2 \mid a$ .... (1)

$A_2 \rightarrow A_1A_1 \mid b$ .... (2)

Step 3.1: In (1) $j>i$ so **leave the production as it is.**

Step 3.2: In (2) $j<i$ so apply **substitution rule.**

$$A_2 \rightarrow A_2A_2A_1 \mid aA_1 \mid b \quad \text{....(new 2)}$$

Step 3.3: In (new 2) $j=i$ so apply **elimination of left recursion rule.**

$$A_2 \rightarrow aA_1\mathbf{A_3} \mid b\mathbf{A_3} \mid aA_1 \mid b \quad \text{....(new 2)}$$

$$\mathbf{A_3} \rightarrow A_2A_1\mathbf{A_3} \mid A_2A_1 \quad \text{....(3)}$$

Step 3.4: In (3) $j<i$ so apply **substitution rule.**

$$A_3 \rightarrow aA_1A_3A_1A_3 \mid bA_3A_1A_3 \mid aA_1A_1A_3 \mid bA_1A_3 \mid aA_1A_3A_1 \mid bA_3A_1 \mid aA_1A_1 \mid bA_1 \quad \text{....(new 3)}$$

Step 4: Now (1) is in $A_i \rightarrow A_j\alpha$ where $j<i$ so we apply substitution rule to convert it to GNF.

$$A_1 \rightarrow aA_1A_2 \mid bA_2 \mid aA_1A_3A_2 \mid bA_3A_2 \mid a \quad \text{...(new 1)}$$

Final production rule:

$$A_1 \rightarrow aA_1A_2 \mid bA_2 \mid aA_1A_3A_2 \mid bA_3A_2 \mid a$$

$$A_2 \rightarrow aA_1A_3 \mid bA_3 \mid aA_1 \mid b$$

$$A_3 \rightarrow aA_1A_3A_1A_3 \mid bA_3A_1A_3 \mid aA_1A_1A_3 \mid bA_1A_3 \mid aA_1A_3A_1 \mid bA_3A_1 \mid aA_1A_1 \mid bA_1$$

Now the given CFG is converted to GNF.

Example:
S → XB | AA
A → a | SA
B → b
X → a

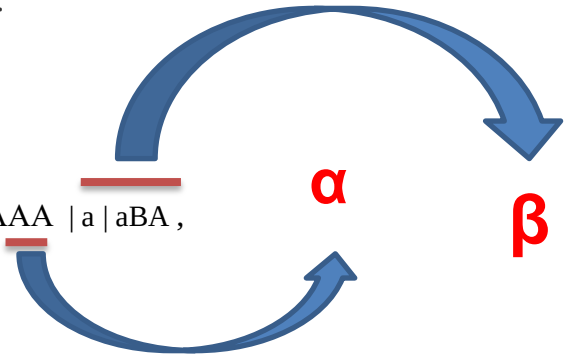
Solution:

As the given grammar G is already in CNF and there is no left recursion, so we can skip step 1 and step 2 anddirectly go to step 3.
The production rule A → SA is not in GNF, so we substitute S → XB | AA in the production rule A → SA as:
S → XB | AA
A → a | XBA | AAA

B → b
X → a

The production rule S → XB and B → XBA is not in GNF, so we substitute X → a in the production rule S →XB and B → XBA as:

S → aB | AA
A → a | aBA | AAA
B → b
X → a



To remove left recursion (A → AAA) from A → AAA | a | aBA ,
We get : S → aB | AA
A → aC | aBAC
C → AAC | ε
B → b
X → a

Now we will remove null production C → ε, we get:

S → aB | AA
A → aC | aBAC | a | aBA
C → AAC | AA
B → b
X → a

The production rule S → AA is not in GNF, so we substitute A → aC | aBAC | a | aBA in production rule S → AA as:

S → aB | aCA | aBACA | aA | aBAA
A → aC | aBAC | a | aBA
C → AAC
C → aCA | aBACA | aA | aBAA
B → b
X → a

The production rule C → AAC is not in GNF, so we substitute A → aC | aBAC | a | aBA in production rule C → AAC as:

S → aB | aCA | aBACA | aA | aBAA
A → aC | aBAC | a | aBA
C → aCAC | aBACAC | aAC | aBAAC
C → aCA | aBACA | aA | aBAA
B → b
X → a

Hence, this is the GNF form for the grammar G.

UNIT –II

Introduction to Compilers: Definition of compiler – Interpreterbootstrapping –phases of compiler. Lexical Analysis: Roles of Lexical analyzer –Input buffering – specification of tokens – Recognition of Tokens – A language for specifying lexical analyzers – design of a Lexical analyzer.

Introduction to Compilers

INTRODUCTION TO LANGUAGE PROCESSING

As Computers became inevitable and indigenous part of human life, and several languages withdifferent and more advanced features are evolved into this stream to satisfy or comfort the user in communicating with the machine , the development of the translators or mediator Software's have become essential to fill the huge gap between the human and machine understanding.

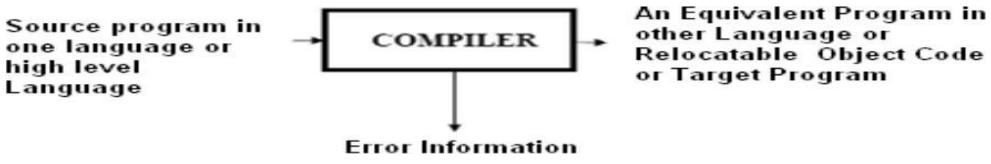
This process is called Language Processing to reflect the goal and intent of the process. On the way to this process to understand it in a better way, we have to be familiar with some key terms and concepts explained in following lines.

LANGUAGE TRANSLATORS

Is a computer program which translates a program written in one (Source) language to its equivalent program in other [Target]language. The Source program is a high level language where as the Target language can be any thing from the machine language of a target machine (between Microprocessor to Supercomputer) to another high level language program.

Two commonly Used Translators are Compiler and Interpreter

1. **Compiler** : Compiler is a program, reads program in one language called Source Language and translates in to its equivalent program in another Language called Target Language, in addition to this its presents the error information to the User.



If the target program is an executable machine-language program, it can then be called by the users to process inputs and produce outputs.

2. **Interpreter**: An interpreter is another commonly used language processor. Instead of producing a target program as a single translation unit, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

Difference between compiler and interpreter:

Compiler	Interpreter
It is a translator that translates high level to low level language	It is a translator that translates high level to low level language
It displays the errors after the whole program is executed.	It checks line by line for errors.
Examples: Basic, lower version of Pascal.	Examples: C, C++, Cobol, higher version of Pascal.

Definition of compile

Compiler: Is a translator that takes as input a source program written in high level language and converts it into its equivalent target program in machine language. In addition to above the compiler also

- Reports to its user the presence of errors in the source program.
- Facilitates the user in rectifying the errors, and execute the code.

Assembler: Is a program that takes as input an assembly language program and converts it into itsequivalent machine language code.

Loader / Linker: This is a program that takes as input a relocatable code and collects the libraryfunctions, relocatable object files, and produces its equivalent absolute machine code.

Specifically,

- **Loading** consists of taking the relocatable machine code, altering the relocatable addresses, and placing the altered instructions and data in memory at the proper locations.
- **Linking** allows us to make a single program from several files of relocatable machine code. These files may have been result of several different compilations, one or more may be library routines provided by the system available to any program that needs them.

In addition to these translators, programs like interpreters, text formatters etc., may be usedin language processing system. To translate a program in a high level language program to an executable one, the Compiler performs by default the compile and linking functions.

TYPES OF COMPILERS:

Based on the specific input it takes and the output it produces, the Compilers can be classifiedinto the following types;

Traditional Compilers(C, C++, Pascal): These Compilers convert a source program in a HLLinto its equivalent in native machine code or object code.

Interpreters(LISP, SNOBOL, Java1.0): These Compilers first convert Source code into intermediate code, and then interprets (emulates) it to its equivalent machine code.

Cross-Compilers: These are the compilers that run on one machine and produce code foranother machine.

Incremental Compilers: These compilers separate the source into user defined- steps; Compiling/recompiling step- by- step; interpreting steps in a given order Converters (e.g. COBOL to C++): These Programs will be compiling from one high level language to another.

Just-In-Time (JIT) Compilers (Java, Micosoft.NET): These are the runtime compilers from intermediate language (byte code, MSIL) to executable code or native machine code. These perform type -based verification which makes the executable code more trustworthy

Ahead-of-Time (AOT) Compilers (e.g., .NET ngen): These are the pre-compilers to the native code for Java and .NET

Binary Compilation: These compilers will be compiling object code of one platform into objectcode of another platform

Bootstraping

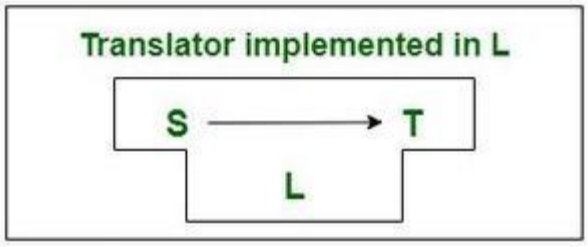
Bootstrapping is a process in which simple language is used to translate more complicated program which in turn may handle for more complicated program. This complicated program can further handle even more complicated program and so on. Writing a compiler for any high level language is a complicated process. It takes lot of time to write a compiler from scratch. Hence simple language is used to generate target code in some stages.

- Bootstrapping is widely used in the compilation development.
- Bootstrapping is used to produce a self-hosting compiler. Self-hosting compiler is a type of compiler that can compile its own source code.
- Bootstrap compiler is used to compile the compiler and then you can use this compiled compiler to compile everything else as well as future versions of itself.
- Bootstrapping is an important technique in compiler design that allows for greater control over the optimization and code generation process, while ensuring compatibility between the compiler and the target language.

A compiler can be characterized by three languages:

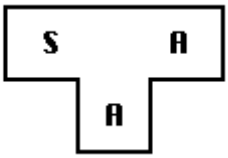
1. Source Language
2. Target Language
3. Implementation Language

The T- diagram shows a compiler SCIT for Source S, Target T, implemented in I.

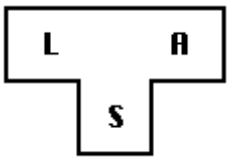


Follow some steps to produce a new language L for machine A:

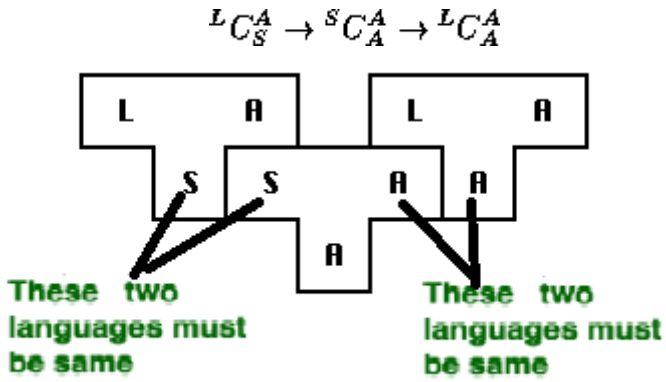
1. Create a compiler ${}_{A}{}^{S}C^{A}$ for subset, S of the desired language, L using language "A" and that compiler runs on machine A.



2.Create a compiler ${}^{L}C_{S}^{A}$ for language L written in a subset of L.



3. Compile $^{L}C^{A}_{S}$ using the compiler $^{S}C^{A}_{A}$ to obtain $^{L}C^{A}_{A}$. $^{L}C^{A}_{A}$ is a compiler for language L, which runs on machine A and produces code for machine A.



Advantages:

1. Bootstrapping ensures that the compiler is compatible with the language it is designed to compile, as it is written in the same language.
2. It allows for greater control over the optimization and code generation process.
3. It provides a high level of confidence in the correctness of the compiler because it is self-hosted.

Disadvantages:

1. It can be a time-consuming process, especially for complex languages or compilers.
2. Debugging a bootstrapped compiler can be challenging since any errors or bugs in the compiler will affect the subsequent versions of the compiler.
3. Bootstrapping requires that a minimal version of the compiler be written in a different language, which can introduce compatibility issues between the two languages.
4. Overall, bootstrapping is a useful technique in compiler design, but it requires careful planning and execution to ensure that the benefits outweigh the drawbacks.

Phases of compiler

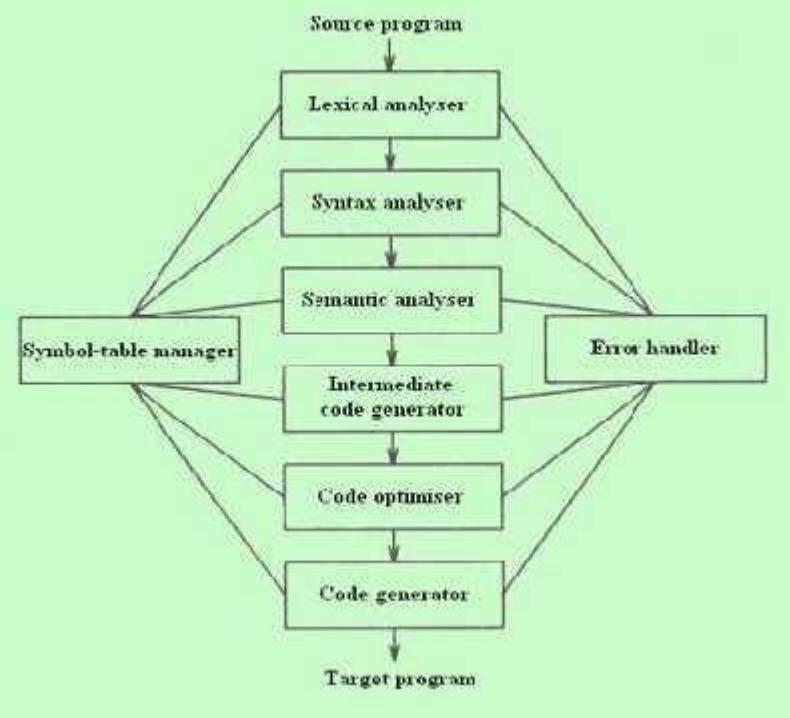
A Compiler operates in phases, each of which transforms the source program from onerepresentation into another. The following are the phases of the compiler:

Main phases:

1) Lexical analysis
2) Syntax analysis
3) Semantic analysis
4) Intermediate code generation
5) Code optimization
6) Code generation

Sub-Phases:

1) Symbol table management
2) Error handling

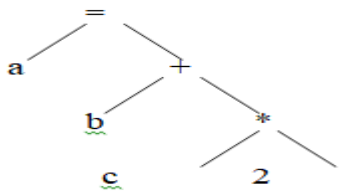


LEXICAL ANALYSIS:

- It is the first phase of the compiler. It gets input from the source program and producestokens as output.
- It reads the characters one by one, starting from left to right and forms the tokens.
- Token : It represents a logically cohesive sequence of characters such as keywords,operators, identifiers, special symbols etc.
  Example: a + b = 20
  Here, a,b,+,=,20 are all separate tokens.
  Group of characters forming a token is called the Lexeme.
- The lexical analyser not only generates a token but also enters the lexeme into the symboltable if it is not already there.

SYNTAX ANALYSIS:

- It is the second phase of the compiler. It is also known as parser.
- It gets the token stream as input from the lexical analyser of the compiler and generatessyntax tree as the output.
- Syntax tree:
  It is a tree in which interior nodes are operators and exterior nodes are operands.
- Example: For a=b+c*2, syntax tree is



SEMANTIC ANALYSIS:

- It is the third phase of the compiler.
- It gets input from the syntax analysis as parse tree and checks whether the given syntax iscorrect or not.
- It performs type conversion of all the data types into real data types.

INTERMEDIATE CODE GENERATION:

- It is the fourth phase of the compiler.
- It gets input from the semantic analysis and converts the input into output as intermediatecode such as three-address code.
- The three-address code consists of a sequence of instructions, each of which has atmostthree operands.
  Example: t1=t2+t3

CODE OPTIMIZATION:

- It is the fifth phase of the compiler.
- It gets the intermediate code as input and produces optimized intermediate code asoutput.
- This phase reduces the redundant code and attempts to improve the intermediate code sothat faster-running machine code will result.
- During the code optimization, the result of the program is not affected.
- To improve the code generation, the optimization involves
  - deduction and removal of dead code (unreachable code).
  - calculation of constants in expressions and terms.
  - collapsing of repeated expression into temporary string.
  - loop unrolling.
  - moving code outside the loop.
  - removal of unwanted temporary variables.

CODE GENERATION:

- It is the final phase of the compiler.
- It gets input from code optimization phase and produces the target code or object code asresult.
- Intermediate instructions are translated into a sequence of machine instructions thatperform the same task.
- The code generation involves
  - allocation of register and memory
  - generation of correct references
  - generation of correct data types
  - generation of missing code

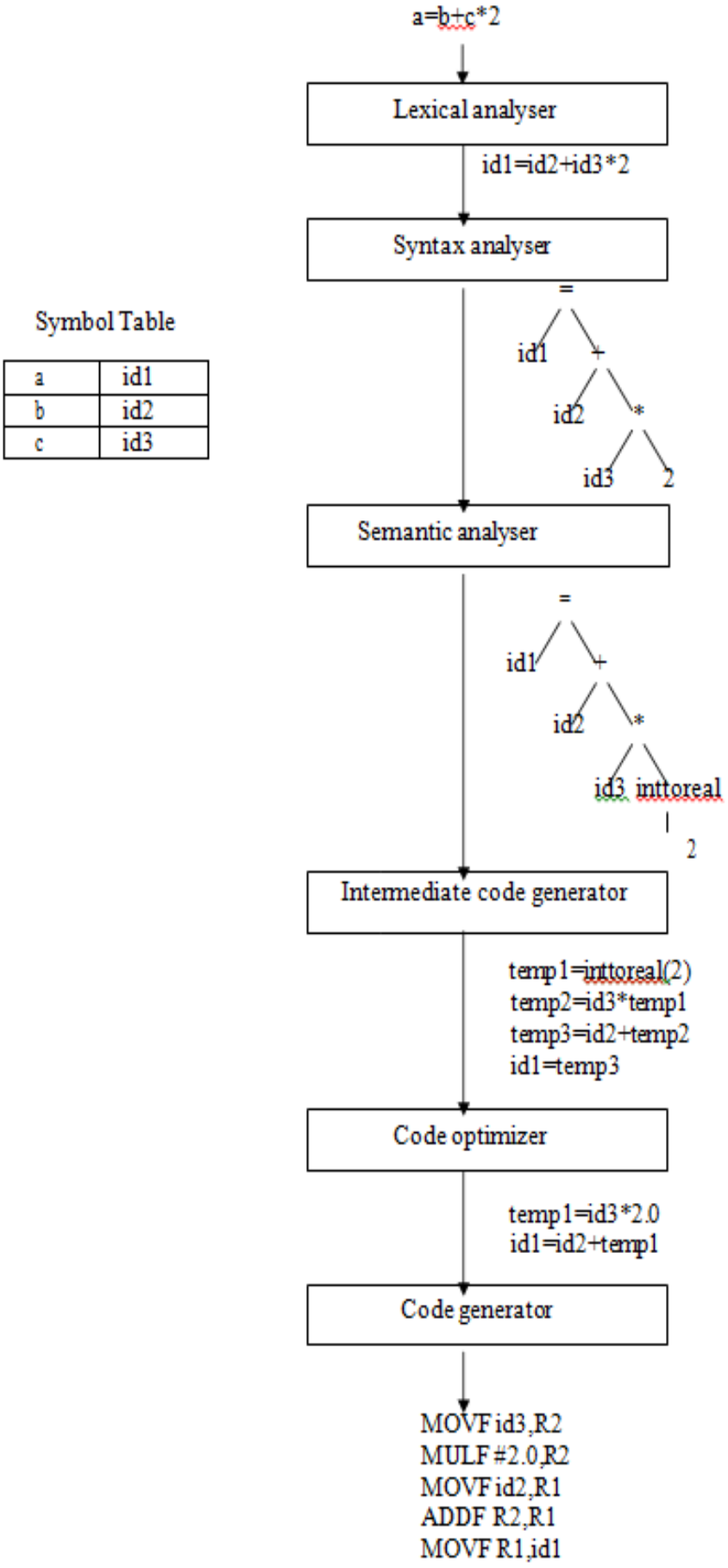
SYMBOL TABLE MANAGEMENT:

- Symbol table is used to store all the information about identifiers used in the program.
- It is a data structure containing a record for each identifier, with fields for the attributes ofthe identifier.
- It allows to find the record for each identifier quickly and to store or retrieve data fromthat record.
- Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

ERROR HANDLING:

- Each phase can encounter errors. After detecting an error, a phase must handle the errorso that compilation can proceed.
- In lexical analysis, errors occur in separation of tokens.
- In syntax analysis, errors occur during construction of syntax tree.
- In semantic analysis, errors occur when the compiler detects constructs with rightsyntactic structure but no meaning and during type conversion.
- In code optimization, errors occur when the result is affected by the optimization.
- In code generation, it shows error when code is missing etc.

To illustrate the translation of source code through each phase, consider the statement a=b+c*2.The figure shows the representation of this statement after each phase:


a=b+c*2
Lexical analyser
id1=id2+id3*2
Syntax analyser
Symbol Table
a id1
b id2
c id3
=
id1
+
id2
*
id3
2
Semantic analyser
=
id1
+
id2
*
id3
inttoreal
2
Intermediate code generator
temp1=inttoreal(2)
temp2=id3*temp1
temp3=id2+temp2
id1=temp3
Code optimizer
temp1=id3*2.0
id1=id2+temp1
Code generator
MOVF id3,R2
MULF #2.0,R2
MOVF id2,R1
ADDF R2,R1
MOVF R1,id1

COUSINS OF COMPILER

1. Preprocessor
2. Assembler
3. Loader and Link-editor

PREPROCESSOR

A preprocessor is a program that processes its input data to produce output that is used asinput to another program. The output is said to be a preprocessed form of the input data, which is often used by some subsequent programs like compilers.

They may perform the following functions :

1. Macro processing
2. File Inclusion
3. Rational Preprocessors
4. Language extension

1. Macro processing:

A macro is a rule or pattern that specifies how a certain input sequence should be mapped to an output sequence according to a defined procedure. The mapping process that instantiatesa macro into a specific output sequence is known as macro expansion.

2. File Inclusion:

Preprocessor includes header files into the program text. When the preprocessor finds an #include directive it replaces it by the entire content of the specified file.

3. Rational Preprocessors:

These processors change older languages with more modern flow-of-control and data-structuring facilities.

4. Language extension :

These processors attempt to add capabilities to the language by what amounts to built-in macros. For example, the language Equel is a database query language embedded in C.

ASSEMBLER

Assembler creates object code by translating assembly instruction mnemonics into machine code. There are two types of assemblers:

- One-pass assemblers go through the source code once and assume that all symbols will be defined before any instruction that references them.
- Two-pass assemblers create a table with all symbols and their values in the first pass, and then use the table in a second pass to generate code.

LINKER AND LOADER

A linker or link editor is a program that takes one or more objects generated by a compiler and combines them into a single executable program.

Three tasks of the linker are :

1. Searches the program to find library routines used by program, e.g. printf(), math routines.
2. Determines the memory locations that code from each module will occupy and relocates its instructions by adjusting absolute references
3. Resolves references among files.

A loader is the part of an operating system that is responsible for loading programs inmemory, one of the essential stages in the process of starting a program.

GROUPING OF THE PHASES

Compiler can be grouped into front and back ends:

- **Front end**: analysis (machine independent)

These normally include lexical and syntactic analysis, the creation of the symbol table,semantic analysis and the generation of intermediate code. It also includes error handling that goes along with each of these phases.

- **Back end:** synthesis (machine dependent)

It includes code optimization phase and code generation along with the necessary errorhandling and symbol table operations.

Compiler passes

A collection of phases is done only once (single pass) or multiple times (multi pass)

- **Single pass**: usually requires everything to be defined before being used in sourceprogram.
- **Multi pass**: compiler may have to keep entire program representation in memory.

Several phases can be grouped into one single pass and the activities of these phases are interleaved during the pass. For example, lexical analysis, syntax analysis, semantic analysis andintermediate code generation might be grouped into one pass.

COMPILER CONSTRUCTION TOOLS

These are specialized tools that have been developed for helping implement variousphases of a compiler. The following are the compiler construction tools:

1) Parser Generators:
-These produce syntax analyzers, normally from input that is based on a context-freegrammar.
-It consumes a large fraction of the running time of a compiler.
-Example-YACC (Yet Another Compiler-Compiler).

2) Scanner Generator:
-These generate lexical analyzers, normally from a specification based on regular expressions.
-The basic organization of lexical analyzers is based on finite automation.

3) Syntax-Directed Translation:
-These produce routines that walk the parse tree and as a result generate intermediate code.
-Each translation is defined in terms of translations at its neighbor nodes in the tree.

4) Automatic Code Generators:
-It takes a collection of rules to translate intermediate language into machine language. Therules must include sufficient details to handle different possible access methods for data.

5) Data-Flow Engines:
-It does code optimization using data-flow analysis, that is, the gathering of information abouthow values are transmitted from one part of a program to each other part.

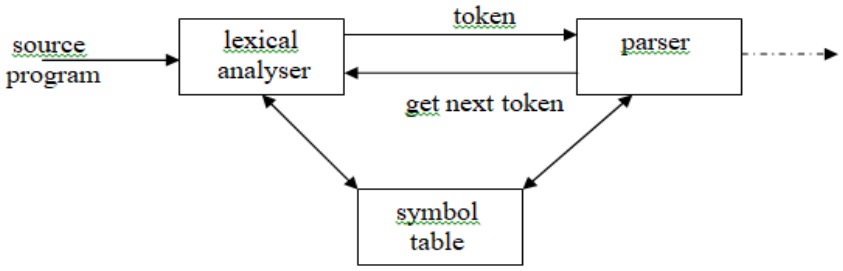
Roles of Lexical analyser

LEXICAL ANALYSIS

Lexical analysis is the process of converting a sequence of characters into a sequence oftokens. A program or function which performs lexical analysis is called a lexical analyzer or scanner. A lexer often exists as a single function which is called by a parser or another function.

THE ROLE OF THE LEXICAL ANALYZER

- The lexical analyzer is the first phase of a compiler.
- Its main task is to read the input characters and produce as output a sequence of tokensthat the parser uses for syntax analysis.



- Upon receiving a "get next token" command from the parser, the lexical analyzer readsinput characters until it can identify the next token.

ISSUES OF LEXICAL ANALYZER

There are three issues in lexical analysis:

- To make the design simpler.
- To improve the efficiency of the compiler.
- To enhance the computer portability.

TOKENS

A token is a string of characters, categorized according to the rules as a symbol (e.g., IDENTIFIER, NUMBER, COMMA). The process of forming tokens from an input stream ofcharacters is called tokenization.

A token can look like anything that is useful for processing an input text stream or textfile. Consider this expression in the C programming language: sum=3+2;

Lexeme	Token type
sum	Identifier
=	Assignment operator
3	Number
+	Addition operator
2	Number
;	End of statement

LEXEME:

Collection or group of characters forming tokens is called Lexeme.

PATTERN:

A pattern is a description of the form that the lexemes of a token may take.

In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

Attributes for Tokens

Some tokens have attributes that can be passed back to the parser. The lexical analyzer collects information about tokens into their associated attributes. The attributes influence the translation of tokens.

i) **Constant** : value of the constant
ii) **Identifiers**: pointer to the corresponding symbol table entry.

Example: In the following C language statement

printf ("Total = %d\n", score) ;

both **printf** and **score** are lexemes matching the pattern for token id, and **"Total = %d\n"** is a

lexeme matching literal [or string].

There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.

1. Simplicity of design is the most important consideration. The separation of Lexical and Syntactic analysis often allows us to simplify at least one of these tasks. For example, a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer.

2. Compiler efficiency is improved. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.

3. Compiler portability is enhanced: Input-device-specific peculiarities can be restricted to the lexical analyzer.

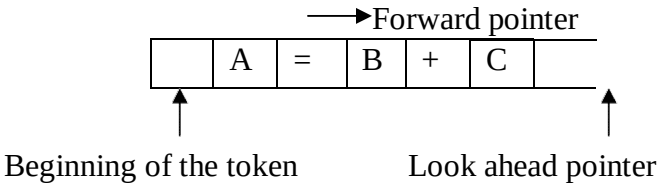
ERROR RECOVERY STRATEGIES IN LEXICAL ANALYSIS:

The following are the error-recovery actions in lexical analysis:
1) Deleting an extraneous character.
2) Inserting a missing character.
3) Replacing an incorrect character by a correct character.
4) Transforming two adjacent characters.
5) Panic mode recovery: Deletion of successive characters from the token until error is resolved.

Input buffering

INPUT BUFFERING

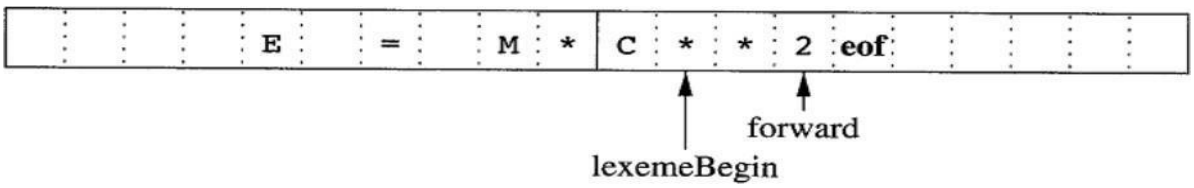
We often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. As characters are read from left to right, each character is stored in the buffer to form a meaningful token as shown below:



We introduce a two-buffer scheme that handles large look a heads safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

BUFFER PAIRS

Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character. An important scheme involves two buffers that are alternately reloaded.



Each buffer is of the same size N, and N is usually the size of a disk block, e.g., 4096 bytes. Using one system read command we can read N characters in to a buffer, rather than using one system call per character. If fewer than N characters remain in the input file, then a special character, represented by eof, marks the end of the source file and is different from any possible character of the source program. Two pointers to the input are maintained:

1. The Pointer **lexemeBegin**, marks the beginning of the current lexeme, whose extent we are attempting to determine.

2. Pointer **forward** scans ahead until a pattern match is found; the exact strategy whereby this determination is made will be covered in the balance of this chapter.

Once the next lexeme is determined, forward is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, 1exemeBegin is set to the character immediately after the lexeme just found. In Fig, we see forward has passed the end of the next lexeme, ** (the FORTRAN exponentiation operator), and must be retracted one position to its left.

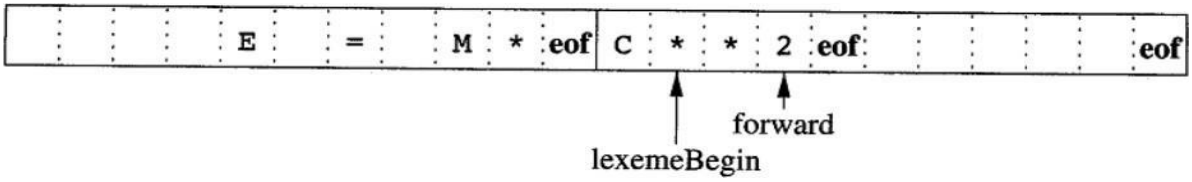
Advancing forward requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer.

As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greater than N, we shall never overwrite the lexeme in its buffer before determining it.

Sentinels To Improve Scanners Performance:

If we use the above scheme as described, we must check, each time we advance forward, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read (the latter may be a multi way branch).

We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a **sentinel** character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **eof**. Figure shows the same arrangement as Figure above but with the sentinels added. Note that eof retains its use as a marker for the end of the entire input.



Any eof that appears other than at the end of a buffer means that the input is at an end. Below Figure summarizes the algorithm for advancing forward. Notice how the first test, which can be part of a multiway branch based on the character pointed to by forward, is the only test we make, except in the case where we actually are at the end of a buffer or the end of the input.

```
switch ( *forward++ )
 {
 case eof: if (forward is at end of first buffer ){
                    reload second buffer;
                    forward = beginning of second buffer;}
                  else if (forward is at end of second buffer
                  ) {
                    reload first buffer;
                    forward = beginning of first buffer;
                  }
                    else /* eof within a buffer marks the end of input
                      */terminate lexical analysis;
                 break;
                 }
```

Specification of tokens

Introduction:-

To understand how the language theory, following terms are to be understood:

1. Alphabets
2. Strings
3. Special symbols
4. Language
5. Longest match rule
6. Operations
7. Notations
8. Representing valid tokens of a language in regular expression
9. Finite automata

1. Alphabets: Any finite set of symbols

- {0,1} is a set of binary alphabets,
- {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of Hexadecimal alphabets,
- {a-z, A-Z} is a set of English language alphabets.

2. Strings: Any finite sequence of alphabets is called a string.

3. Special symbols: A typical high-level language contains the following symbols:

Arithmetic Symbols	Addition(+), Subtraction(-), Multiplication(*), Division(/)
Punctuation	Comma(,), Semicolon(;), Dot(.)
Assignment	=
Special assignment	+=, -=, *=, /=
Comparison	==, !=. <. <=. >, >=
Preprocessor	#

4. Language: A language is considered as a finite set of strings over some finite set of alphabets.

5. Longest match rule: When the lexical analyzer read the source-code, it scans the code letter by letter and when it encounters a whitespace, operator symbol, or special symbolsit decides that a word is completed.

6. Operations: The various operations on languages are:

- Union of two languages L and M is written as, L U M = {s | s is in L or s is in M}
- Concatenation of two languages L and M is written as, LM = {st | s is in L and t is in M}
- The Kleene Closure of a language L is written as, L* = Zero or more occurrence of language L.

7. Notations: If r and s are regular expressions denoting the languages L(r) and L(s), then

- **Union** : L(r)UL(s)
- **Concatenation** : L(r)L(s)
- **Kleene closure** : (L(r))*

8. Representing valid tokens of a language in regular expression: If x is a regular expression, then

- x* means zero or more occurrence of x.
- x+ means one or more occurrence of x.

9. Finite automata: Finite automata is a state machine that takes a string of symbols as inputand changes its state accordingly. If the input string is successfully processed and the automata reaches its final state, it is accepted. The mathematical model of finite automata consists of:

- Finite set of states (Q)
- Finite set of input symbols (Σ)
- One Start state (q0)
- Set of final states (qf)
- Transition function (δ)

The transition function (δ) maps the finite set of state (Q) to a finite set of input symbols (Σ),
$Q \times \Sigma \rightarrow Q$

SPECIFICATION OF TOKENS

There are 3 specifications of tokens:
1) Strings
2) Language
3) Regular expression

An **alphabet** or character class is a finite set of symbols.

A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet

A **language** is any countable set of strings over some fixed alphabet.

In language theory, the terms "sentence" and "word" are often used as synonyms for "string." The length of a string s, usually written |s|, is the number of occurrences of symbols in s.For example, banana is astring of length six. The empty string, denoted ε, is the string of length zero.

Operations on strings

The following string-related terms are commonly used:

1. A **prefix** of string s is any string obtained by removing zero or more symbols from the end of string s. For example, ban is a prefix of banana.

2. A **suffix** of string s is any string obtained by removing zero or more symbols from the beginning of s.
   For example, nana is a suffix of banana.
3. A **substring** of s is obtained by deleting any prefix and any suffix from s.Forexample, nan is a substring of banana.
4. The **proper prefixes, suffixes, and substrings** of a string s are those prefixes, suffixes, andsubstrings, respectively of s that are not ε or not equal to s itself.
5. A **subsequence** of s is any string formed by deleting zero or more not necessarily consecutive positions of s.
   For example, baan is a subsequence of banana.

Operations on languages:

The following are the operations that can be applied to languages:

1. Union
2. Concatenation
3. Kleene closure
4. Positiveclosure

The following example shows the operations on strings:

Let L={0,1} and S={a,b,c}

1. Union : L U S={0,1,a,b,c}
2. Concatenation : L.S={0a,1a,0b,1b,0c,1c}
3. Kleene closure : L^*={ ε,0,1,00….}
4. Positive closure : L^+={0,1,00….}

Regular Expressions

Each regular expression r denotes a language L(r).

Here are the rules that define the regular expressions over some alphabet Σ and the languages that those expressions denote:

1. ε is a regular expression, and L(ε) is { ε }, that is, the language whose sole member is the emptystring.
2. If ‘a’ is a symbol in Σ, then ‘a’ is a regular expression, and L(a) = {a}, that is, the language with onestring, of length one, with ‘a’ in its one position.
3. Suppose r and s are regular expressions denoting the languages L(r) and L(s). Then,
   a) (r)|(s) is a regular expression denoting the language L(r) U L(s).
   b) (r)(s) is a regular expression denoting the language L(r)L(s).
   c) (r)* is a regular expression denoting (L(r))*.
   d) (r) is a regular expression denoting L(r).
4. The unary operator * has highest precedence and is left associative.
5. Concatenation has second highest precedence and is left associative.
6. has lowest precedence and is left associative

Regular set

A language that can be defined by a regular expression is called a regular set.
If two regular expressions r and s denote the same regular set, we say they are equivalent andwrite r = s.

There are a number of algebraic laws for regular expressions that can be used tomanipulate intoequivalent forms.
For instance, r|s = s|r is commutative; r|(s|t)=(r|s)|t is associative.

Regular Definitions

Giving names to regular expressions is referred to as a Regular definition. If Σ is an alphabet ofbasic symbols, then a regular definition is a sequence of definitions of the form

$d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
……$d_n \rightarrow r_n$

1. Each d_i is a distinct name.
2. Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$.

Example: Identifiers is the set of strings of letters and digits beginning with a letter. Regulardefinition for this set:
letter → A | B | …. | Z | a | b | …. | z |digit → 0
| 1 | …. | 9
id → letter (letter | digit) *

Shorthands

Certain constructs occur so frequently in regular expressions that it is convenient tointroducenotational shorthands for them.

1. *One or more instances (+)*:
- The unary postfix operator + means " one or more instances of" .
- If r is a regular expression that denotes the language L(r), then $(r)^+$ is a regular expressionthatdenotes the language $(L (r))^+$
- Thus the regular expression a^+ denotes the set of all strings of one or more a's.
- The operator $^+$ has the same precedence and associativity as the operator *.

2. *Zero or one instance (?)*:
- The unary postfix operator ? means "zero or one instance of".
- The notation r? is a shorthand for r | ε.
- *If 'r' is a regular expression, then (r)? is a regular expression that denotes the languageL(r) U {ε }.*

3. *Character Classes*:
- The notation [abc] where a, b and c are alphabet symbols denotes the regular expressiona | b | c.
- Character class such as [a – z] denotes the regular expression a | b | c | d | ….|z.
- We can describe identifiers as being strings generated by the regular expression,[A–Za–z][A–Za–z0–9]*

Non-regular Set

A language which cannot be described by any regular expression is a non-regular set.
Example: The set of all strings of balanced parentheses and repeating strings cannot be described by a regular expression. This set can be specified by a context-free grammar.

Recognition of Tokens

Consider the following grammar fragment:

stmt → if expr then stmt
| if expr then stmt else stmt
| ε

expr → term relop term
| term

term → id
| num

where the terminals if , then, else, relop, id and num generate sets of strings given by thefollowing regular definitions:

if → if
then → then
else → else
relop → <|<=|=|<>|>|>=
id → letter(letter|digit)*
num → digit+ (.digit+)?(E(+|-)?digit+)?

For this language fragment the lexical analyzer will recognize the keywords if, then, else,as well asthe lexemes denoted by relop, id, and num. To simplify matters, we assume keywords are reserved; that is,they cannot be used as identifiers.

Transition diagrams

It is a diagrammatic representation to depict the action that will take place when a lexicalanalyzer iscalled by the parser to get the next token. It is used to keep track of information aboutthe characters that areseen as the forward pointer scans the input.

Transition diagram for relational operators

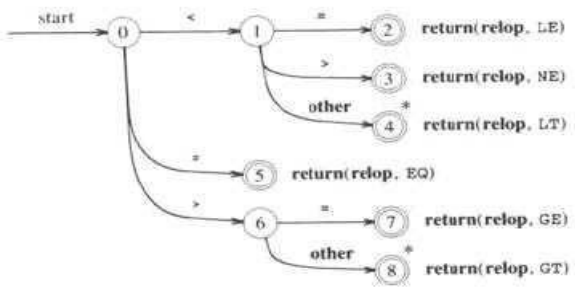
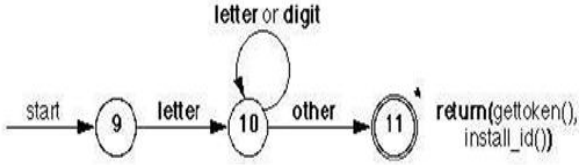


Fig. 3.12. Transition diagram for relational operators.

Transition diagram for identifiers and keywords



A language for specifying lexical analyzers

There is a wide range of tools for constructing lexical analyzers.

1. Lex
2. YACC

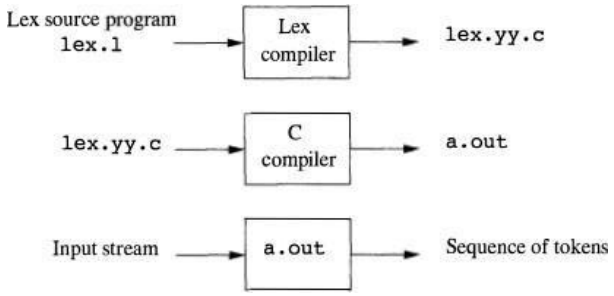
Lex is a computer program that generates lexical analyzers. Lex is commonly used withthe yacc parser generator.

1. 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 sequenceof tokens.
- It reads the input stream and produces the source code as output throughimplementing 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 compilerruns 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 Specification

A Lex program consists of three parts:

{ definitions }
%%
{ rules }
%%
{ user subroutines }

Definitions include declarations of variables, constants, and regular definitions

Rules are statements of

theformp1
{action1}
p2 {action2}

. . .

p_n {$action_n$}

where p_i is regular expression and $action_i$ describes what action the lexical analyzershould take when pattern p_i matches a lexeme. Actions are written in C code.

User subroutines are auxiliary procedures needed by the actions. Thesecan be compiled separately and loaded with the lexical analyzer.

2. YACC- YET ANOTHER COMPILER-COMPILER

- YACC stands for **Yet Another Compiler Compiler**.
- YACC provides a tool to produce a parser for a given grammar.
- YACC is a program designed to compile a LALR (1) grammar.
- It is used to produce the source code of the syntactic analyzer of the languageproduced by LALR (1) grammar.
- The input of YACC is the rule or grammar and the output is a C program.

These are some points aboutYACC:

Input: A CFG- file.y

Output: A parser y.tab.c (yacc)

- The output file "file.output" contains the parsing tables.
- The file "file.tab.h" contains declarations.
- The parser called the yyparse ().
- Parser expects to use a function called yylex () to get tokens.

The basic operational sequence is as follows:

This file contains the desired grammar in YACC format.

It shows the YACC program.

It is the c source program created by YACC.

C Compiler

Executable file that will parse grammar given in gram.Y

Example 1:- /*lex program to count number of words*/

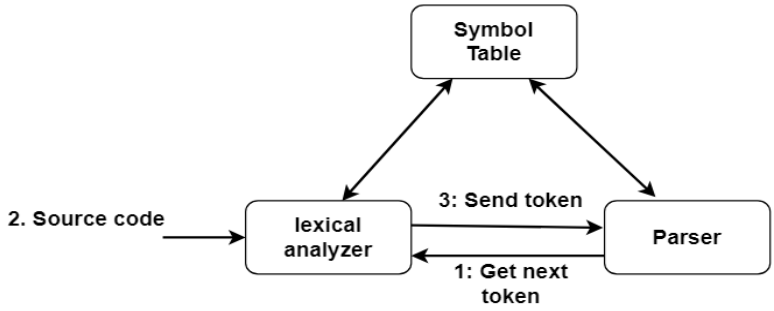
```
%{
#include<stdio.h>
#include<string.h>
int i = 0;
%}

/* Rules Section*/
%%
([a-zA-Z0-9])* {i++;} /* Rule for counting   number of words*/

"\n" {printf("%d\n", i); i = 0;}
%%

int yywrap(void){}

int main()
{
        // The function that starts the analysis
        yylex();

        return 0;
}
```

Example2:- The following Lex Program counts the number of words-

```
%{
#include<stdio.h>
#include<string.h>
int count = 0;
%}

/* Rules Section*/
%%
([a-zA-Z0-9])*    {count++;}  /* Rule for counting number of words*/

"\n" {printf("Total Number of Words : %d\n", count); count = 0;}
%%

int yywrap(void){}

int main()
{
   // The function that starts the analysis
   yylex();

   return 0;
}
```

DESIGN OF LEXICAL ANALYSER

Lexical Analysis is the first phase of compiler design where input is scanned to identify tokens

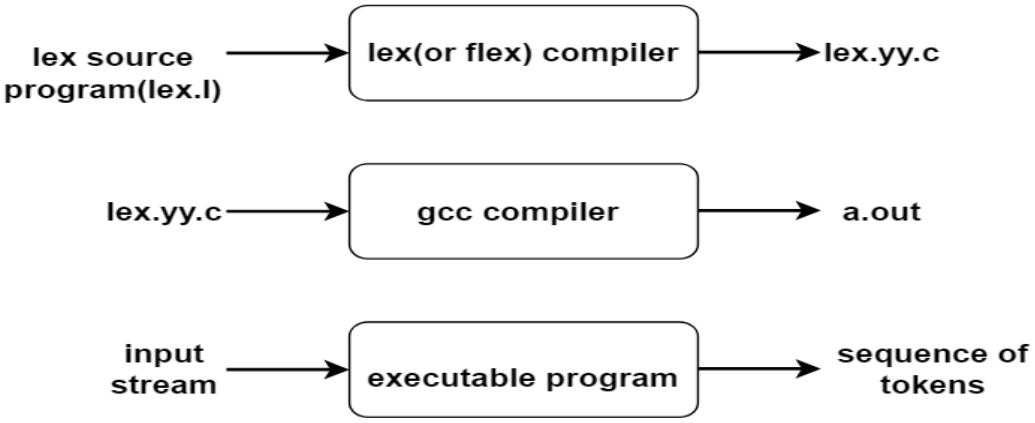


A **lexeme** is an instance of a token.
A **token** is a sequence of characters representing a unit of information in the source program.
We can either hand code a lexical analyzer or use a lexical analyzer generator to design a lexical analyzer.
Hand-coding the steps involve the following;

1. Specification of tokens by use of regular expressions.
2. Construction of a finite automata equivalent to a regular expression.
3. Recognition of tokens by a constructed finite automata.

Generating a lexical analyzer.

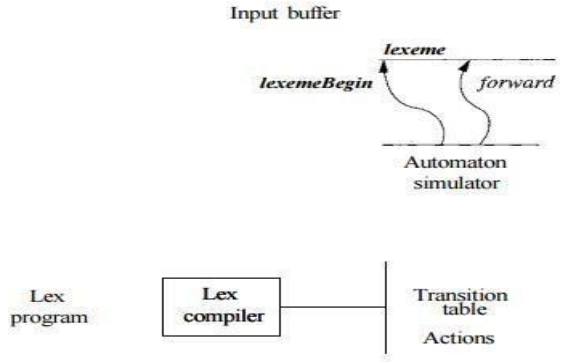
A ***lex*** or ***flex*** is a program that is used to generate a lexical analyzer.
These translate regular definitions into C source code for efficient lexical analysis.



A ***lexical analyzer generator*** systematically translates regular expressions to *NFA* which is then translated to an efficient *DFA*.

1. The Structure of the Generated Analyzer

Below Figure Overviews the architecture of a lexical analyzer generated by Lex. The program that serves as the lexical analyzer includes a fixed program that simulates an automaton; at this point we leave open whether that automaton is deterministic or nondeterministic. The rest of the lexical analyzer consists of components that are created from the Lex program by Lex itself.



In above diagram explains a Lex program is turned into a transition table and actions, which are used by a finite-automaton simulator

These components are:

- A transition table for the automaton.
- Those functions that are passed directly through Lex to the output
- The actions from the input program, which appear as fragments of code tobeinvoked at the appropriate time by the automaton simulator.

To construct the automaton, we begin by taking each regular-expression pattern in the Lex program and converting it to an NFA. We need a single automaton that will recognize lexemes matching any of the patterns in the program, so we combine all the NFA's into one by introducing a new start state with e-transitions to each of the start states of the NFA's *N{* for pattern *pi*. This construction is shown in Fig. 3.50.

Example: We shall illustrate the ideas of this section with the following simple, abstract example:

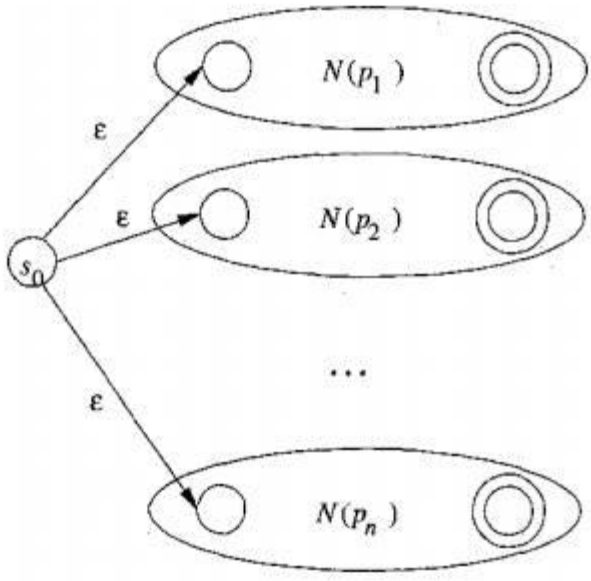


Figure 3.50: An NFA constructed from a **Lex** program

a	{ action A_1 for pattern p_1 }
abb	{ action A_2 for pattern p_2 }
a*b+	{ action A_3 for pattern p_3 }

Note that these three patterns present some conflicts of the type discussed in earlier. In particular, string abb matches both the second and third patterns, but we shall considerit a lexeme for pattern p2, since that pattern is listed first in the above Lex program. Then, input strings such as aabbb • • • have many prefixes that match the third pattern. The Lex rule is to take the longest, so we continue reading 6's, until another a is met, whereupon we report the lexeme to be the initial a's followed by as many 6's as there are.

Three NFA's that recognize the three patterns. The third is a simplification of whatwould come out of Algorithm. Then, Fig. 3.52 shows these three NFA's combined into a single NFA by the addition of start state 0 and three e- transitions. •

2. Pattern Matching Based on NFA's

If the lexical analyzer simulates an NFA such as that of Fig. 3.52, then it must read input beginning at the point on its input which we have referred to as *lexemeBegin.* As it moves the pointer called *forward* ahead in the input, it calculates the set of states it is in at each point, following Algorithm.

Eventually, the NFA simulation reaches a point on the input where there are no next states. At that point, there is no hope that any longer prefix of the input would ever get the NFA to an accepting state; rather, the set of states will always be empty. Thus,we are ready to decide on the longest prefix that is a lexeme matching some pattern.

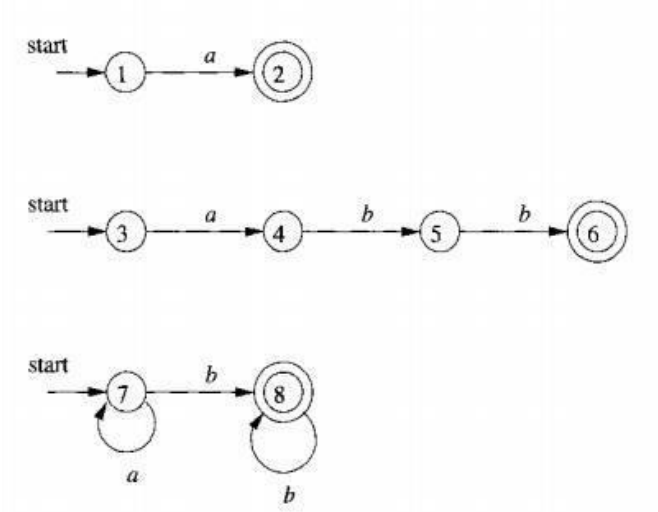


Figure 3.51: NFA's for **a**, **abb**, and **a*b+**

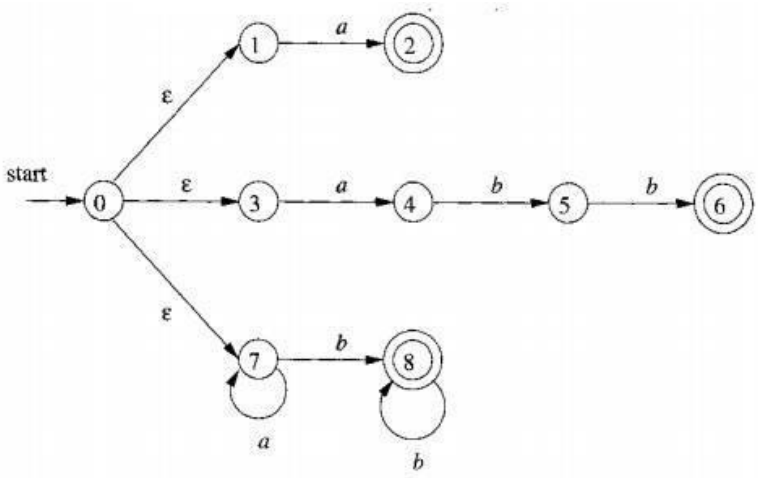


Figure 3.52: Combined NFA

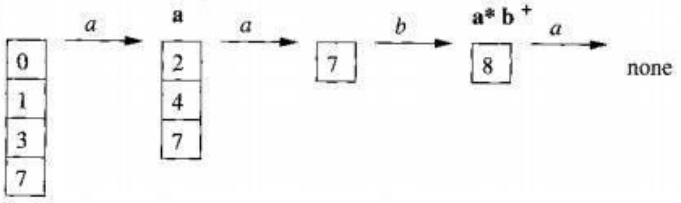


Figure 3.53: Sequence of sets of states entered when processing input *aaba*

We look backwards in the sequence of sets of states, until we find a set that includes one or more accepting states. If there are several accepting states in that set, pick the one associated with the earliest pattern *pi* in the list from the **Lex** program. Move the *forward* pointer back to the end of the lexeme, and perform the action *Ai* associated with pattern *pi*.

Example: Suppose we have the patterns of the input begins *aaba*. Figure 3.53 shows the sets of states of the NFA of Fig. 3.52 that we enter, starting with *e- closure* of the initial state 0, which is {0,1,3,7}, and proceeding from there. After reading the fourth input symbol, we are in an empty set of states, since in Fig. 3.52, there are no transitions out ofstate 8 on input *a*.

Thus, we need to back up, looking for a set of states that includes an accepting state. Notice that, as indicated in Fig. 3.53, after reading a we are in a set that includes state 2 and therefore indicates that the pattern a has been matched. However, after reading aab, we are in state 8, which indicates that a *b + has been matched; prefix aab is the longest prefix that gets us to an accepting state. We therefore select aab as the lexeme, and execute action A3, which should include a return to the parser indicating that the token whose pattern is p3 = a * b + has been found. •

3. DFA's for Lexical Analyzers

Another architecture, resembling the output of Lex, is to convert the NFA for all the patterns into an equivalent DFA, using the subset construction of Algorithm. Within each DFA state, if there are one or more accepting NFA states, determine the first pattern whose accepting state is represented, and make that pattern the output of the DFA state.

Example : Figure 3.54 shows a transition diagram based on the DFA that is constructed by the subset construction from the NFA in Fig. 3.52. The accepting states are labeled by the pattern that is identified by that state. For instance, the state {6,8} has two accepting states, corresponding to patterns **abb** and **a * b** $^{+}$ **.** Since the formeris listed first, that is the pattern associated with state {6,8} . •

We use the DFA in a lexical analyzer much as we did the NFA. We simulate the DFA until at some point there is no next state (or strictly speaking, the next stateis 0, the *dead state* corresponding to the empty set of NFA states). At that point, we back up through the sequence of states we entered and, as soon as we meet an accepting DFA state, we perform the action associated with the pattern for that state.

Example : Suppose the DFA of Fig. 3.54 is given input *abba*. The se-quence ofstates entered is 0137,247,58,68, and at the final *a* there is no tran-sition out of state 68. Thus, we consider the sequence from the end, and in this case, 68 itself is an accepting state that reports pattern *p2* = **abb . •**

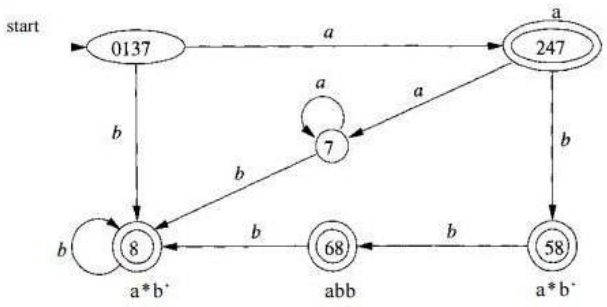


Figure 3.54: Transition graph for DFA handling the patterns **a**, **abb**, and **a*b***

4. Implementing the Lookahead Operator

Lex lookahead operator / in a **Lex** pattern *r\/r2* is sometimes necessary, because the pattern r*i for a particular token may need to describe some trailing context *r2* inorder to correctly identify the actual lexeme. When converting the pattern *r\/r2* to an NFA, we treat the / as if it were e, so we do not actually look for a / on the input. However, if the NFA recognizes a prefix *xy* of the input buffer as matching this regular expression, the end of the lexeme is not where the NFA entered its accepting state. Rather the end occurs when the NFA enters a state *s* such that

1. *s* has an e-transition on the (imaginary) /,
2. There is a path from the start state of the NFA to state *s* that spells out *x.*
3. There is a path from state *s* to the accepting state that spells out *y.*
4. *x* is as long as possible for any *xy* satisfying conditions **1-3.**

If there is only one e-transition state on the imaginary / in the NFA, then the end of the lexeme occurs when this state is entered for the last time as the following example illustrates. If the NFA has more than one e-transition state on the imaginary /, then the general problem of finding the correct state *s* is much more difficult.

Dead States in DFA's

Technically, the automaton in Fig. 3.54 is not quite a DFA. The reasonis that a DFA has a transition from every state on every input symbol in its input alphabet. Here, we have omitted transitions to the dead state 0, and we have therefore omitted the transitions from the dead state to itself on every input. Previous NFA-to-DFA examples did not have a way to get from the start state to 0, but the NFA of Fig. 3.52 does.

However, when we construct a DFA for use in a lexical analyzer, it is important that we treat the dead state differently, since we must know when there is no longer any possibility of recognizing a longer lexeme. Thus, we suggest always omitting transitions to the dead state and elimi-nating the deadstate itself. In fact, the problem is harder than it appears, since an NFA-to-DFAconstruction may yield several states that cannot reach any accepting state, and we must know when any of these states have been reached.

Combine all these states into one dead state, so their identification becomes easy. It is also interesting to note that if we construct a DFA from a regular expression, then the DFA will not have any states besides 0 that cannot lead to an accepting

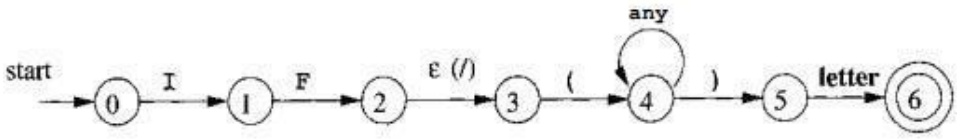


Figure 3.55: NFA recognizing the keyword IF

UNIT III

❖ THE ROLE OF PARSER

The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language. It reports any syntax errors in the program. It also recovers from commonly occurring errors so that it can continue processing its input.

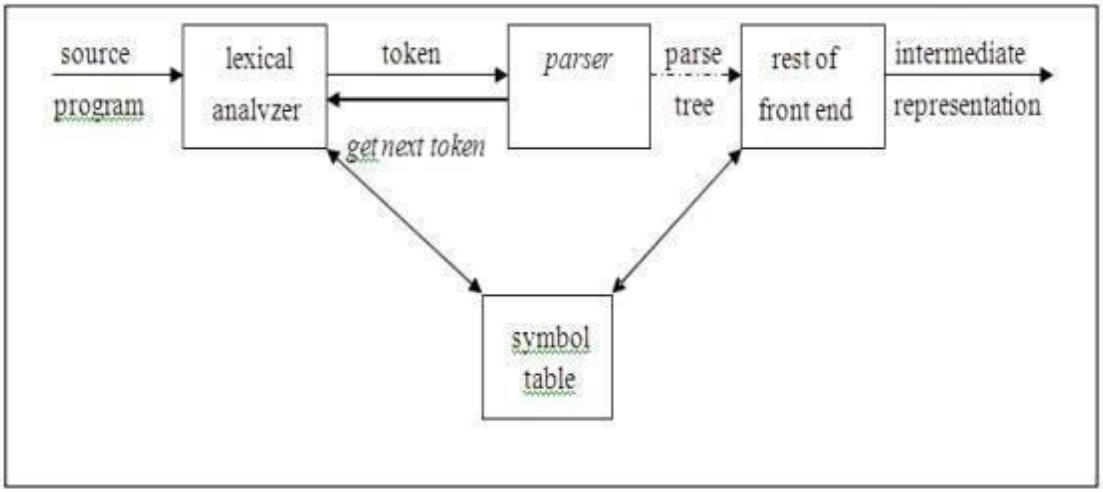


Fig. 2.1 Position of parser in compiler model

1. It verifies the structure generated by the tokens based on the grammar.
2. It constructs the parse tree.
3. It reports the errors.
4. It performs error recovery.

Issues :

Parser cannot detect errors such as:

1. Variable re-declaration
2. Variable initialization before use
3. Data type mismatch for an operation.

The above issues are handled by Semantic Analysis phase.

Syntax error handling:

Programs can contain errors at many different levels. For example :

1. Lexical, such as misspelling an identifier, keyword or operator.
2. Syntactic, such as an arithmetic expression with unbalanced parentheses.
3. Semantic, such as an operator applied to an incompatible operand.
4. Logical, such as an infinitely recursive call.

Functions of error handler:

1. It should report the presence of errors clearly and accurately.
2. It should recover from each error quickly enough to be able to detect subsequent errors.
3. It should not significantly slow down the processing of correct programs.

Error recovery strategies:

The different strategies that a parse uses to recover from a syntactic error are:

1. Panic mode
2. Phrase level
3. Error productions
4. Global correction

Panic mode recovery:

On discovering an error, the parser discards input symbols one at a time until a synchronizing token is found. The synchronizing tokens are usually delimiters, such as semicolon or end. It has the advantage of simplicity and does not go into an infinite loop. When multiple errors in the same statement are rare, this method is quite useful.

Phrase level recovery:

On discovering an error, the parser performs local correction on the remaining input that allows it to continue. Example: Insert a missing semicolon or delete an extraneous semicolon etc.

Error productions:

The parser is constructed using augmented grammar with error productions. If an error production is used by the parser, appropriate error diagnostics can be generated to indicate the erroneous constructs recognized by the input.

Global correction:

Given an incorrect input string x and grammar G, certain algorithms can be used to find a parse tree for a string y, such that the number of insertions, deletions and changes of tokens is as small as possible. However, these methods are in general too costly in terms of time and space.

CONTEXT-FREE GRAMMARS

A Context-Free Grammar is a quadruple that consists of
terminals,
Non-terminals,
start symbol and
productions.

Terminals: These are the basic symbols from which strings are formed.
Non-Terminals: These are the syntactic variables that denote a set of strings.
These help to define the language generated by the grammar.

Start Symbol: One non-terminal in the grammar is denoted as the "Start-symbol" and the set of strings it denotes is the language defined by the grammar.

Productions : It specifies the manner in which terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal, followed by an arrow, followed by a string of non-terminals and terminals.

Example of context-free grammar:

The following grammar defines simple arithmetic expressions
: expr → expr op expr expr → (expr)

expr → - expr
expr → id
op → +
op → -
op → *
op → /
op → ↑

In this grammar,

id + - * / ↑ ( ) are terminals.

expr , op are non-terminals.
expr is the start symbol.
Each line is a production.

Derivations:

Two basic requirements for a grammar are :
1. To generate a valid string.
2. To recognize a valid string.

Derivation is a process that generates a valid string with the help of grammar by replacing the non-terminals on the left with the string on the right side of the production.

Example : Consider the following grammar for arithmetic expressions :

E→E+E|E*E|(E)|-E| id

To generate a valid string - (id+id) from the grammar the steps are
1. E → - E
2. E → - (E)
3. E → - (E+E)
4. E → - (id+E)
5. E → - (id+id)

In the above derivation,

E is the start symbol

-(id+id) is the required sentence(only terminals).

Strings such as E, -E, -(E), . . . are called sentinel forms.

Types of derivations:

The two types of derivation are:

1. Left most derivation
2. Right most derivation.

ϖ In leftmost derivations, the leftmost non-terminal in each sentinel is always chosen first for replacement.

In rightmost derivations, the rightmost non-terminal in each sentinel is always chosen first for replacement.

Example:

Given grammar G : E → E+E | E*E | (E) | - E | id Sentence to be derived : - (id+id)

Left Most Derivation

E→-E

E→-(E)

E→-(E+E)

E→-(id+E)

E→-(id+id)

Right Most Derivation

E→-E

E→-(E)

E→-(E+E)

E→-(E+id)

E→-(id+id)

String that appear in leftmost derivation are called left sentinel forms.

String that appear in rightmost derivation are called right sentinel forms.

Sentinels:

Given a grammar G with start symbol S, if S → α , where α may contain non-terminals or terminals, then α is called the sentinel form of G.

Yield or frontier of tree:

Each interior node of a parse tree is a non-terminal. The children of node can be a terminal or non-terminal of the sentinel forms that are read from left to right. The sentinel form in the parse tree is called yield or frontier of the tree.

Ambiguity:

A grammar that produces more than one parse for some sentence is said to be ambiguous

grammar.

Example : Given grammar G : E → E+E | E*E | (E) | - E | id

The sentence id+id*id has the following two distinct leftmost derivations:

E → E+ E

E → E* E

E → id + E

E→E+ E * E

E → id + E * E

E → id + E * E

E → id + id * E

E → id + id * E

E → id + id * id

E → id + id * id

The two corresponding trees are,

Fig. 2.2 Two parse trees for id+id*id

WRITING A GRAMMAR

A *grammar* consists of a number of *productions*. Each production has an abstract symbol called a *nonterminal* as its *left-hand side*, and a sequence of one or more nonterminal and *terminal* symbols as its *right-hand side*. For each grammar, the terminal symbols are drawn from a specified *alphabet*.

There are four categories in writing a grammar :

1. Regular Expression Vs Context Free Grammar

2. Eliminating ambiguous grammar.
3. Eliminating left-recursion
4. Left-factoring.

REGULAR EXPRESSION	CONTEXT-FREE GRAMMAR
It is used to describe the tokens of programming languages.	It consists of a quadruple where S → start symbol, P → production, T → terminal, V → variable or non- terminal.
It is used to check whether the given input is valid or not using **transition diagram.**	It is used to check whether the given input is valid or not using **derivation**.
The transition diagram has set of states and edges.	The context-free grammar has set of productions.
It has no start symbol.	It has start symbol.
It is useful for describing the structure of lexical constructs such as identifiers, constants, keywords, and so forth.	It is useful in describing nested structures such as balanced parentheses, matching begin-end's and so on.

Eliminating ambiguity:

Ambiguity of the grammar that produces more than one parse tree for leftmost or rightmost derivation can be eliminated by re-writing the grammar.

Consider this example,

G: *stmt*→**if** *expr* **then** *stmt*|**if***expr***then***stmt***els***tmt***e**|**other** This grammar is ambiguous since the string **if E1 then if E2 then S1 else S2 has the** following two parse trees for leftmost derivation (Fig. 2.3)

To eliminate ambiguity, the following grammar may be used: *stmt*→*matched*|*unmatchedstmt_stmt*

matched→**if***exprstmt***then***matched***else***matchedtmt_stmt*|**other** *unmatched*→**if***exprs***then***mtst mt*|**if***expr***then***matched***else***unmatchedtmt_stmt*

Eliminating Left Recursion:

A grammar is said to be *left recursive* if it has an on-terminal *A* such that there is a derivation A=>Aα for some string α. Top-down parsing methods cannot handle left-recursive grammars. Hence, left recursion can be eliminated as follows:

If there is→ aαA |β production A can be replaced with as

A→βA'

A'→αA' | ε

without changing the set of strings derivable from A.

Left factoring:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. When it is not clear which of two alternative productions to use to expand a non-terminal A, we can rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

If there is any→αβ1|2αβ, production it can be A rewritten as

A→αA'

A'→β1|2 β

Consider the grammar ,

G : S → iEtS | iEtSeS | a

E→b

Left factored, this grammar becomes

S → iEtSS' | a

S' → eS |ε E → b

PARSING

It is the process of analyzing a continuous stream of input in order to determine its grammatical structure with respect to a given formal grammar.

Parse tree:

Graphical representation of a derivation or deduction is called a parse tree. Each interior node of the parse tree is a non-terminal; the children of the node can be terminals or non-terminals.

Types of parsing:

1. Top down parsing
2. Bottom up parsing

Ø Top-down parsing : A parser can start with the start symbol and try to transform it to the input string. Example : LL Parsers.

Ø Bottom-up parsing : A parser can start with input and attempt to rewrite it into the start symbol. Example : LR Parsers.

TOP-DOWN PARSING

It can be viewed as an attempt to find a left-most derivation for an input string or an attempt to construct a parse tree for the input starting from the root to the leaves.

Types – of top down parsing :

1. Recursive descent parsing
2. Predictive parsing

RECURSIVE DESCENT PARSING

Typically, top-down parsers are implemented as a set of recursive functions that descent through a parse tree for a string. This approach is known as recursive descent parsing, also known as LL(k) parsing where the first L stands for left-to-right, the second L stands for leftmost-derivation, and k indicates k-symbol lookahead.

Therefore, a parser using the single-symbol look-ahead method and top-down parsing without backtracking is called LL(1) parser. In the following sections, we will also use an extended BNF notation in which some regulation expression operators are to be incorporated.

This parsing method may involve backtracking.

Example for :Backtracking

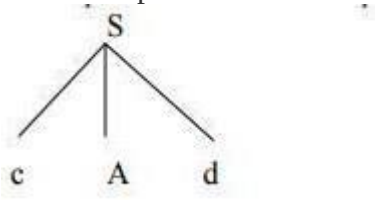
Consider the grammar G : S → cAd
A→ab|a

and the input string w=cad.

The parse tree can be constructed using the following top-down approach :

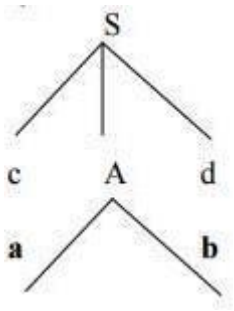
Step1:

Initially create a tree with single node labeled S. An input pointer points to 'c', the first symbol of w. Expand the tree with the production of S.



Step2:

The leftmost leaf 'c' matches the first symbol of w, so advance the input pointer to the second symbol of w 'a' and consider the next leaf 'A'. Expand A using the first alternative.

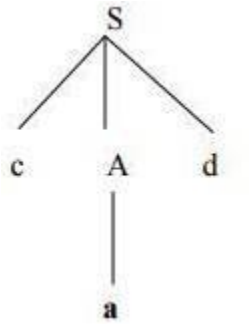


Step3:

The second symbol 'a' of w also matches with second leaf of tree. So advance the input pointer to third symbol of w 'd'**.**But the third leaf of tree is b which does not match with the input symbol **d.**Hence discard the chosen production and reset the pointer to second **backtracking.**

Step4:

Now try the second alternative for A.



Now we can halt and announce the successful completion of parsing.

Predictive parsing

It is possible to build a non recursive predictive parser by maintaining a stack explicitly, rather than implicitly via recursive calls. The key problem during predictive parsing is that of determining the production to be applied for a nonterminal . The non recursive parser in figure looks up the production to be applied in parsing table. In what follows, we shall see how the table can be constructed directly from certain grammars.

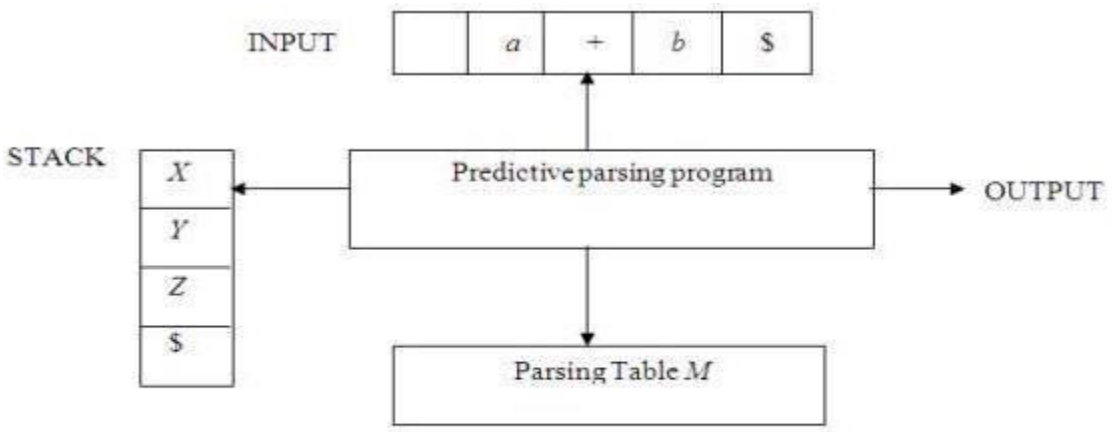


Fig. 2.4 Model of a nonrecursive predictive parser

A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream. The input buffer contains the string to be parsed, followed by $, a symbol used as a right end marker to indicate the end of the input string. The stack contains a sequence of grammar symbols with $ on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on top of $. The parsing table is a two dimensional array M[A,a] where A is a nonterminal, and a is a terminal or the symbol $. The parser is controlled by a program that behaves as follows. The program considers X, the symbol on the top of the stack, and a, the current input symbol. These two symbols determine the action of the parser. There are three possibilities.

1 If X= a=$, the parser halts and announces successful completion of parsing.

3 If X is a nonterminal, the program consults entry M[X,a] of the parsing table M. This entry will be either an X-production of the grammar or an error entry. If, for example, M[X,a]={X- >UVW}, the parser replaces X on top of the stack by WVU(with U on top). As output, we shall assume that the parser just prints the production used; any other code could be executed here. If M[X,a]=error, the parser calls an error recovery routine

Predictive parsing table construction:

The construction of a predictive parser is aided by two functions associated with a grammar G :

3. FIRST
4. FOLLOW

Rules for first( ):

1. If X is terminal, then FIRST(X) is {X}.
2. If X → ε is a production, then add ε to FIRST(X).
3. If X is non-terminal and X → aα is a production then add a to FIRST(X).
4. If X is non-terminal and X → Y1 Y2…Yk is a production, then place a in FIRST(X) if for some i, a is in FIRST(Yi), and ε is in all of FIRST(Y1),…,FIRST(Yi-1);that is, Y1,….Yi-1=> ε. If ε is in FIRST(Yj) for all j=1,2,..,k, then add ε to FIRST(X).

Rules for follow( ):

1. If S is a start symbol, then FOLLOW(S) contains $.
2. If there is a production A → αBβ, then everything in FIRST(β) except ε is placed in follow(B).
3. If there is a production A → αB, or a production A → αBβ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

Algorithm for construction of predictive parsing table:

Input : Grammar G

Output : Parsing table M

Method :

1. For each production A → α of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST(α), add A → α to M[A, a].
3. If ε is in FIRST(α), add A → α to M[A, b] for each terminal b in FOLLOW(A). If ε is in FIRST(α) and $ is in FOLLOW(A) , add A → α to M[A, $].
4. Make each undefined entry of M be error.

Example:

Consider the following grammar :

E→E+T|T

T→T*F|F

F→(E)|id

After eliminating left-recursion the grammar is

E →TE'

E' → +TE' | ε

T →FT'

T' → *FT' | ε

F → (E)|id

First( ) :

FIRST(E) = { (, id}

FIRST(E') ={+ , ε }

FIRST(T) = { (, id}

FIRST(T') = {*, ε }

FIRST(F) = { (, id }

Follow( ):

FOLLOW(E) = { $,) }

FOLLOW(E') = { $,) }

FOLLOW(T) = { +, $,) }

FOLLOW(T') = { +, $,) }

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

Predictive parsing Table

NON-TERMINAL	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Stack Implementation

stack	Input	Output
$E	id+id*id $	
$E'T	id+id*id $	E→TE'
$E'T'F	id+id*id $	T→FT'
$E'T'id	id+id*id $	F→id
$E'T'	+id*id $	
$E'	+id*id $	T'→ε
$E'T+	+id*id $	E'→+TE'
$E'T	id*id $	
$E'T'F	id*id $	T→FT'
$E'T'id	id*id $	F→id
$E'T'	*id $	
$E'T'F*	*id $	T'→*FT'
$E'T'F	id $	
$E'T'id	id $	F→id
$E'T'	$	
$E'	$	T'→ε
$	$	E'→ε

LL(1) grammar:
The parsing table entries are single entries. So each location has not more than one entry. This type of grammar is called LL(1) grammar.

Consider this following grammar:
S→iEtS | iEtSeS| a

E→b

After eliminating left factoring, we have
S→iEtSS'|a
S'→ eS | ε
E→b

To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals.
FIRST(S) = { i, a }
FIRST(S') = {e, ε }
FIRST(E) = { b}

FOLLOW(S) = { $,e }
FOLLOW(S') = { $,e }
FOLLOW(E) = {t}

Parsing table:

NON-TERMINAL	a	b	e	i	t	$
S	S→a			S→iEtSS'		
S'			S'→eS S'→ε			S'→ε
E		E→b				

Since there are more than one production, the grammar is not LL(1) grammar.

❖ BOTTOM-UP PARSING

Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing. A general type of bottom-up parser is a shift-reduce parser.

SHIFT-REDUCE PARSING

Shift-reduce parsing is a type of bottom-up parsing that attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top). The reductions trace out the right-most derivation in reverse.

Example:
Consider the grammar:
E→E+E
E→E*E
E→(E)
E→id

And the input string id1+id2*id3

The rightmost derivation is :
E→E+E
→ E+E*E
→ E+E*id3
→ E+id2*id3
→ id1+id2* id3

In the above derivation the underlined substrings are called handles.

Handle pruning:

A rightmost derivation in reverse can be obtained by "handle pruning". (i.e.) if w is a sentence or string of the grammar at hand, then w = γn, where γn is the nth right sentinel form of
some rightmost derivation.

Actions in shift-reduce parser:

- shift - The next input symbol is shifted onto the top of the stack.
- reduce - The parser replaces the handle within a stack with a non-terminal.
- accept - The parser announces successful completion of parsing.
- error - The parser discovers that a syntax error has occurred and calls an error recovery routine.

Conflicts in shift-reduce parsing:

There are two conflicts that occur in shift-reduce parsing:

1. Shift-reduce conflict: The parser cannot decide whether to shift or to reduce.
2. Reduce-reduce conflict: The parser cannot decide which of several reductions to make.

Stack implementation of shift-reduce parsing :

Stack	Input	Action
$	$id_1+id_2*id_3$ $	shift
$ id_1	$+id_2*id_3$ $	reduce by E→id
$E	$+id_2*id_3$ $	shift
$E+	id_2*id_3 $	shift
$E+$id_2$	$*id_3$ $	reduce by E→id
$E+E	$*id_3$ $	shift
$E+E*	id3 $	shift
$E+E*id3	$	reduce by E→id
$E+E*E	$	reduce by E→E *E
$E+E	$	reduce by E→E+E
$E	$	accept

1. Shift-reduce conflict:

Example:

Consider the grammar:

E→E+E | E*E | id and input id+id*id

Stack	Input	Action	Stack	Input	
$E+E	*id$	Reduce by E→E+E	$E+E	*id$	Shift
$E	*id$	Shift	$E+E*	id$	Shift
$E*	id$	Shift	$E+E*id	$	Reduce by E→id
$E*id	$	Reduce by E→id	$E+E*E	$	Reduce by E→E*E
$E*E	$	Reduce by E→E*E	$E+E	$	Reduce by E→E*E
$E			$E		

2. Reduce-reduce conflict:

Consider the grammar: M→R+R|R+c|R

R→c

and input c+c

Stack	Input	Action	Stack	Input	Action
$	c+c$	Shift	$	c+c$	Shift
$c	+c$	Reduce by R→c	$c	+c$	Reduce by R→c
$R	+c$	Shift	$R	+c$	Shift
$R+	c$	Shift	$R+	c$	Shift
$R+c	$	Reduce by R→c	$R+c	$	Reduce by M→R+c
$R+R	$	Reduce by M→R+R	$M	$	
$M	$				

Viable prefixes:

α is a viable prefix of the grammar if there is w such that αw is a right

The set of prefixes of right sentinel forms that can appear on the stack of a shift-reduce parser are called viable prefixes

The set of viable prefixes is a regular language.

LR PARSERS

An efficient bottom-up syntax analysis technique that can be used CFG is called LR(k) parsing. The 'L' is for left-to-right scanning of the input, the 'R' for constructing a rightmost derivation in reverse, and the 'k' for the number of input symbols. When 'k' is omitted, it is
assumed to be 1.

Advantages of LR parsing:

1. It recognizes virtually all programming language constructs for which CFG can be written.
2. It is an efficient non-backtracking shift-reduce parsing method.
3. A grammar that can be parsed using LR method is a proper superset of a grammar that can be parsed with predictive parser
4. 4. It detects a syntactic error as soon as possible.

Drawbacks of LR method:

It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

Types of LR parsing method:

1. SLR- Simple LR
   Easiest to implement, least powerful.
2. CLR- Canonical LR
   Most powerful, most expensive.
3. LALR- Look-Ahead LR
   Intermediate in size and cost between the other two methods.

The LR parsing algorithm:

The schematic form of an LR parser is as follows:

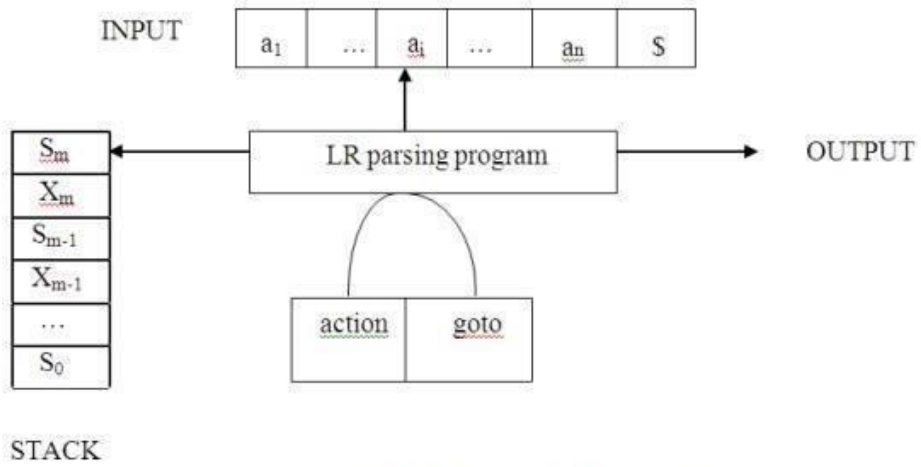


Fig. 2.5 Model of an LR parser

It consists of an input, an output, a stack, a driver program, and a pa parts (action and goto).

- Ø The driver program is the same for all LR parser.
- Ø The parsing program reads characters from an input buffer one at a time.
- Ø The program uses a stack to store a string of the form s0X1s1X2s2…Xmsm, where sm is on top. Each Xi is a grammar symbol and each si is a state.
- Ø The parsing table consists of two parts : action and goto functions.

Action : The parsing program determines sm, the state currently on top of stack, and ai, the current input symbol. It then consults action[sm,ai] in the action table which can have one of four values:

1. shift s, where s is a state,
2. reduce by a grammar production A → β,
3. accept,
4. 4. error.

Goto : The function goto takes a state and grammar symbol as arguments and produces a state.

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 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.

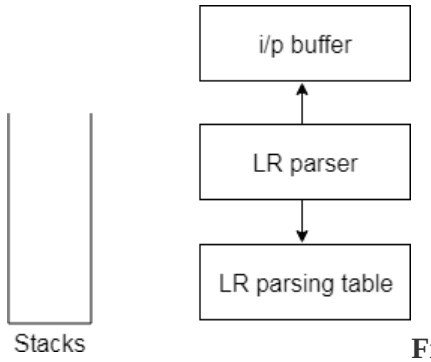


Fig: Block diagram of LR parser

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

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:

SLR (1) Grammar

E→E+T|T
T→T*F|F
F → (E) | id

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

E`→•E
E→•E+T
E→•T
T→•T*F
T→•F
F→•(E)
F → •id

Add Augment production to the I0 State and Compute the Closure

I0:E`→•E
E→•E+T
E→•T

T→•T*F

T→•F

F→•(E)

F → •id

Go to (I0, E) = closure (E` → E•, E → E• + T) **=I1**

Go to (I0, T) = closure (E → T•, T→ T• * F) **=I2**

Go to (I0, F) = Closure (T → F•) = T → F**•= I3**

Go to (I0, ( ) = Closure (F→(•E),E→•E+T, E→•T, T→•T*F, T→•F, F→•(E), F → •id) **= I4**

Go to (I0, id) = closure (F → id•) = F → id• **= I5**

Go to (I1, +) = Closure (E → E +•T, T→•T*F, T→•F, T→•(E), F → •id) **= I6**

Go to (I2, *) = Closure(T→ T *• F, F→•(E), F → •id) **= I7**

Go to (I4, E) = Closure(F→(E•), E→E•+T) **= I8**

Go to (I4, T) = Closure(E → T•, T→ T• * F) **=I2**

Go to (I4, F) = Closure(T → F•) = T → F**•= I3**

Go to (I4, ( ) = Closure(F→(•E),E→•E+T, E→•T, T→•T*F, T→•F, F→•(E), F → •id) **= I4**

Go to (I4, id) = Closure(F → id•) = F → id• **= I5**

Go to (I6, T) = Closure(E→E+T•, T→T•*F) **= I9**

Go to (I6, F) = Closure(T → F•) = T → F**•= I3**

Go to (I6, ( ) = Closure(F→(•E),E→•E+T, E→•T, T→•T*F, T→•F, F→•(E), F → •id) **= I4**

Go to (I6, id) = Closure(F → id•) = F → id• **= I5**

Go to (I7, F) = Closure(T→T*F•)**= I10**

Go to (I7, ( ) = Closure(F→(•E),E→•E+T, E→•T, T→•T*F, T→•F, F→•(E), F → •id) **= I4**

Go to (I7, id) = Closure(F → id•) = F → id• **= I5**

Go to (I8,)) = Closure(F→(E) **•= I11**

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

Go to (I9, *) = Closure(T→ T *• F, F→•(E), F → •id) **= I7**

SLR (1) Table

States	Action						Goto		
	+	*	(	)	id	$	E	T	F
I0			S4		S5		1	2	3
I1	S6					Accept			
I2	R2	S7		R2		R2			
I3	R4	R4		R4		R4			
I4			S4		S5		8	2	3
I5	R6	R6		R6		R6			

I6			S4		S5			9	3
I7			S4		S5				10
I8	S6			S11					
I9	R1	S7		R1		R1			
I10	R3	R3		R3		R3			
I11	R5	R5		R5		R5			

Explanation:

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

- I1 contains the final item which drives E` → E• and follow (E`) = {$}, so action {I1, $} = Accept
- I2 contains the final item which drives E → T• and follow (E) = {+,), $}, so action {I2, +} = R2, 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, action {I3,)} = R4
- I5 contains the final item which drives F → id• and follow (F) = {+, *, $,)}, so action {I5, +} = R6, action {I5, *} = R6, action {I5, $} = R6, action {I5,)} = R6
- I9 contains the final item which drives E → E + T• and follow (E) = {+,), $}, so action {I9, +} = R1, action {I9, $} = R1, action {I9,)} = R1
- I10 contains the final item which drives T → T * F• and follow (T) = {+, *,), $}, so action {I10, +} = R3, action {I10, *} = R3, action {I10, $} = R3,action {I10,)} = R3
- I11 contains the final item which drives F → (E)• and follow (T) = {+, *,), $}, so action {I11, +} = R5, action {I11, *} = R5, action {I11, $} = R5,action {I11,)} = R5.

Parsing:

Stack	Input	Action
0	Id*id+id$	
0 id 5	*id+id$	(0,id)=S5 shift id 5
0 F 3	*id+id$	(5,*) = r6 reduce F->id and (0,F)=3
0 T 2	*id+id$	(3,*)=r4 T->F and (0,T) = 2

0 T 2 * 7	id+id$	(2,*)=S7 shift *, 7
0 T 2 * 7 id 5	+id$	(7,id)=S5 shift id, 5
0 T 2 * 7 F 10	+id$	(5,+)=r6 F->id (7,F)=10
0 T 2	+id$	(10,+)=r3 T->T*F and (0,T)=2
0 E 1	+id$	(2,+)=r2 E->T and (0,E)=1
0 E 1 + 6	Id$	(1,+) = S6 shift +and 6
0 E 1 + 6 id 5	$	(6,id)=S5 shift 5 and id
0 E 1 + 6 F 3	$	(5,$)=r6 F->id and (6,F)=3
0 E 1 + 6 T 9	$	(3,$)=r4 T->F and (6,T)=9
0 E 1	$	(9,$)=r1 E->E+T and (0,E)=1
0 E 1	$	(1,$)=Accept

CLR (1) Grammar

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

S → AA

A → aA

A → b

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

S` → •S, $

S → •AA, $

A → •aA, a/b

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= Goto(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

Goto(I3,a)=Closure(A→a•A,a/b)=(sameasI3)
Go to (I3, b) = Closure (A → b•, a/b) = (same as I4)

I4= Goto(I0,b)=closure(A→b•,a/b)=A→b•,a/b
I5= Goto(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, $

Goto(I6,a)=Closure(A→a•A,$)=(sameasI6)
Go to (I6, b) = Closure (A → b•, $) = (same as I7)

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

CLR (1) Parsing table:

States	Action			Goto	
	a	b	$	S	A
0	S3	S4		1	2
1			ACCEPT		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

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

S → AA
A → aA
A → b

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

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

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/$}

LALR (1) Parsing table:

States	Action			Goto	
	a	b	$	S	A
0	S36	S47		1	2
1			ACCEPT		
2	S36	S47			5
36	S36	S47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Difference between LL and LR parser:

LL Parser	LR Parser
First L of LL is for left to right and second L is for leftmost derivation.	L of LR is for left to right and R is for rightmost derivation.
It follows the left most derivation.	It follows reverse of right most derivation.
Using LL parser parser tree is constructed in top down manner.	Parser tree is constructed in bottom up manner.
In LL parser, non-terminals are expanded.	In LR parser, terminals are compressed.
Starts with the start symbol(S).	Ends with start symbol(S).
Ends when stack used becomes empty.	Starts with an empty stack.
Pre-order traversal of the parse tree.	Post-order traversal of the parser tree.
Terminal is read after popping out of stack.	Terminal is read before pushing into the stack.
It may use backtracking or dynamic programming.	It uses dynamic programming.
LL is easier to write.	LR is difficult to write.
Example: LL(0), LL(1)	**Example:** LR(0), SLR(1), LALR(1), CLR(1)

UNIT IV

❖ Syntax Directed Definition

Syntax Directed Definition (SDD) is a kind of abstract specification. It is generalization of context free grammar in which each grammar production **X –> a** is associated with it a set of production rules of the form s = f(b_1, b_2, ……b_k) where s is the attribute obtained from function f. The attribute can be a string, number, type or a memory location. Semantic rules are fragments of code which are embedded usually at the end of production and enclosed in curly braces ({ }).

Example:

E --> E_1 + T { E.val = E_1.val + T.val}

Annotated Parse Tree – The parse tree containing the values of attributes at each node for given input string is called annotated or decorated parse tree.

Features –

- High level specification
- Hides implementation details
- Explicit order of evaluation is not specified

Types of attributes – There are two types of attributes:

1. Synthesized Attributes – These are those attributes which derive their values from their children nodes i.e. value of synthesized attribute at node is computed from the values of attributes at children nodes in parse tree.

Example:

E --> E_1 + T { E.val = E_1.val + T.val}

In this, E.val derive its values from E_1.val and T.val

Computation of Synthesized Attributes –

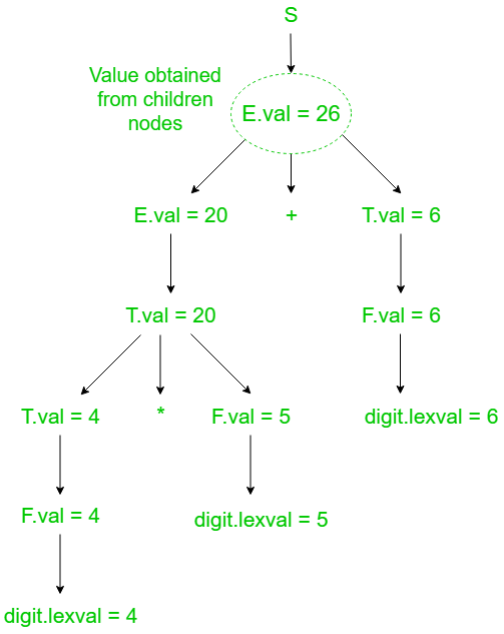
- Write the SDD using appropriate semantic rules for each production in given grammar.
- The annotated parse tree is generated and attribute values are computed in bottom up manner.
- The value obtained at root node is the final output.

Example: Consider the following grammar

- S --> E
- E --> E_1 + T
- E --> T
- T --> T_1 * F
- T --> F
- F --> digit
- The SDD for the above grammar can be written as follow

Production	Semantic Actions
S --> E	Print(E.val)
E --> E_1 + T	E.val = E_1.val + T.val
E --> T	E.val = T.val
T --> T_1 * F	T.val = T_1.val * F.val
T --> F	T.val = F.val
F --> digit	F.val = digit.lexval

Let us assume an input string **4 * 5 + 6** for computing synthesized attributes. The annotated parse tree for the input string is



Annotated Parse Tree

For computation of attributes we start from leftmost bottom node.

- The rule F –> digit is used to reduce digit to F and the value of digit is obtained from lexical analyzer which becomes value of F i.e. from semantic action F.val = digit.lexval.
- Hence, F.val = 4 and since T is parent node of F so, we get T.val = 4 from semantic action T.val = F.val.
- Then, for T –> T_1 * F production, the corresponding semantic action is T.val = T_1.val * F.val . Hence, T.val = 4 * 5 = 20
- Similarly, combination of E_1.val + T.val becomes E.val i.e. E.val = E_1.val + T.val = 26. Then, the production S –> E is applied to reduce E.val = 26 and semantic action associated with it prints the result E.val . Hence, the output will be 26.

2. Inherited Attributes – These are the attributes which derive their values from their parent or sibling nodes i.e. value of inherited attributes are computed by value of parent or siblingnodes.

Example:

L --> T { L.in = T.type }

Computation of Inherited Attributes –

- Construct the SDD using semantic actions.
- The annotated parse tree is generated and attribute values are computed in top down manner.

Example: Consider the following grammar

D--> T L

T --> int

T --> float

T --> double

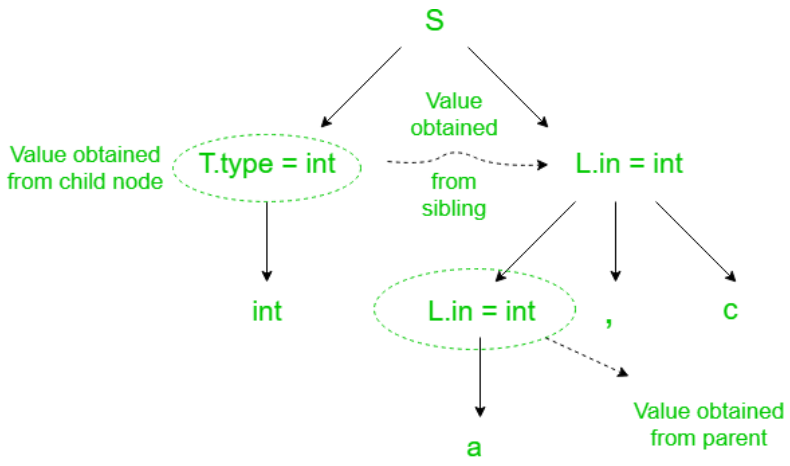
L --> L_1, id

L --> id

The SDD for the above grammar can be written as follow

Production	Semantic Actions
S --> T L	L.in = T.type
T --> int	T.type = int
T --> float	T.type = float
T --> double	T.type = double
L --> L_1 , id	L_1.in = L.in Enter_type(id.entry , L.in)
L --> id	Entry_type(id.entry , L.in)

Let us assume an input string **int a, c** for computing inherited attributes. The annotated parse tree for the input string is



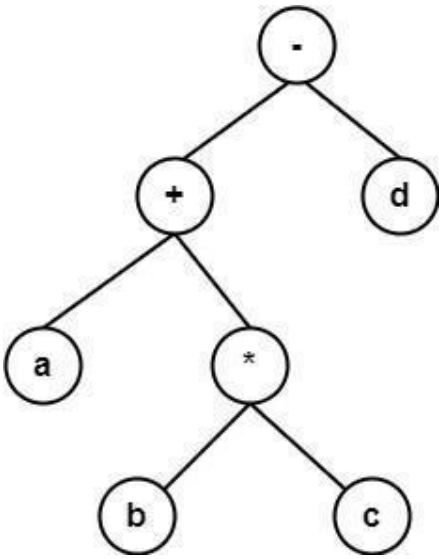
Annotated Parse Tree

The value of L nodes is obtained from T.type (sibling) which is basically lexical value obtained as int, float or double. Then L node gives type of identifiers a and c. The computation of type is done in top down manner or preorder traversal. Using function Enter_type the type of identifiers a and c is inserted in symbol table at corresponding id.entry.

❖ Syntax Tree

Tree in which each leaf node describes an operand & each interior node an operator. The syntax tree is shortened form of the Parse Tree.

Example1 − Draw Syntax Tree for the string a + b ∗ c − d.



Rules for constructing a syntax tree

Each node in a syntax tree can be executed as data with multiple fields.

In the node for an operator, one field recognizes the operator and the remaining field includes a pointer to the nodes for the operands.

The operator is known as the label of the node. The following functions are used to create the nodes of the syntax tree for the expressions with binary operators.

Each function returns a pointer to the recently generated node.

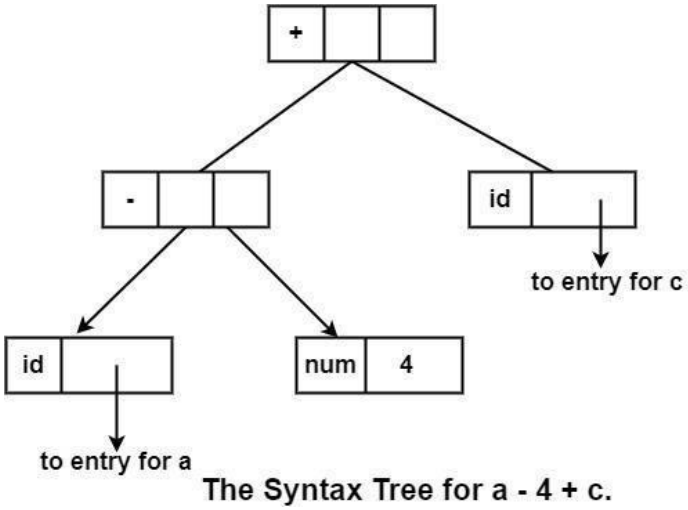
- **mknode (op, left, right)** − It generates an operator node with label op and two field including pointers to left and right.
- **mkleaf (id, entry)** − It generates an identifier node with label id and the field including the entry, a pointer to the symbol table entry for the identifier.
- **mkleaf (num, val)** − It generates a number node with label num and a field including val, the value of the number.

For example, construct a syntax tree for an expression a − 4 + c. In this sequence, p_1, p_2, … . . p_5 are pointers to the symbol table entries for identifier 'a' and 'c' respectively.

- p_1 – mkleaf (id, entry a);
- p_2 – mkleaf (num, 4);
- p_3 – mknode ('–', p_1, p_2)
- p_4 – mkleaf(id, entry c)
- p_5 – mknode('+', p_3, p_4);

The tree is generated in a bottom-up fashion.

- The function calls mkleaf (id, entry a) and mkleaf (num 4) construct the leaves for a and 4.
- The pointers to these nodes are stored using $p_1$and p_2. The call mknodes ('−', p_1, p_2) then make the interior node with the leaves for a and 4 as children. The syntax tree will be

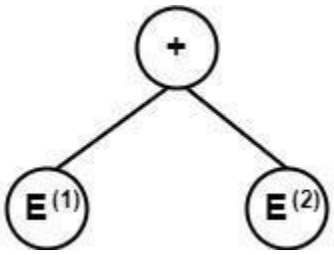


The Syntax Tree for a - 4 + c.

Syntax Directed Translation of Syntax Trees

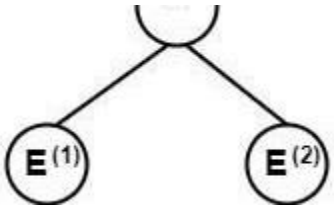
Production	**Semantic Action**
$E \rightarrow E^{(1)} + E^{(2)}$	{E. VAL = Node (+, $E^{(1)}$. VAL, $E^{(2)}$. VAL)}
$E \rightarrow E^{(1)} * E^{(2)}$	{E. VAL = Node (∗, $E^{(1)}$. VAL, $E^{(2)}$. VAL)})
$E \rightarrow (E^{(1)})$	{E. VAL = $E^{(1)}$. VAL}
$E \rightarrow$- $E^{(1)}$	{E. VAL = UNARY (−, $E^{(1)}$. VAL}
E → id	{E. VAL = Leaf (id)}

Node (+, $^{(1)}$, **VAL**, $^{(2)}$. **VAL**) will create a node labeled +.

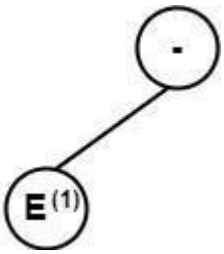


$E^{(1)}$. VAL &$E^{(2)}$. VAL **are left & right children of this node.**

Similarly, Node (∗, $E^{(1)}$. VAL, $E^{(2)}$. VAL) will make the syntax as −

- 

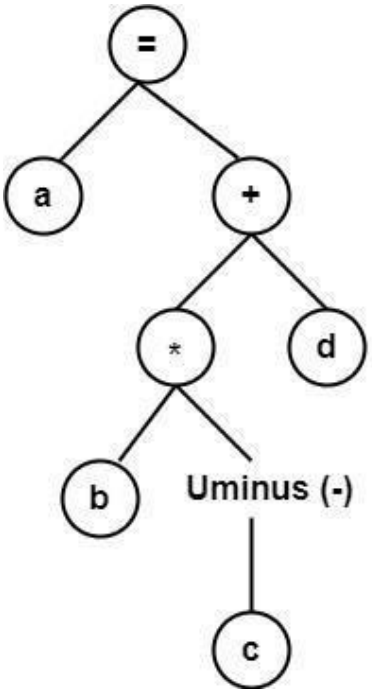
Function **UNARY** (−, $E^{(1)}$. VAL)will make a node – (unary minus) & $E^{(1)}$. VAL will be the only child of it.



Function **LEAF (id)** will create a Leaf node with label id.

Example2 − Construct a syntax tree for the expression.

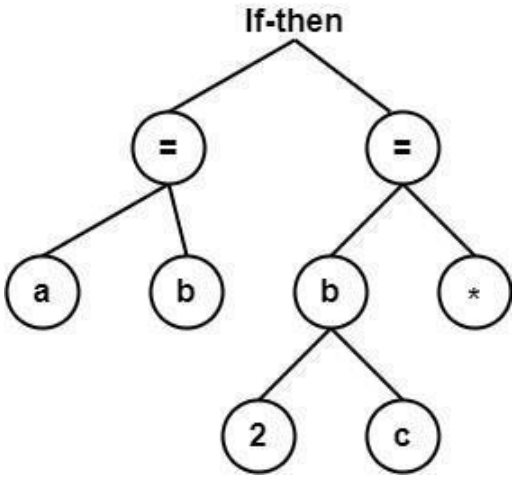
- a = b ∗ −c + d
- **Solution**



Example3 − Construct a syntax tree for a statement.

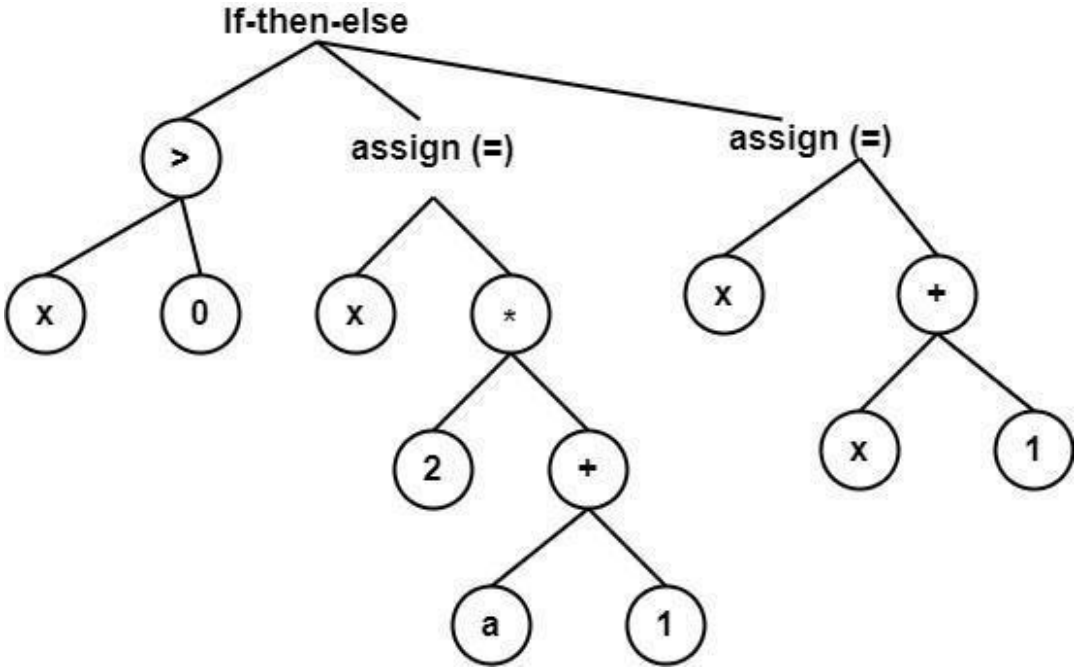
If a = b then b = 2 * c

Solution

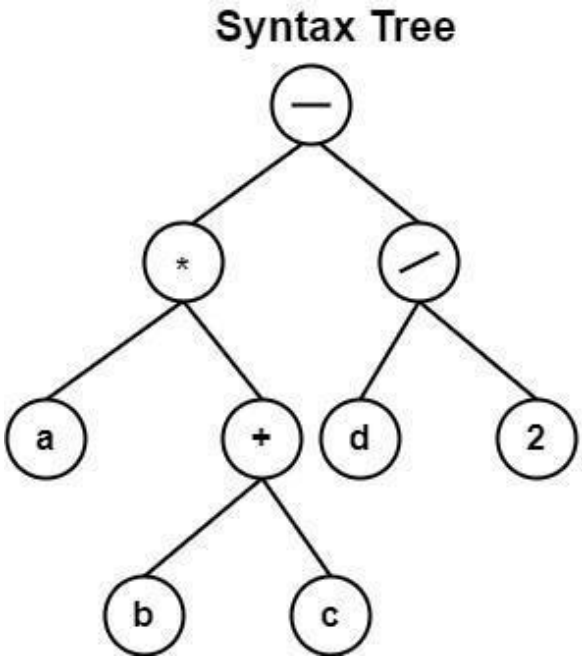


Example4 − Consider the following code. Draw its syntax Tree

If x > 0 then x = 2 * (a + 1) else x = x + 1.



Example5 − Draw syntax tree for following arithmetic expression a * (b + c) – d /2. Also, write given expression in postfix form.



❖ Intermediate code Generation

Intermediate code is used to translate the source code into the machine code. Intermediate code lies between the high-level language and the machine language.

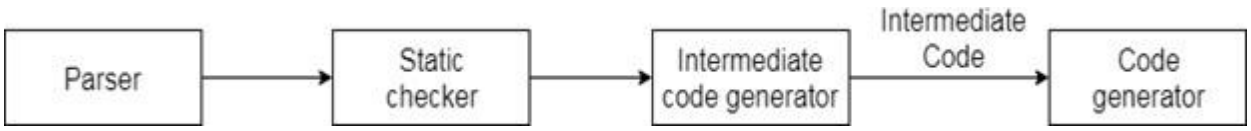


Fig: Position of intermediate code generator

- If the compiler directly translates source code into the machine code without generating intermediate code then a full native compiler is required for each new machine.
- The intermediate code keeps the analysis portions same for all the compilers that's why it doesn't need a full compiler for every unique machine.
- Intermediate code generator receives input from its predecessor phase and semantic analyzer phase. It takes input in the form of an annotated syntax tree.
- Using the intermediate code, the second phase of the compiler synthesis phase is changed according to the target machine.

Intermediate representation

Intermediate code can be represented in two ways:

1. High Level intermediate code:

High level intermediate code can be represented as source code. To enhance performance of source code, we can easily apply code modification. But to optimize the target machine, it is less preferred.

2. Low Level intermediate code

Low level intermediate code is close to the target machine, which makes it suitable for register and memory allocation etc. it is used for machine-dependent optimizations.

The following are commonly used intermediate code representations:

1. **Postfix Notation:** Also known as reverse Polish notation or suffix notation. The ordinary (infix) way of writing the sum of a and b is with an operator in the middle: a + b The postfix notation for the same expression places the operator at the right end as ab +. In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by e1e2 +. No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression. In postfix notation, the operator follows the operand.

   Example 1: The postfix representation of the expression (a + b) * c is : **ab + c ***

   Example 2: The postfix representation of the expression (a – b) * (c + d) + (a – b) is:

   ab–cd+*ab-+

2. **Three-Address Code:** A statement involving no more than three references (two for operands and one for result) is known as a three address statement. A sequence of three address statements is known as a three address code. Three address statement is of form x = y op z, where x, y, and z will have address (memory location). Sometimes a statement might contain less than three references but it is still called a three address statement.

   Example: The three address code for the expression

   a + b * c + d :

   T 1 = b * c

   T 2 = a + T 1

   T 3 = T 2 + d T 1 , T 2 , T 3 are temporary variables.

   There are 3 ways to represent a Three-Address Code in compiler design:
   i)Quadruples

   ii)Triples

   iii)Indirect Triples

3. **Syntax Tree:** A syntax tree is nothing more than a condensed form of a parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by the single link in the syntax tree the internal nodes are operators and child nodes are operands. To form a syntax tree put parentheses in the expression, this way it’s easy to recognize which operand should come first.

❖ Abstract Syntax Tree

Syntax trees are called as **Abstract Syntax Trees** because-

- They are abstract representation of the parse trees.
- They do not provide every characteristic information from the real syntax.
- For example- no rule nodes, no parenthesis etc.

Example 1

Considering the following grammar-

$E \rightarrow E + T \mid T$

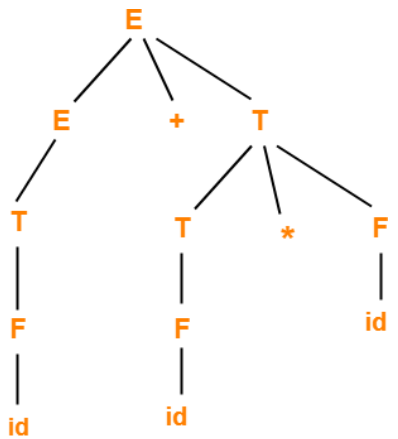
$T \rightarrow T \times F \mid F$

$F \rightarrow (E) \mid id$

Generate the following for the string id + id x id

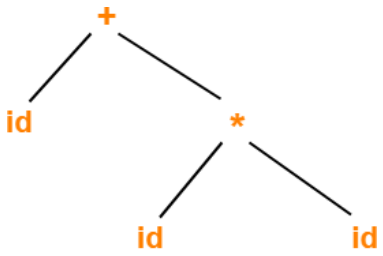
1. Parse tree
2. Syntax tree
3. Directed Acyclic Graph (DAG)

Parse Tree-



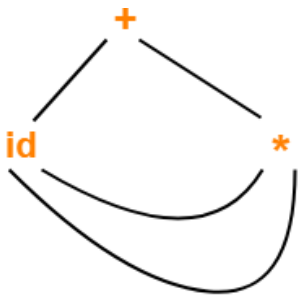
Parse Tree

Syntax Tree-



Syntax Tree

Directed Acyclic Graph-



Directed Acyclic Graph

Example 2

Construct a syntax tree for the following arithmetic expression-

(a + b) * (c – d) + ((e / f) * (a + b))

Solution-

Step-01:

We convert the given arithmetic expression into a postfix expression as-

(a + b) * (c – d) + ((e / f) * (a + b))

ab+ * (c – d) + ((e / f) * (a + b))

ab+ * cd- + ((e / f) * (a + b))

ab+ * cd- + (ef/ * (a + b))

ab+ * cd- + (ef/ * ab+)

ab+ * cd- + ef/ab+*

ab+cd-* + ef/ab+*

ab+cd-*ef/ab+*+

Step-02:

We draw a syntax tree for the above postfix expression.

Steps Involved

Start pushing the symbols of the postfix expression into the stack one by one.

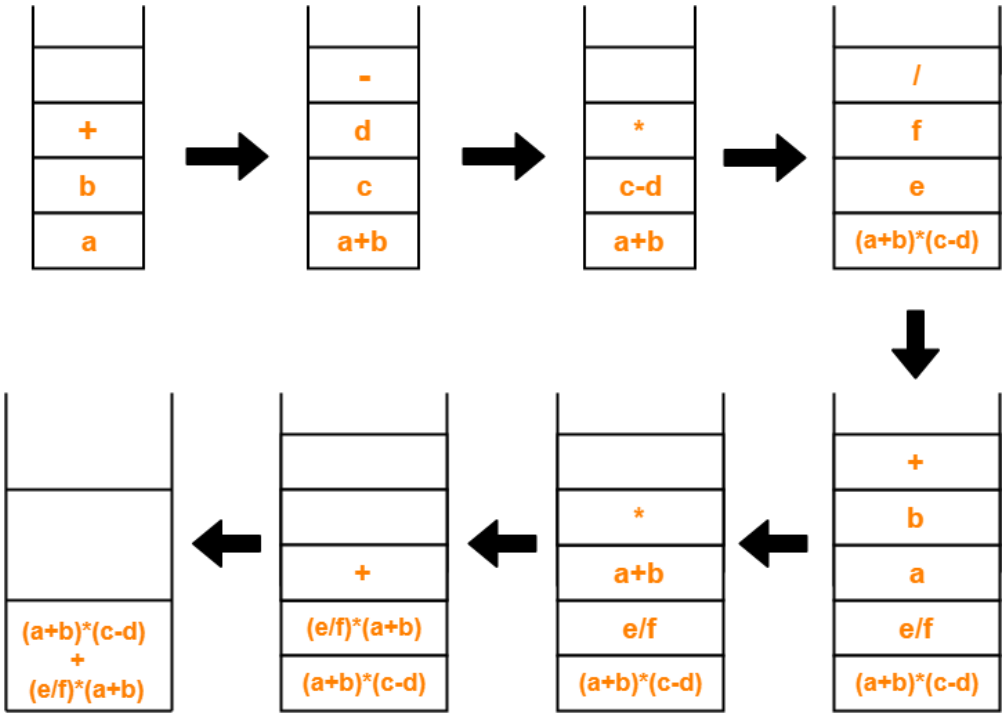
When an operand is encountered,

- Push it into the stack.

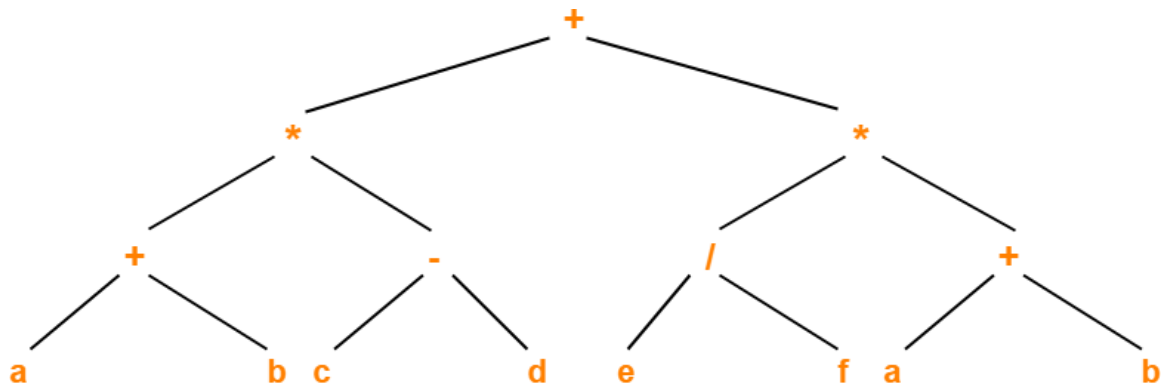
When an operator is encountered

- Push it into the stack.
- Pop the operator and the two symbols below it from the stack.
- Perform the operation on the two operands using the operator you have in hand.
- Push the result back into the stack.

Continue in the similar manner and draw the syntax tree simultaneously.



The required syntax tree is-



Syntax Tree

❖ Three address code:

- Three-address code is an intermediate code. It is used by the optimizing compilers.
- In three-address code, the given expression is broken down into several separate instructions. These instructions can easily translate into assembly language.
- Each Three address code instruction has at most three operands. It is a combination of assignment and a binary operator.

Example

Given Expression:

a := (-c * b) + (-c * d)

Three-address code is as follows:

t_1 := -c
t_2 := b*t_1
t_3 := -c
t_4 := d * t_3
t_5 := t_2 + t_4
a := t_5

t is used as registers in the target program.

The three address code can be represented in two forms: **quadruples** and **triples**.

Quadruples

The quadruples have four fields to implement the three address code. The field of quadruples contains the name of the operator, the first source operand, the second source operand and the result respectively.

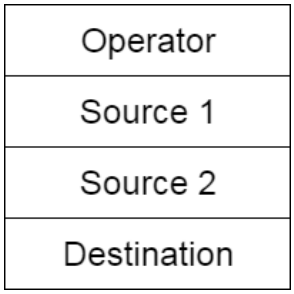


Fig: Quadruples field

Example

a := -b * c + d

Three-address code is as follows:

t_1 := -b

t_2 := c + d

t_3 := t_1 * t_2

a := t_3

These statements are represented by quadruples as follows:

	Operator	**Argument 1**	**Argument 2**	**Result**
(0)	Uminus	b	-	t1
(1)	+	c	d	t2
(2)	*	t1	t2	t3
(3)	=	t3	-	a

Triples:

The triples have three fields to implement the three address code. The field of triples contains the name of the operator, the first source operand and the second source operand.

In triples, the results of respective sub-expressions are denoted by the position of expression. Triple is equivalent to DAG while representing expressions.

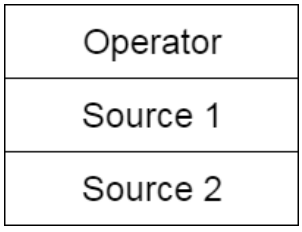


Fig: Triples field

Example

a := -b * c + d

Three-address code is as follows:

t_1 := -b

t_2 := c + d

t_3 := t_1 * t_2

a := t_3

These statements are represented by triples follows:

	Operator	**Argument 1**	**Argument 2**
(0)	Uminus	b	-
(1)	+	c	d
(2)	*	(0)	(1)
(3)	=	a	(2)

Indirect Triples:

It uses Indirect Addressing

a := -b * c + d

Three-address code is as follows:

t_1 := -b

t_2 := c + d

t_3 := t_1 * t_2

a := t_3

These statements are represented by Indirect triples follows:

	Address
(50)	(0)
(51)	(1)
(52)	(2)
(53)	(3)

	Operator	**Argument 1**	**Argument 2**
(50)	Uminus	b	-
(51)	+	c	d
(52)	*	(50)	(51)
(53)	=	a	(52)

❖ SDT to Three address code:

1. Translation of Assignment Statements

In the syntax directed translation, assignment statement is mainly deals with expressions. The expression can be of type real, integer, array and records.

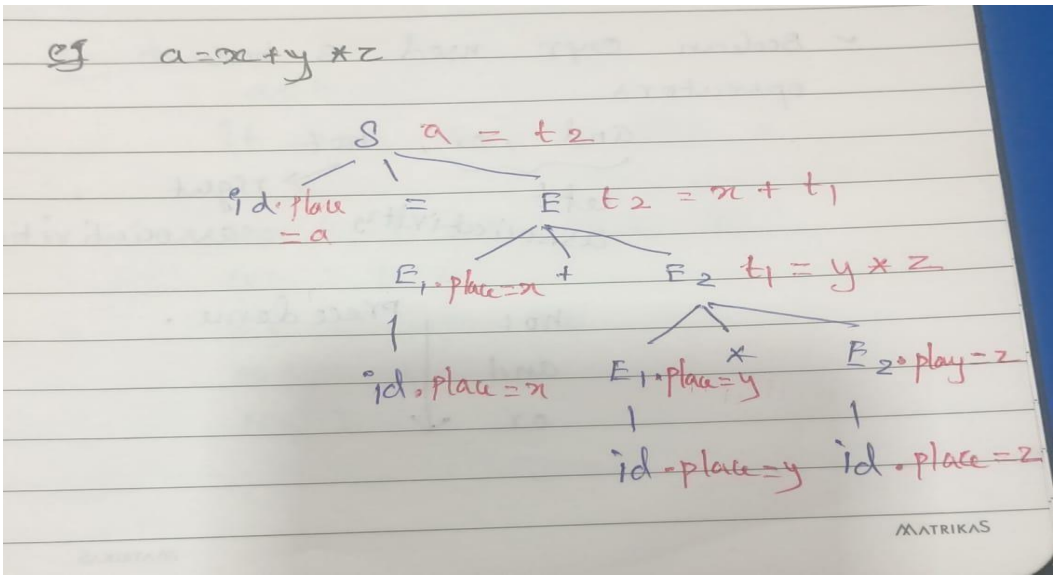
Consider the grammar

1. S → id := E
2. E → E1 + E2
3. E → E1 * E2
4. E → (E1)
5. E → id

The translation scheme of above grammar is given below:

Production rule	Semantic actions
S → id :=E	{gen (id.place=E.place);}
E → E1 + E2	{E.place=newtemp(); Emit (E.place = E1.place '+' E2.place) }
E → E1 * E2	{E.place=newtemp(); Emit (E.place = E1.place '*' E2.place) }
E → (E1)	{E.place = E1.place}
E → id	{E.place = id.place}

- The Emit function is used for appending the three address code to the output file. Otherwise it will report an error.
- The newtemp() is a function used to generate new temporary variables.
- E.place holds the value of E.



Three address code

t1=y*z

t2=x+t1

a=t2

2. Translation of Boolean Expression Numerical Representation

Boolean expressions have two primary purposes. They are used for computing the logical values. They are also used as conditional expression using if-then-else or while-do.

Consider the grammar

1. E → E OR E
2. E → E AND E
3. E → NOT E
4. E → (E)
5. E → id relop id
6. E → TRUE
7. E → FALSE

The relop is denoted by <, >, <, >.

The AND and OR are left associated.

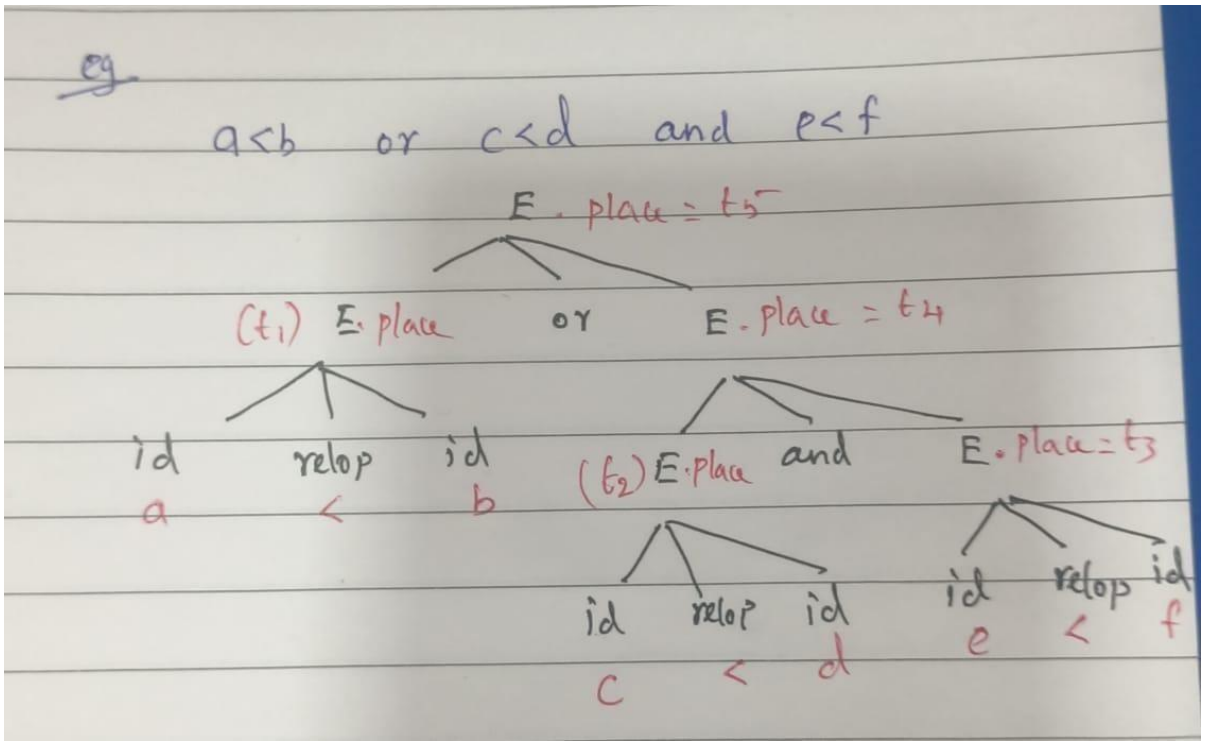
NOT has the higher precedence then AND and lastly OR.

Production rule	Semantic actions
E → E1 OR E2	{E.place=newtemp(); Emit (E.place ':=' E1.place 'OR' E2.place) }
E → E1 AND E2	{E.place=newtemp(); Emit (E.place ':=' E1.place 'AND' E2.place) }
E → NOT E1	{E.place=newtemp(); Emit (E.place ':=' 'NOT' E1.place) }
E → (E1)	{E.place = E1.place}
E → id relop id2	{E.place=newtemp(); Emit ('if' id1.place relop.op id2.place 'goto' nextstate+3); EMIT(E.place':=''0') EMIT('goto'nextstate+2) EMIT(E.place':=''1')
E → TRUE	{E.place:=newtemp(); Emit(E.place':=''1') }
E → FALSE	{E.place:=newtemp(); Emit(E.place':=''0') }

The EMIT function is used to generate the three address code and the newtemp( ) function is used to generate the temporary variables.

The E → id relop id2 contains the next_state and it gives the index of next three address statements in the output sequence.

Here is the example which generates the three address code using the above translation scheme:



Three address code

a<b OR c<d AND e<f

1. 100: **if** a<b **goto** 103
2. 101: t1:=0
3. 102: **goto** 104
4. 103: t1:=1
5. 104: **if** c<d **goto** 107
6. 105: t2:=0
7. 106: **goto** 108
8. 107: t2:=1
9. 108: **if** e<f **goto** 111
10. 109: t3:=0
11. 110: **goto** 112
12. 111: t3:= 1
13. 112: t4:= t2 AND t3
14. 113: t5:= t1 OR t4

3. Translation of Boolean Expression control flow statements

Control statements are the statements that change the flow of execution of statements.

Consider the Grammar

S → if E then S1

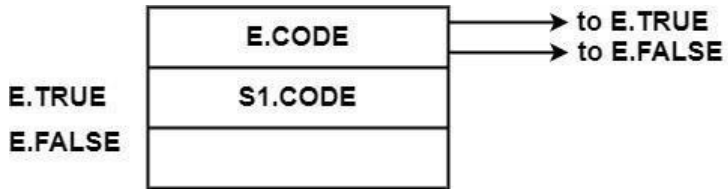
|if E then S1 else S2

|while E do S1

In this grammar, E is the Boolean expression depending upon which S1 or S2 will be executed.

Following representation shows the order of execution of an instruction of if-then, ifthen-else, & while do.

- **S → if E then S1**



E.CODE & S.CODE are a sequence of statements which generate three address code.

E.TRUE is the label to which control flow if E is true.

E.FALSE is the label to which control flow if E is false.

The code for E generates a jump to E.TRUE if E is true and a jump to S.NEXT if E is false.

∴ E.FALSE=S.NEXT in the following table.

In the following table, a new label is allocated to E.TRUE.

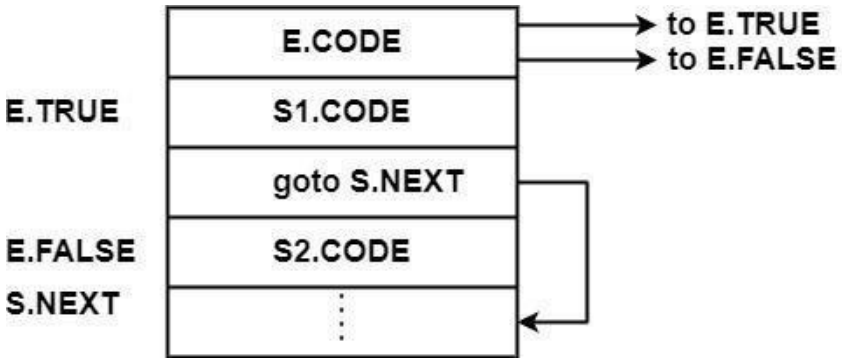
When S1.CODE will be executed, and the control will be jumped to statement following S, i.e., to S1.NEXT.

∴ S1. NEXT = S. NEXT.

Syntax Directed Translation for "If E then S1."

Production	Semantic Rule
S → if E then S1	E. TRUE = newlabel; E. FALSE = S. NEXT; S1. NEXT = S. NEXT; S. CODE = E. CODE | |GEN (E. TRUE '– ') | | S1. CODE

- **S → If E then S1 else S2**



If E is true, control will go to E.TRUE, i.e., S1.CODE will be executed and after that S.NEXT appears after S1.CODE.

If E.CODE will be false, then S2.CODE will be executed.

Initially, both E.TRUE & E.FALSE are taken as new labels. Hen S1.CODE at label E.TRUE is executed, control will jump to S.NEXT.

Therefore, after S1, control will jump to the next statement of complete statement S.

S1.NEXT=S.NEXT

Similarly, after S2.CODE, the next statement of S will be executed.

∴ S2.NEXT=S.NEXT

Syntax Directed Translation for "If E then S1 else S2."

Production	Semantic Rule
S → if E then S1 else S2	E. TRUE = newlabel; E. FALSE = newlabel; S1. NEXT = S. NEXT; S2. NEXT = S. NEXT; S. CODE = E. CODE | | GEN (E. TRUE '− ') | | S1. CODE GEN(goto S. NEXT) | | GEN (E. FALSE −) | | S2. CODE

- **S → while E do S1**

Another important control statement is while E do S1, i.e., statement S1 will be executed till Expression E is true. Control will arrive out of the loop as the expression E will become false

A Label S. BEGIN is created which points to the first instruction for E. Label E. TRUE is attached with the first instruction for S1. If E is true, control will jump to the label E. TRUE & S1. CODE will be executed. If E is false, control will jump to E. FALSE. After S1. CODE, again control will jump to S. BEGIN, which will again check E. CODE for true or false.

∴ S1. NEXT = S. BEGIN

If E. CODE is false, control will jump to E. FALSE, which causes the next statement after S to be executed.

∴ E. FALSE = S. NEXT

Syntax Directed Translation for " → while E do S1 "

Production	Semantic Rule
S → while E do S1	S. BEGIN = newlabel; E. TRUE = newlabel; E. FALSE = S. NEXT; S1. NEXT = S. BEGIN; S. CODE = GEN(S. BEGIN '– ') | | E. CODE | | GEN(E. TRUE '– ')| | S1. CODE | | GEN('goto' S. BEGIN)

Example:

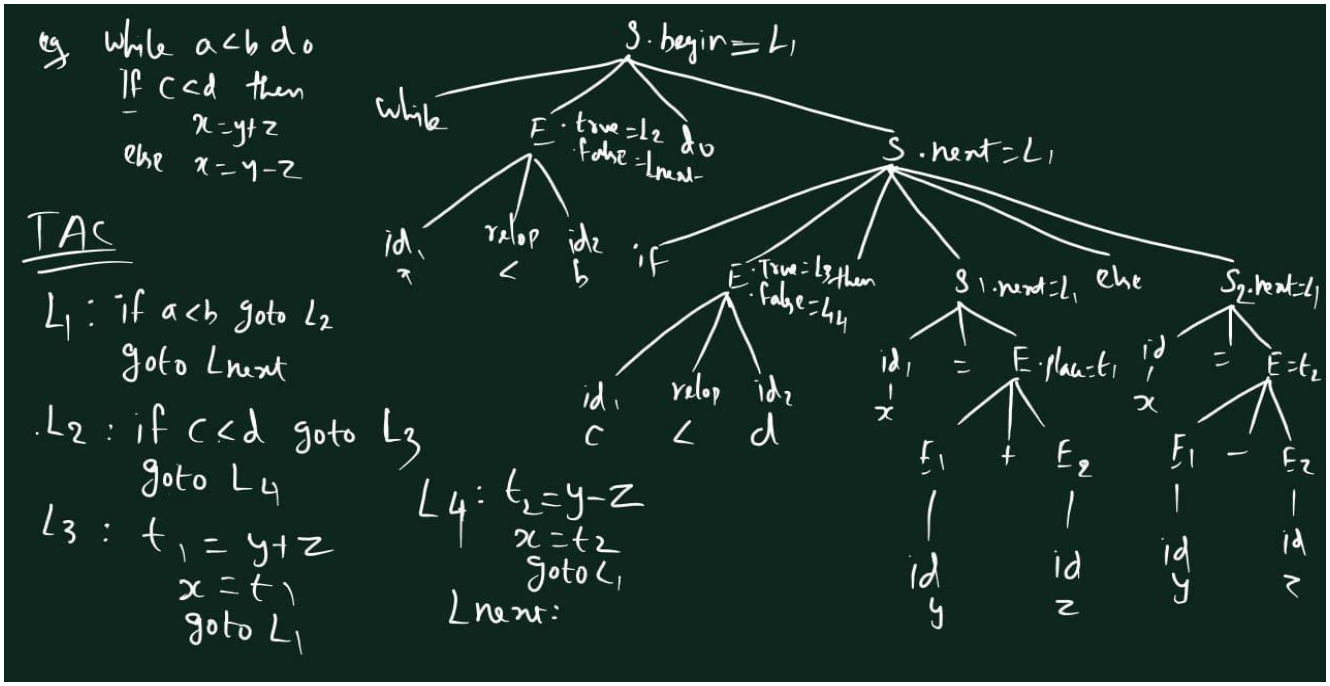
While a < b do

If c < d then

x : = y + z

Else

x := y - z



Then the translation is

L1: `if a < b goto L2`

`goto Lnext`

L2: `if c < d goto L3`

`goto L4`

L3: `t1 := y + z`

`x := t1`

`goto L1`

L4: `t2 := y - z`

`x := t2`

`goto L1`

Lnext:

UNIT V

❖ Basic Blocks:

A basic block is a simple combination of statements. Except for entry and exit, the basic blocks do not have any branches like in and out. It means that the flow of control enters at the beginning and it always leaves at the end without any halt. The execution of a set of instructions of a basic block always takes place in the form of a sequence.

The first step is to divide a group of three-address codes into the basic block. The new basic block always begins with the first instruction and continues to add instructions until it reaches a jump or a label. If no jumps or labels are identified, the control will flow from one instruction to the next in sequential order.

The algorithm for the construction of the basic block is described below step by step:

Algorithm: The algorithm used here is partitioning the three-address code into basic blocks.
Input: A sequence of three-address codes will be the input for the basic blocks.
Output: A list of basic blocks with each three address statements, in exactly one block, is considered as the output.
Method: We'll start by identifying the intermediate code's leaders. The following are some guidelines for identifying leaders:

1. The first instruction in the intermediate code is generally considered as a leader.
2. The instructions that target a conditional or unconditional jump statement can be considered as a leader.
3. Any instructions that are just after a conditional or unconditional jump statement can be considered as a leader.

Each leader's basic block will contain all of the instructions from the leader until the instruction right before the following leader's start.

Example of basic block:
Three Address Code for the expression a = b + c – d is:

T1 = b + c

T2 = T1 - d

a = T2

This represents a basic block in which all the statements execute in a sequence one after the other.

Basic Block Construction:

Let us understand the construction of basic blocks with an example:

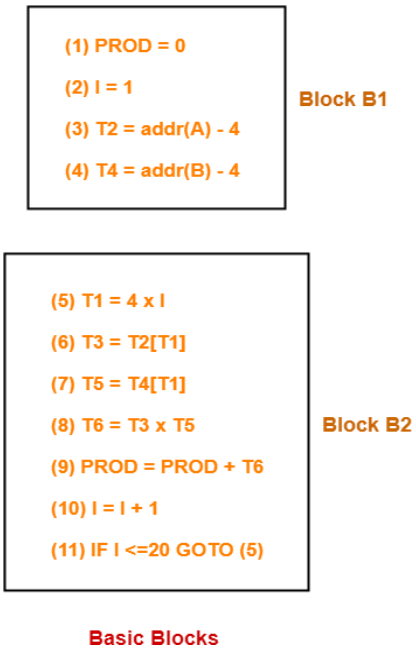
Example:

1. PROD = 0
2. I = 1
3. T2 = addr(A) – 4
4. T4 = addr(B) – 4
5. T1 = 4 x I
6. T3 = T2[T1]
7. T5 = T4[T1]
8. T6 = T3 x T5
9. PROD = PROD + T6
10. I = I + 1
11. IF I <=20 GOTO (5)

Using the algorithm given above, we can identify the number of basic blocks in the above three-address code easily-

There are two Basic Blocks in the above three-address code:

- **B1 –** Statement 1 to 4
- **B2 –** Statement 5 to 11

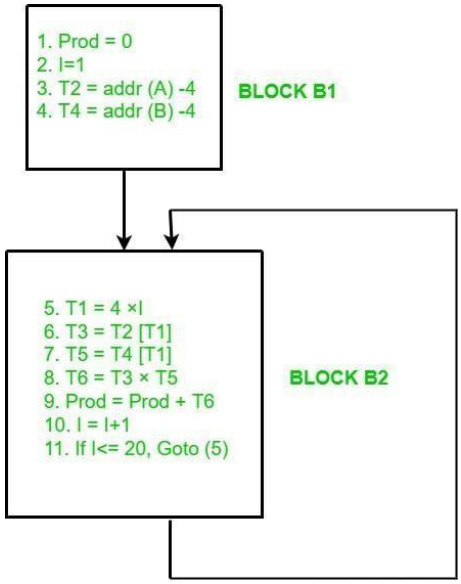


Basic Blocks

❖ **Flow Graph:**

A flow graph is simply a directed graph. For the set of basic blocks, a flow graph shows the flow of control information. A control flow graph is used to depict how the program control is being parsed among the blocks. A flow graph is used to illustrate the flow of control between basic blocks once an intermediate code has been partitioned into basic blocks. When the beginning instruction of the Y block follows the last instruction of the X block, an edge might flow from one block X to another block Y.

Let's make the flow graph of the example that we used for basic block formation:

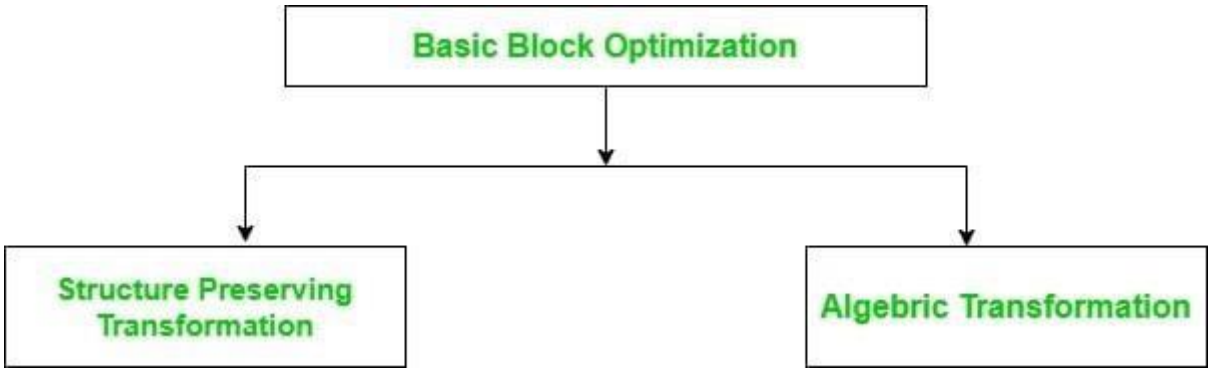


❖ **Optimization of Basic Blocks**

Optimization is applied to the basic blocks after the intermediate code generation phase of the compiler. Optimization is the process of transforming a program that improves the code by consuming fewer resources and delivering high speed. In optimization, high-level codes are replaced by their equivalent efficient low-level codes. Optimization of basic blocks can be machine-dependent or machine-independent. These transformations are useful for improving the quality of code that will be ultimately generated from basic block.

There are two types of basic block optimizations:

1. Function preserving transformations/ Structure preserving transformations
2. Algebraic transformations



Structure-Preserving Transformations:

The structure-preserving transformation on basic blocks includes:

1. Dead Code Elimination
2. Common Sub expression Elimination
3. Renaming of Temporary variables
4. Interchange of two independent adjacent statements

1. Dead Code Elimination:

Dead code is defined as that part of the code that never executes during the program execution. So, for optimization, such code or dead code is eliminated. The code which is never executed during the program (Dead code) takes time so, for optimization and speed, it is eliminated from the code. Eliminating the dead code increases the speed of the program as the compiler does not have to translate the dead code.

Example:

```
// Program with Dead code
int main()
{
   x = 2
   if (x > 2)
     cout << "code"; // Dead code
   else
     cout << "Optimization";
   return 0;
}
// Optimized Program without dead code
```

```
int main()
{
   x = 2;
   cout << "Optimization"; // Dead Code Eliminated
   return 0;
}
```

2. Common Sub expression Elimination:

In this technique, the sub-expression which are common are used frequently are calculated only once and reused when needed.

Example:

a = (x+y)+z; b = x+y;

Temp saves value for later use. Becomes

t = x+y; a = t+z; b = t;

3. Renaming of Temporary Variables:

Statements containing instances of a temporary variable can be changed to instances of a new temporary variable without changing the basic block value.

Example: Statement t = a + b can be changed to x = a + b where t is a temporary variable and x is a new temporary variable without changing the value of the basic block.

4. Interchange of Two Independent Adjacent Statements:

If a block has two adjacent statements which are independent can be interchanged without affecting the basic block value.

Example:
t1 = a + b

t2 = c + d

These two independent statements of a block can be interchanged without affecting the value of the block.

Algebraic Transformation:

Countless algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set. Some of the algebraic transformation on basic blocks includes:

1. Constant Folding
2. Copy Propagation
3. Strength Reduction

1. Constant Folding:
Solve the constant terms which are continuous so that compiler does not need to solve this expression.

Example:
x = 2 * 3 + y ⇒ x = 6 + y (Optimized code)

2. Copy Propagation:
It is of two types, Variable Propagation, and Constant Propagation.

Variable Propagation:

x = y ⇒ *z = y + 2 (Optimized code)*
z = x + 2

Constant Propagation:

x = 3 ⇒ *z = 3 + a (Optimized code)*
z = x + a

3. Strength Reduction:
Replace expensive statement/ instruction with cheaper ones.

x = 2 * y (costly) ⇒ x = y + y (cheaper)
x = 2 * y (costly) ⇒ x = y << 1 (cheaper)

❖ Principal Sources of Optimization:

A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.

Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

1. Function-Preserving Transformations

- There are a number of ways in which a compiler can improve a program without changing the function it computes.
- Function preserving transformations examples:
  - Common sub expression elimination

- Copy propagation,
- Dead-code elimination
- Constant folding

Common Sub expressions elimination:

- An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid re-computing the expression if we can use the previously computed value.
- For example

t1: = 4*i

t2: = a [t1]

t3: = 4*j

t4: = 4*i

t5: = n

t6: = b [t4] +t5

- The above code can be optimized using the common sub-expression elimination as

t1: = 4*i

t2: = a [t1]

t3: = 4*j

t5: = n

t6: = b [t1] +t5

- The common sub expression t4: =4*i is eliminated as its computation is already in t1 and the value of i is not been changed from definition to use.

Copy Propagation:

- Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.
- For example:

x=Pi;

```
A=x*r*r;
```

- The optimization using copy propagation can be done as follows: A=Pi*r*r;
- Here the variable x is eliminated

Dead-Code Eliminations:

- A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point.
- A related idea is dead or useless code, statements that compute values that never get used.
- While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.
- Example:

```
i=0;

if(i=1)

    {

     a=b+5;

     }
```

- Here, 'if' statement is dead code because this condition will never get satisfied.

Constant folding:

- Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.
- For example,

  a=6/2 can be replaced by

  a=3 there by eliminating a division operation

2. Loop Optimizations:

- In loops, especially in the inner loops, programs tend to spend the bulk of their time.
- The running time of a program may be improved if the number of instructions in an inner loop is decreased, even if we increase the amount of code outside that loop.

Three techniques are important for loop optimization:

1. Code motion- which moves code outside a loop;

2. Induction-variable elimination- which we apply to
replace variables from inner loop.

3.Reduction in strength -which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

Code Motion:

- An important modification that decreases the amount of code in a loop is code motion.
- This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop.
- Note that the notion "before the loop" assumes the existence of an entry for the loop.
- For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

while (i <= limit-2) /* statement does not change limit*/

Code motion will result in the equivalent of

t= limit-2;

while (i<=t) /* statement does not change limit or t */

Induction Variables :

- Loops are usually processed inside out.

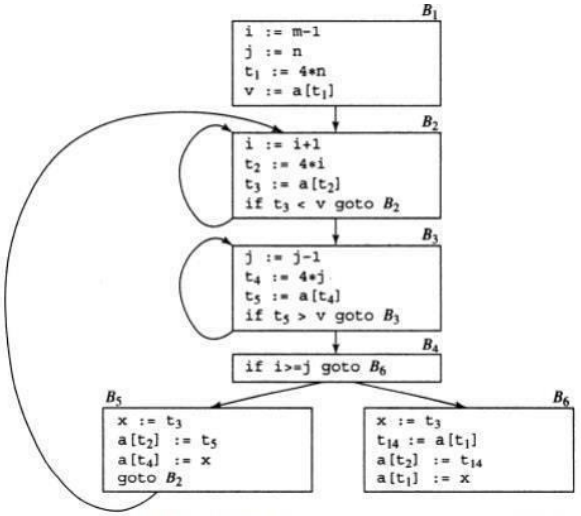


Fig. 5.3 B5 and B6 after common subexpression elimination

- For example consider the loop around B3. Note that the values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because 4*j is assigned to t4. Such identifiers are called induction variables.
- For the inner loop around B3 in Fig.5.3 we cannot get rid of either j or t4 completely; t4 is used in B3 and j in B4.
- As B3, t4 follows that just after the statement j:=j-1 the relationship t4:= 4*j-4 must hold.
- We may therefore replace the assignment t4:= 4*j by t4:= t4-4. The only problem is that t4 does not have a value when we enter block B3 for the first time. Since we must maintain the relationship t4=4*j on entry to the block B3,
- The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

Reduction In Strength:

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine.
- Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.
- For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine.

❖ DAG representation for basic blocks

A DAG for basic block is a directed acyclic graph with the following labels on nodes:

1. The leaves of graph are labelled by unique identifier and that identifier can be variable names or constants.
2. Interior nodes of the graph is labelled by an operator symbol.
3. Nodes are also given a sequence of identifiers for labels to store the computed value.

    - DAGs are a type of data structure. It is used to implement transformations on basic blocks.
    - DAG provides a good way to determine the common sub-expression.
    - It gives a picture representation of how the value computed by the statement is used in subsequent statements.

Algorithm for construction of DAG

Input: It contains a basic block

Output: It contains the following information:

- Each node contains a label. For leaves, the label is an identifier.
- Each node contains a list of attached identifiers to hold the computed values.

Case (i) x:= y OP z
Case (ii) x:= OP y
Case (iii) x:= y

Method:

Step 1:

If y operand is undefined then create node(y).

If z operand is undefined then for case(i) create node(z).

Step 2:

For case(i), create node(OP) whose right child is node(z) and left child is node(y).

For case(ii), check whether there is node(OP) with one child node(y).

For case(iii), node n will be node(y).

Output:

For node(x) delete x from the list of identifiers. Append x to attached identifiers list for the node n found in step 2. Finally set node(x) to n.

Example:

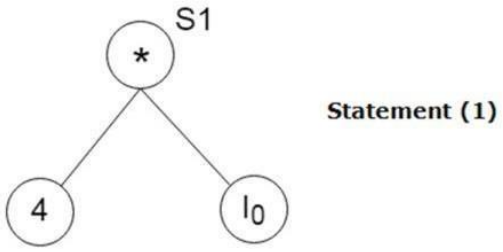
Consider the following three address statement:

1. S1:= 4 * i
2. S2:= a[S1]
3. S3:= 4 * i
4. S4:= b[S3]
5. S5:= s2 * S4
6. S6:= prod + S5
7. Prod:= s6

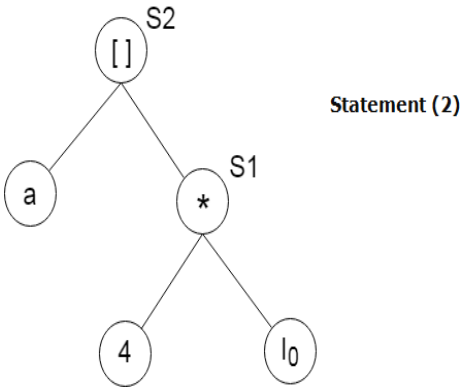
8. S7:= i+1
9. i := S7
10. **if** i<= 20 **goto** (1)

Stages in DAG Construction:

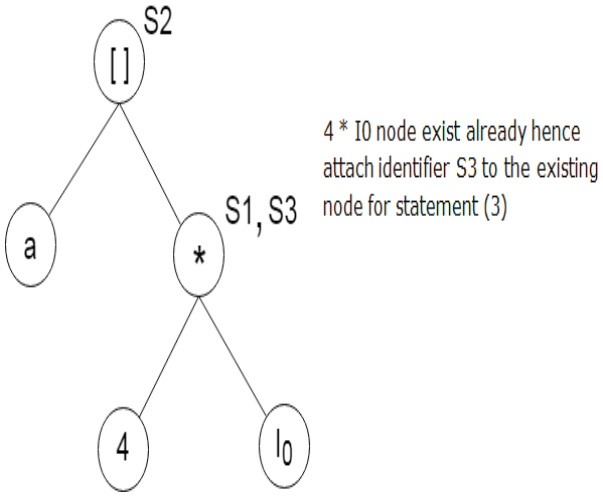
(a)



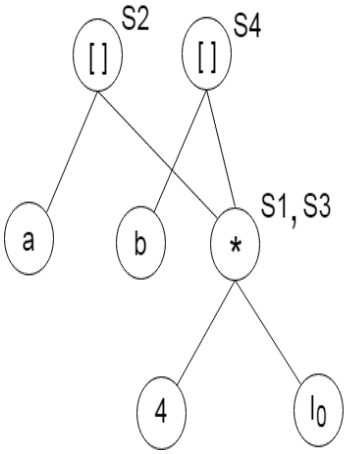
(b)



(c)

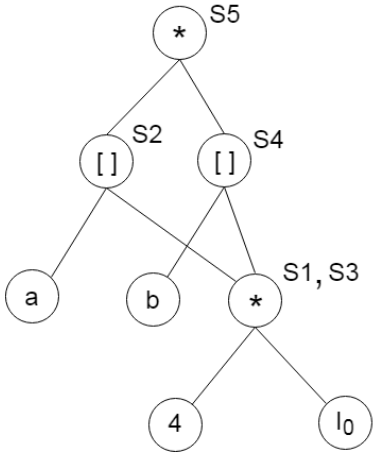


(d)



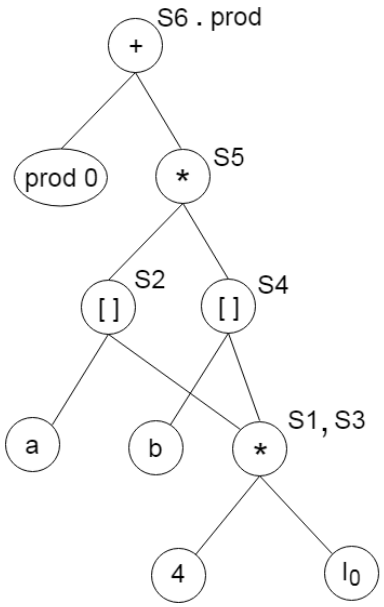
Statement (4)

(e)



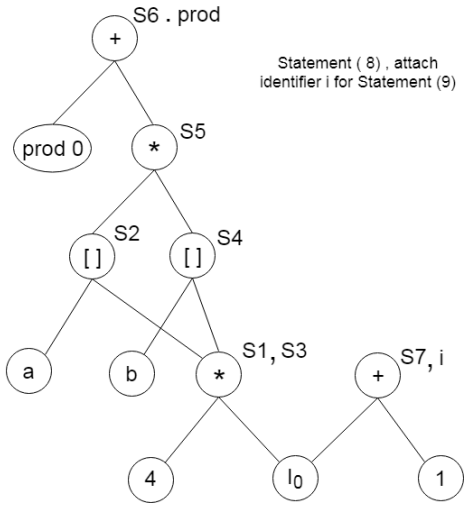
Statement (5)

(f)

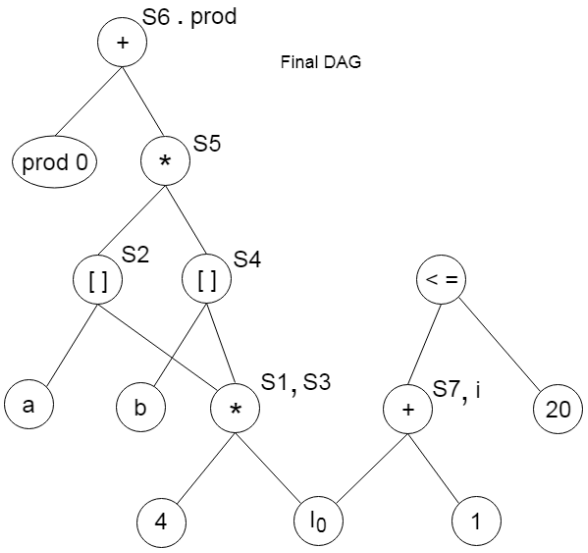


Statement (6), attach identifier prod for Statement (7)

(g)



(h)



Application of DAGs:

1. We can automatically detect common sub expressions.
2. We can determine which identifiers have their values used in the block.
3. We can determine which statements compute values that could be used outside the block.

❖ Simple Code Generator

Code generator is used to produce the target code for three-address statements. It uses registers to store the operands of the three address statement.

Example:

Consider the three address statement x:= y + z. It can have the following sequence of codes:

MOV x, R_0
ADD y, R_0

Register and Address Descriptors:

- A register descriptor contains the track of what is currently in each register. The register descriptors show that all the registers are initially empty.
- An address descriptor is used to store the location where current value of the name can be found at run time.

A code-generation algorithm:

The algorithm takes a sequence of three-address statements as input. For each three address statement of the form a:= b op c perform the various actions. These are as follows:

1. Invoke a function getreg to find out the location L where the result of computation b op c should be stored.
2. Consult the address description for y to determine y'. If the value of y currently in memory and register both then prefer the register y' . If the value of y is not already in L then generate the instruction **MOV y' , L** to place a copy of y in L.
3. Generate the instruction **OP z' , L** where z' is used to show the current location of z. if z is in both then prefer a register to a memory location. Update the address descriptor of x to indicate that x is in location L. If x is in L then update its descriptor and remove x from all other descriptor.
4. If the current value of y or z have no next uses or not live on exit from the block or in register then alter the register descriptor to indicate that after execution of x : = y op z those register will no longer contain y or z.

Generating Code for Assignment Statements:

The assignment statement d:= (a-b) + (a-c) + (a-c) can be translated into the following sequence of three address code:

t:= a-b
u:= a-c
v:= t +u
d:= v+u

Code sequence for the example is as follows:

Statement	Code Generated	Register descriptor Register empty	Address descriptor
t:= a - b	MOV a,R0 SUB b,R0	R0 contains t	t in R0
u:= a - c	MOV a,R1 SUB c,R1	R0 contains t R1 Contains u	t in R0 u in R0
v:= t + u	ADD R1, R0	R0 contains u R1 Contains v	u in R1 v in R1
d:= v + u	ADD R1,R0 MOV R0,d	R0 contains d	d in R0 d in R0 and Memory

❖ REGISTER ALLOCATION AND ASSIGNMENT

What is Register Allocation?
Register allocation deals with various strategies for deciding what values in a program should reside in registers

Why is Register Allocation Important?
Generally, the instructions involving only register operands are faster than those involving memory operands. So, efficient utilization of available registers is important in generating good code, and register allocation plays a vital role in optimization.

When is Register Allocation Done?
- It can be done in intermediate language prior to the machine code generation, or it can be done in the machine language
- In the latter case, the machine code uses symbolic names for registers, and the register allocation turns into register numbers
- Register allocation in the intermediate language has the advantage that the same register allocator can be used for several target machines

What are the approaches for Register Allocation?
There are two approaches, which are
- Local allocators: These are based on usage counts
- Global or intraproceduaral allocators: These are based on the concept of graph coloring

Local register allocation
Register allocation is only within a basic block. It follows top-down approach.
Assign registers to the most heavily used variables
- Traverse the block
- Count uses

- Use count as a priority function
- Assign registers to higher priority variables first

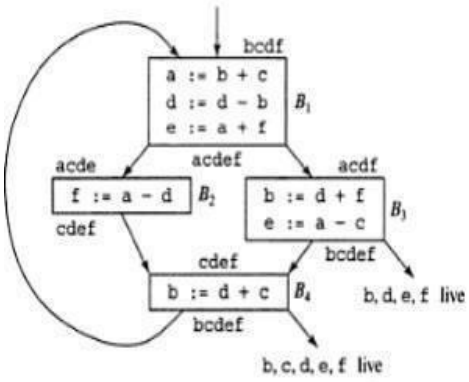


Fig 4.3 Flow graph of an inner loop

- **Usage Count**

  Use(a,B2)+2*(live(a,B2) = 1+2*0=1

  Use(b,B3)+2*(live(a,B2)=1+2*1=3

  If usage count of a=4 b=6 c= 4 d=5 e=4 f=4

R0	R1	R2	R3
b	d	a/c/f	……

Advantage

Heavily used values reside in registers

Disadvantage

Does not consider non-uniform distribution of uses

Global register allocation

The method is also very simple and involves two passes:

- In the first pass, the target machine instructions or intermediate code instructions are selected as though there are infinite number of symbolic registers
- In the second pass, for each procedure, a register inference graph is constructed in which the nodes are symbolic registers and an edge connects to nodes if one is live at a point where the other is defined, i.e., if both are live at the same time

Local allocation does not take into account that some instructions (e.g. those in loops) execute more frequently. It forces us to store/load at basic block endpoints since each block has no knowledge of the context of others.

To find out the live range(s) of each variable and the area(s) where the variable is used/defined global allocation is needed. Cost of spilling will depend on frequencies and locations of uses.

Register allocation depends on:

- Size of live range
- Number of uses/definitions
- Frequency of execution
- Number of loads/stores needed.
- Cost of loads/stores needed.

Register allocation by graph colouring

Now, an attempt is made to color the graph using k colors, where k is the number of available registers. We know that a graph is said to be colored if each node has been assigned a color in such a way that no two adjacent nodes have the same color

Once the coloring is over, register allocation can be made

Global register allocation can be seen as a graph colouring problem.

Basic idea:

1. Identify the live range of each variable

2. Build an interference graph that represents conflicts between live ranges (two nodes are connected if the variables they represent are live at the same moment)
3. Try to assign as many colours to the nodes of the graph as there are registers so that two neighbours have different colours

❖ PEEPHOLE OPTIMIZATION

A statement-by-statement code-generations strategy often produces target code that contains redundant instructions and suboptimal constructs. The quality of such target code can be improved by applying "optimizing" transformations to the target program.

A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.

The peephole is a small, moving window on the target program. The code in the peephole need not be contiguous, although some implementations do require this. It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.

Objectives of Peephole Optimization:

The objective of peephole optimization is as follows:

1. To improve performance
2. To reduce memory footprint
3. To reduce code size

Characteristics of peephole optimizations:

- Redundant-instructions elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms
- Eliminating Unreachable code

Redundant Loads And Stores:

If we see the instructions sequence
(1) MOV R0,a
(2) MOV a,R0

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of a is already in register R0.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

Unreachable Code:

Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabelled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable debug is 1. In C, the source code might look like:

```
#define debug 0
….

If ( debug ) {
Print debugging information

}
```

In the intermediate representations the if-statement may be translated as:

If debug =1 goto L1 goto L2

L1: print debugging information L2: .....(a)

One obvious peephole optimization is to eliminate jumps over jumps .Thus no matter what the value of debug; (a) can be replaced by:

```
        If debug ≠1 goto L2
        Print debugging information
L2:.....(b)

        If debug ≠0 goto L2
        Print debugging information
L2:.....(c)
```

As the argument of the statement of (c) evaluates to a constant true it can be replaced

By goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

Flows-Of-Control Optimizations:

The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

goto L1
….

L1: gotoL2 (d)

by the sequence

goto L2
….

L1: goto L2

If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump .Similarly, the sequence

if a < b goto L1
….

L1: goto L2 (e)

can be replaced by

If a < b goto L2

….

L1: goto L2

Ø Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

goto L1

L1: if a < b goto L2 (f) L3:

may be replaced by

If a < b goto L2
goto L3

…….

L3:

While the number of instructions in(e) and (f) is the same, we sometimes skip the unconditional jump in (f), but never in (e).Thus (f) is superior to (e) in execution time

Algebraic Simplification:

There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them. For example, statements such as

x := x+0 or
x := x * 1

are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

Reduction in Strength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

$X^2 \rightarrow X*X$

Use of Machine Idioms:

The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like i : =i+1.

i:=i+1 → i++
i:=i-1 → i- -