**Grammars:** Notation and concepts for languages and Grammars, sets and string, Discussion and classification of Grammars, Scanner regular expression, regular definition, finite automata, LL and LR Grammars, ambiguous grammar.

## Grammar

A grammar **G** can be formally written as a 4-tuple (N, T, S, P) where −

* **N** or **V*N*** is a set of variables or non-terminal symbols.
* **T** or **∑** is a set of Terminal symbols.
* **S** is a special variable called the Start symbol, S ∈ N
* **P** is Production rules for Terminals and Non-terminals. A production rule has the form α → β, where α and β are strings on V*N* ∪ ∑ and least one symbol of α belongs to VN.

### Example

Grammar G1 −

({S, A, B}, {a, b}, S, {S → AB, A → a, B → b})

Here,

* **S, A,** and **B** are Non-terminal symbols;
* **a** and **b** are Terminal symbols
* **S** is the Start symbol, S ∈ N
* Productions, **P : S → AB, A → a, B → b**

### Example

Grammar G2 −

(({S, A}, {a, b}, S,{S → aAb, aA → aaAb, A → ε } )

Here,

* **S** and **A** are Non-terminal symbols.
* **a** and **b** are Terminal symbols.
* **ε** is an empty string.
* **S** is the Start symbol, S ∈ N
* Production **P : S → aAb, aA → aaAb, A → ε**

## Derivations from a Grammar

Strings may be derived from other strings using the productions in a grammar. If a grammar **G** has a production **α → β**, we can say that **x α y** derives **x β y** in **G**. This derivation is written as −

***x α y ⇒G x β y***

### Example

Let us consider the grammar −

G2 = ({S, A}, {a, b}, S, {S → aAb, aA → aaAb, A → ε } )

Some of the strings that can be derived are −

S ⇒ aAb using production S → aAb

⇒ aaAbb using production aA → aAb

⇒ aaaAbbb using production aA → aAb

⇒ aaabbb using production A → ε

The set of all strings that can be derived from a grammar is said to be the language generated from that grammar. A language generated by a grammar **G** is a subset formally defined by

L(G)={W|W ∈ ∑\*, S ⇒G **W**}

If **L(G1) = L(G2)**, the Grammar **G1** is equivalent to the Grammar **G2**.

### Example

If there is a grammar

G: N = {S, A, B} T = {a, b} P = {S → AB, A → a, B → b}

Here **S** produces **AB**, and we can replace **A** by **a**, and **B** by **b**. Here, the only accepted string is **ab**, i.e.,

L(G) = {ab}

### Example

Suppose we have the following grammar −

G: N = {S, A, B} T = {a, b} P = {S → AB, A → aA|a, B → bB|b}

The language generated by this grammar −

L(G) = {ab, a2b, ab2, a2b2, ………}

= {am bn | m ≥ 1 and n ≥ 1}

## Construction of a Grammar Generating a Language

We’ll consider some languages and convert it into a grammar G which produces those languages.

### Example

***Problem*** − Suppose, L (G) = {am bn | m ≥ 0 and n > 0}. We have to find out the grammar **G** which produces **L(G)**.

***Solution***

Since L(G) = {am bn | m ≥ 0 and n > 0}

the set of strings accepted can be rewritten as −

L(G) = {b, ab,bb, aab, abb, …….}

Here, the start symbol has to take at least one ‘b’ preceded by any number of ‘a’ including null.

To accept the string set {b, ab, bb, aab, abb, …….}, we have taken the productions −

S → aS , S → B, B → b and B → bB

S → B → b (Accepted)

S → B → bB → bb (Accepted)

S → aS → aB → ab (Accepted)

S → aS → aaS → aaB → aab(Accepted)

S → aS → aB → abB → abb (Accepted)

Thus, we can prove every single string in L(G) is accepted by the language generated by the production set.

Hence the grammar −

G: ({S, A, B}, {a, b}, S, { S → aS | B , B → b | bB })

### Example

***Problem*** − Suppose, L (G) = {am bn | m > 0 and n ≥ 0}. We have to find out the grammar G which produces L(G).

***Solution*** −

Since L(G) = {am bn | m > 0 and n ≥ 0}, the set of strings accepted can be rewritten as −

L(G) = {a, aa, ab, aaa, aab ,abb, …….}

Here, the start symbol has to take at least one ‘a’ followed by any number of ‘b’ including null.

To accept the string set {a, aa, ab, aaa, aab, abb, …….}, we have taken the productions −

S → aA, A → aA , A → B, B → bB ,B → λ

S → aA → aB → aλ → a (Accepted)

S → aA → aaA → aaB → aaλ → aa (Accepted)

S → aA → aB → abB → abλ → ab (Accepted)

S → aA → aaA → aaaA → aaaB → aaaλ → aaa (Accepted)

S → aA → aaA → aaB → aabB → aabλ → aab (Accepted)

S → aA → aB → abB → abbB → abbλ → abb (Accepted)

Thus, we can prove every single string in L(G) is accepted by the language generated by the production set.

Hence the grammar −

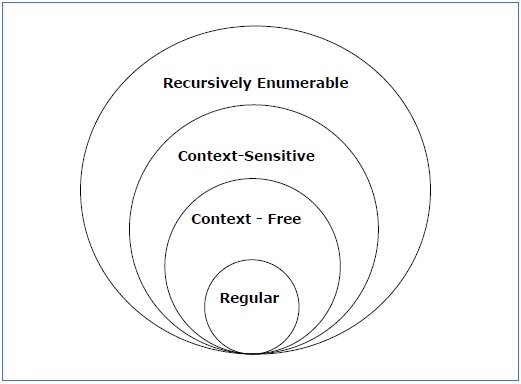
G: ({S, A, B}, {a, b}, S, {S → aA, A → aA | B, B → λ | bB })

# Chomsky Classification of Grammars

According to Noam Chomosky, there are four types of grammars − Type 0, Type 1, Type 2, and Type 3. The following table shows how they differ from each other −

|  |  |  |  |
| --- | --- | --- | --- |
| **Grammar Type** | **Grammar Accepted** | **Language Accepted** | **Automaton** |
| Type 0 | Unrestricted grammar | Recursively enumerable language | Turing Machine |
| Type 1 | Context-sensitive grammar | Context-sensitive language | Linear-bounded automaton |
| Type 2 | Context-free grammar | Context-free language | Pushdown automaton |
| Type 3 | Regular grammar | Regular language | Finite state automaton |

Take a look at the following illustration. It shows the scope of each type of grammar −



## Type - 3 Grammar

**Type-3 grammars** generate regular languages. Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal.

The productions must be in the form **X → a or X → aY**

where **X, Y ∈ N** (Non terminal)

and **a ∈ T** (Terminal)

The rule **S → ε** is allowed if **S** does not appear on the right side of any rule.

### Example

X → ε

X → a | aY

Y → b

## Type - 2 Grammar

**Type-2 grammars** generate context-free languages.

The productions must be in the form **A → γ**

where **A ∈ N** (Non terminal)

and **γ ∈ (T ∪ N)\*** (String of terminals and non-terminals).

These languages generated by these grammars are be recognized by a non-deterministic pushdown automaton.

### Example

S → X a

X → a

X → aX

X → abc

X → ε

## Type - 1 Grammar

**Type-1 grammars** generate context-sensitive languages. The productions must be in the form

**α A β → α γ β**

where **A ∈ N** (Non-terminal)

and **α, β, γ ∈ (T ∪ N)\*** (Strings of terminals and non-terminals)

The strings **α** and **β** may be empty, but **γ** must be non-empty.

The rule **S → ε** is allowed if S does not appear on the right side of any rule. The languages generated by these grammars are recognized by a linear bounded automaton.

### Example

AB → AbBc

A → bcA

B → b

## Type - 0 Grammar

**Type-0 grammars** generate recursively enumerable languages. The productions have no restrictions. They are any phase structure grammar including all formal grammars.

They generate the languages that are recognized by a Turing machine.

The productions can be in the form of **α → β** where **α** is a string of terminals and nonterminals with at least one non-terminal and **α** cannot be null. **β** is a string of terminals and non-terminals.

### Example

S → ACaB

Bc → acB

CB → DB

aD → Db

Regular Grammer:

**Regular Grammar :** A grammar is regular if it has rules of form A -> a or A -> aB or A -> ɛ where ɛ is a special symbol called NULL.

   
**Regular Languages :** A language is regular if it can be expressed in terms of

## 

**Regular Expression:**

A **Regular Expression** can be recursively defined as follows −

* **ε** is a Regular Expression indicates the language containing an empty string. **(L (ε) = {ε})**
* **φ** is a Regular Expression denoting an empty language. **(L (φ) = { })**
* **x** is a Regular Expression where **L = {x}**
* If **X** is a Regular Expression denoting the language **L(X)** and **Y** is a Regular Expression denoting the language **L(Y)**, then
  + **X + Y** is a Regular Expression corresponding to the language **L(X) ∪ L(Y)** where **L(X+Y) = L(X) ∪ L(Y)**.
  + **X . Y** is a Regular Expression corresponding to the language **L(X) . L(Y)** where **L(X.Y) = L(X) . L(Y)**
  + **R\*** is a Regular Expression corresponding to the language **L(R\*)**where **L(R\*) = (L(R))\***
* If we apply any of the rules several times from 1 to 5, they are Regular Expressions.

**Some RE Examples**

|  |  |
| --- | --- |
| **Regular Expressions** | **Regular Set** |
| (0 + 10\*) | L = { 0, 1, 10, 100, 1000, 10000, … } |
| (0\*10\*) | L = {1, 01, 10, 010, 0010, …} |
| (0 + ε)(1 + ε) | L = {ε, 0, 1, 01} |
| (a+b)\* | Set of strings of a’s and b’s of any length including the null string. So L = { ε, a, b, aa , ab , bb , ba, aaa…….} |
| (a+b)\*abb | Set of strings of a’s and b’s ending with the string abb. So L = {abb, aabb, babb, aaabb, ababb, …………..} |
| (11)\* | Set consisting of even number of 1’s including empty string, So L= {ε, 11, 1111, 111111, ……….} |
| (aa)\*(bb)\*b | Set of strings consisting of even number of a’s followed by odd number of b’s , so L = {b, aab, aabbb, aabbbbb, aaaab, aaaabbb, …………..} |
| (aa + ab + ba + bb)\* | String of a’s and b’s of even length can be obtained by concatenating any combination of the strings aa, ab, ba and bb including null, so L = {aa, ab, ba, bb, aaab, aaba, …………..} |

The lexical analyzer needs to scan and identify only a finite set of valid string/token/lexeme that belong to the language in hand. It searches for the pattern defined by the language rules.

**Question 2(a)2017 Describe the constructing rules for regular expression**

Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expressions is known as **regular grammar**. The language defined by regular grammar is known as **regular language**.

Regular expression is an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions serve as names for a set of strings.

There are a **number of algebraic laws that are obeyed by regular expressions,** which can be used to manipulate regular expressions into equivalent forms.

## 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.

**Notations**

If r and s are regular expressions denoting the languages L(r) and L(s), then

* **Union** : (r)|(s) is a regular expression denoting L(r) U L(s)
* **Concatenation** : (r)(s) is a regular expression denoting L(r)L(s)
* **Kleene closure** : (r)\* is a regular expression denoting (L(r))\*
* (r) is a regular expression denoting L(r)

**Union :** If L1 and If L2 are two regular languages, their union L1 ∪ L2 will also be regular. For example, L1 = {an | n ≥ 0} and L2 = {bn | n ≥ 0}  
L3 = L1 ∪ L2 = {an ∪ bn | n ≥ 0} is also regular.  
**Intersection :** If L1 and If L2 are two regular languages, their intersection L1 ∩ L2 will also be regular. For example,  
L1= {am bn | n ≥ 0 and m ≥ 0} and L2= {am bn ∪ bn am | n ≥ 0 and m ≥ 0}  
L3 = L1 ∩ L2 = {am bn | n ≥ 0 and m ≥ 0} is also regular.  
**Concatenation :** If L1 and If L2 are two regular languages, their concatenation L1.L2 will also be regular. For example,  
L1 = {an | n ≥ 0} and L2 = {bn | n ≥ 0}  
L3 = L1.L2 = {am . bn | m ≥ 0 and n ≥ 0} is also regular.  
**Kleene Closure :** If L1 is a regular language, its Kleene closure L1\* will also be regular. For example,  
L1 = (a ∪ b)  
L1\* = (a ∪ b)\*

**Precedence and Associativity**

* \*, concatenation (.), and | (pipe sign) are left associative
* \* has the highest precedence
* Concatenation (.) has the second highest precedence.
* | (pipe sign) has the lowest precedence of all.

**Representing valid tokens of a language in regular expression**

If x is a regular expression, then:

* x\* means zero or more occurrence of x.

i.e., it can generate { e, x, xx, xxx, xxxx, … }

* x+ means one or more occurrence of x.

i.e., it can generate { x, xx, xxx, xxxx … } or x.x\*

* x? means at most one occurrence of x

i.e., it can generate either {x} or {e}.

[a-z] is all lower-case alphabets of English language.

[A-Z] is all upper-case alphabets of English language.

[0-9] is all natural digits used in mathematics.

**Representing occurrence of symbols using regular expressions**

letter = [a – z] or [A – Z]

digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 or [0-9]

sign = [ + | - ]

**Representing language tokens using regular expressions**

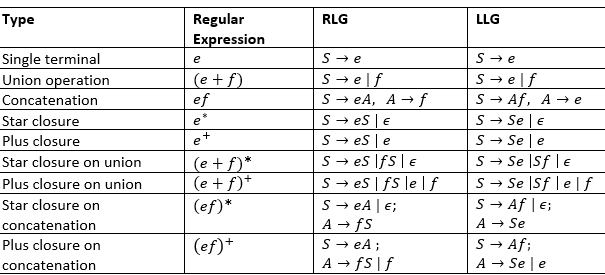
Decimal = (sign)?(digit)+

Identifier = (letter)(letter | digit)\*

The only problem left with the lexical analyzer is how to verify the validity of a regular expression used in specifying the patterns of keywords of a language. A well-accepted solution is to use finite automata for verification.

**Convert Regular expression to regular grammer:**

intuitive rules to convert basic/minimal regular expressions directly to regular grammar (RLG for Right Linear Grammars, LLG for Left Linear Grammars):



**Regular Grammer to regular expression:**

* S --> b | AA
* A --> aA | Abb | ϵ

**Question 2(b) 2017 Convert the following regular grammers to regular expressions?**

Answer:

|  |  |
| --- | --- |
| Regular Grammer | Regular Expression |
|  |  |
|  |  |
|  |  |
|  |  |

**Question2(c) Discuss the importance of regular expression and context free grammer**

Answer:

**Importance of Regular expression:**

Regular expressions are used in [search engines](https://en.wikipedia.org/wiki/Search_engine), search and replace dialogs of [word processors](https://en.wikipedia.org/wiki/Word_processor) and [text editors](https://en.wikipedia.org/wiki/Text_editor), in text processing utilities such as [sed](https://en.wikipedia.org/wiki/Sed) and [AWK](https://en.wikipedia.org/wiki/AWK) and in [lexical analysis](https://en.wikipedia.org/wiki/Lexical_analysis). Many [programming languages](https://en.wikipedia.org/wiki/Programming_language) provide regex capabilities either built-in or via [libraries](https://en.wikipedia.org/wiki/Library_(computing)).

1. a sequence of symbols and characters expressing a string or pattern to be searched for within a longer piece of text.
2. A regular expression is a text pattern consisting of a combination of alphanumeric characters and special characters known as meta characters.

You'll find that regular expressions are used in three different ways:

* Regular text match,
* search and
* replace and
* splitting.

The latter is basically the same as the reverse match i.e. everything the regular expression did not match. Regular expressions are often simply called regexps or RE, but for consistency I'll be referring to it with its full name.

3. Regular expressions can be used to perform all types of text search and text replace operations.

**Importance of Context free grammer:**

# Finite Automata:

Finite automata is a state machine that takes a string of symbols as input and changes its state accordingly.

Finite automata is a recognizer for regular expressions. When a regular expression string is fed into finite automata, it changes its state for each literal. 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:

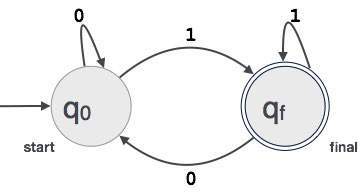
* Finite set of states (Q)
* Finite set of input symbols (Σ)
* One Start state (q0)
* Set of final states (qf)
* Transition function (δ)

The transition function (δ) maps the finite set of state (Q) to a finite set of input symbols (Σ), Q × Σ ➔ Q

**Finite Automata Construction**

Let L(r) be a regular language recognized by some finite automata (FA).

* **States** : States of FA are represented by circles. State names are written inside circles.
* **Start state** : The state from where the automata starts, is known as the start state. Start state has an arrow pointed towards it.
* **Intermediate states** : All intermediate states have at least two arrows; one pointing to and another pointing out from them.
* **Final state** : If the input string is successfully parsed, the automata is expected to be in this state. Final state is represented by double circles. It may have any odd number of arrows pointing to it and even number of arrows pointing out from it. The number of odd arrows are one greater than even, i.e. **odd = even+1**.
* **Transition** : The transition from one state to another state happens when a desired symbol in the input is found. Upon transition, automata can either move to the next state or stay in the same state. Movement from one state to another is shown as a directed arrow, where the arrows points to the destination state. If automata stays on the same state, an arrow pointing from a state to itself is drawn.

**Example** : We assume FA accepts any three digit binary value ending in digit 1. FA = {Q(q0, qf), Σ(0,1), q0, qf, δ}

**Finite Automaton can be classified into two types –**

* Deterministic Finite Automaton (DFA)
* Non-deterministic Finite Automaton (NDFA / NFA)

**Question:3(a) 2017 Define DFA. How does DFA contribute to design a compiler.**

**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.**

**Formal Definition of a DFA**

A DFA can be represented by a 5-tuple (Q, ∑, δ, q0, F) where −

* **Q** is a finite set of states.
* **∑** is a finite set of symbols called the alphabet.
* **δ** is the transition function where δ: Q × ∑ → Q
* **q0** is the initial state from where any input is processed (q0 ∈ Q).
* **F** is a set of final state/states of Q (F ⊆ Q).

**Graphical Representation of a DFA**

A DFA is represented by digraphs called **state diagram**.

* The vertices represent the states.
* The arcs labeled with an input alphabet show the transitions.
* The initial state is denoted by an empty single incoming arc.
* The final state is indicated by double circles.

**Example**

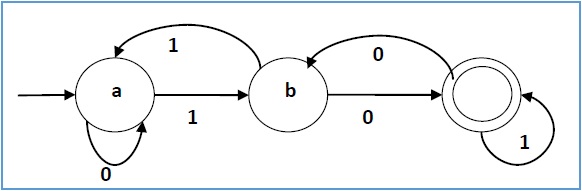
Let a deterministic finite automaton be →

* Q = {a, b, c},
* ∑ = {0, 1},
* q0 = {a},
* F = {c}, and

Transition function δ as shown by the following table −

|  |  |  |
| --- | --- | --- |
| **Present State** | **Next State for Input 0** | **Next State for Input 1** |
| **a** | a | b |
| **b** | c | a |
| **c** | b | c |

Its graphical representation would be as follows –



**How does DFA contribute to design a compiler.**

DFAs are one of the most important thing to design a compiler, since there is a trivial linear time, constant-space, [online algorithm](https://en.wikipedia.org/wiki/Online_algorithm) to simulate a DFA on a stream of input. Also, there are efficient algorithms to find a DFA recognizing:

* the complement of the language recognized by a given DFA.
* the union/intersection of the languages recognized by two given DFAs.

Because DFAs can be reduced to a canonical form ([minimal DFAs](https://en.wikipedia.org/wiki/Dfa_minimization)), there are also efficient algorithms to determine:

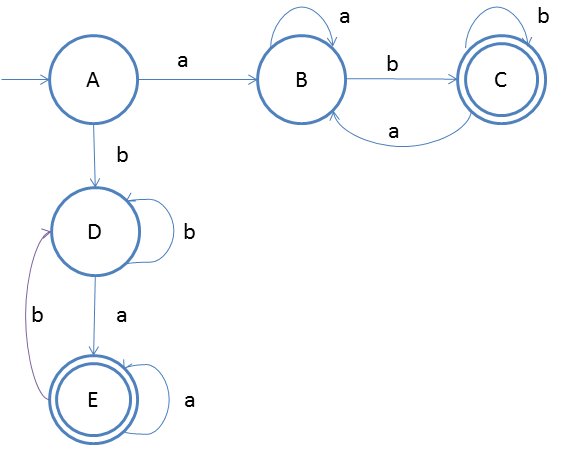
* whether a DFA accepts any strings (Emptiness Problem)
* whether a DFA accepts all strings (Universality Problem)
* whether two DFAs recognize the same language (Equality Problem)
* whether the language recognized by a DFA is included in the language recognized by a second DFA (Inclusion Problem)
* the DFA with a minimum number of states for a particular regular language (Minimization Problem)

DFAs are equivalent in computing power to [nondeterministic finite automata](https://en.wikipedia.org/wiki/Nondeterministic_finite_automata) (NFAs). This is because, firstly any DFA is also an NFA, so an NFA can do what a DFA can do. Also, given an NFA, using the [powerset construction](https://en.wikipedia.org/wiki/Powerset_construction) one can build a DFA that recognizes the same language as the NFA, although the DFA could have exponentially larger number of states than the NFA

**Question:3(b) 2017 construct DFAs accepting the following language:**

**Answer: i.**

Clearly the language is infinite because there is infinite number of strings.



The idea behind this approach is very simple, follow the steps below and you will understand.

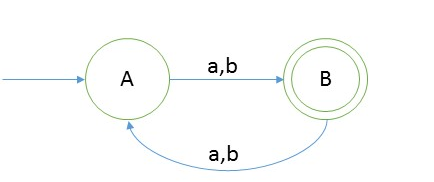
1. We know that if the first input is 'a' then the last input **cannot be** 'a', so we start two flows form start state, one for staring with 'a' and another for starting with 'b'
2. This is also clear that there will be two final states
3. In the first flow starting with 'a' we have already discussed the process in the previous examples. We will accept the input 'a' on state B for multiple times and then on b we will go to final state.
4. On final state we will direct the input 'a' to state B so that the last symbol should not be 'a'
5. The same mechanism will be for another flow staring from start state A on input 'a'.

**Testing 1**

1. Lets take one input string abab(this will describe for strings which starts with a and ends with different symbol b)
2. Scan string from left to right
3. First input is a, so from state A we will go to state B
4. Second input is b, so from state B we will go to state C
5. Third input is a, so from state C we will go to state B
6. Fourth input is b, so from state B we will go to state C **(final state)**

*After end of the string we are at final state so string is accepted.*

**Answer:**

Create a DFA which accepts strings of odd length  
  
**Explanation**  
As we can see that length of string should be even for that language will be = {a, b, bab, aba, aaa, bbb, baa, aaaaa, bbbbb, ….}  
And the language is infinite.

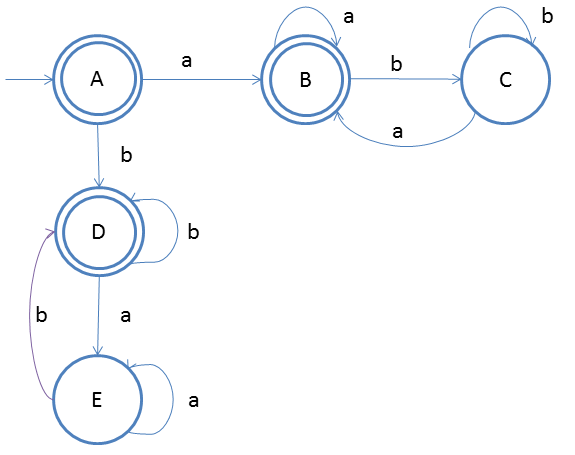
#### Testing

1. Take a string ‘abbbb’ to test whether it is accepted in the above DFA
2. Scan string from left to right
3. first input we get is **a** and from start state(A) on ‘a’ we go to B
4. second input is b, from state B on b we will go to start state again.
5. third input is b, from A we go to second state.
6. from second state we go to first state on fourth input symbol b
7. from A on fifth symbol we will go to state B.
8. Now we reached end of string and we are standing on final state.

So we say that string is accepted.

**Exercise:**

Given: Input alphabet, Σ={a, b}  
Language L = {ε, a, b, aa, bb, aba, bab, ababa, aabba, aaabbba,...}

Clearly the language is infinite because there is infinite number of strings.  
This DFA is complement of the previous example.  
We will explain complementation in the next section. The idea behind this approach is very simple, follow the steps below and you will understand.

1. We know that if the first input is 'a' then the last input **should be** 'a', so we start two flows form start state, one for staring with 'a' and another for starting with 'b'
2. Now just design the very basic DFA accepting for ε, that means the start state should final state.
3. In the first flow when 'a' comes form the input string then we should also reach final state as 'a' is also satify the property of DFA
4. After 'a' if any number of 'a' comes then we do not have any problem, that is why a self-loop on state B
5. On state B if b comes then it should be accepted as B is final state so we will create one more state C to get the 'b'
6. On state C if any number of 'b' comes we do not have issues but if 'a' comes then we have to accept it so we will direct input to state B
7. The same mechanism will be for another flow staring from start state A on input 'b'.

**Testing 1**

1. Lets take one input string aba (this will describe for strings which starts with a and ends with same symbol a)
2. Scan string from left to right
3. First input is a, so from state A we will go to state B
4. Second input is b, so from state B we will go to state C
5. Third input is a, so from state C we will go to state B **(final state)**

*After end of the string we are at final state so string is accepted.*

**NonDeterministic Finite Autometa(NFA)**

In NDFA, for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. Hence, it is called **Non-deterministic Automaton**. As it has finite number of states, the machine is called **Non-deterministic Finite Machine** or **Non-deterministic Finite Automaton**.

**Formal Definition of an NDFA**

An NDFA can be represented by a 5-tuple (Q, ∑, δ, q0, F) where −

* **Q** is a finite set of states.
* **∑** is a finite set of symbols called the alphabets.
* **δ** is the transition function where δ: Q × ∑ → 2Q

(Here the power set of Q (2Q) has been taken because in case of NDFA, from a state, transition can occur to any combination of Q states)

* **q0** is the initial state from where any input is processed (q0 ∈ Q).
* **F** is a set of final state/states of Q (F ⊆ Q).

**Graphical Representation of an NDFA: (same as DFA)**

An NDFA is represented by digraphs called state diagram.

* The vertices represent the states.
* The arcs labeled with an input alphabet show the transitions.
* The initial state is denoted by an empty single incoming arc.
* The final state is indicated by double circles.

**Example**

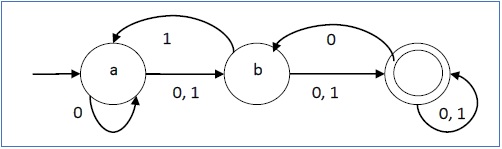
Let a non-deterministic finite automaton be →

* Q = {a, b, c}
* ∑ = {0, 1}
* q0 = {a}
* F = {c}

The transition function δ as shown below −

|  |  |  |  |
| --- | --- | --- | --- |
| **Present State** | **Next State for Input 0** | **Next State for Input 1** | |
| a | a, b | b | |
| b | c | a, c | |
| c | b, c | c |

Its graphical representation would be as follows −



**Question: 3(a) 2015 different between DFA and NFA. Give an example**

**DFA vs NDFA**

The following table lists the differences between DFA and NDFA.

|  |  |
| --- | --- |
| **DFA** | **NDFA** |
| The transition from a state is to a single particular next state for each input symbol. Hence it is called *deterministic*. | The transition from a state can be to multiple next states for each input symbol. Hence it is called *non-deterministic*. |
| Empty string transitions are not seen in DFA. | NDFA permits empty string transitions. |
| Backtracking is allowed in DFA | In NDFA, backtracking is not always possible. |
| Requires more space. | Requires less space. |
| A string is accepted by a DFA, if it transits to a final state. | A string is accepted by a NDFA, if at least one of all possible transitions ends in a final state. |

**Acceptors, Classifiers, and Transducers**

**Acceptor (Recognizer)**

An automaton that computes a Boolean function is called an **acceptor**. All the states of an acceptor is either accepting or rejecting the inputs given to it.

**Classifier**

A **classifier** has more than two final states and it gives a single output when it terminates.

**Transducer**

An automaton that produces outputs based on current input and/or previous state is called a **transducer**. Transducers can be of two types −

* **Mealy Machine** − The output depends both on the current state and the current input.
* **Moore Machine** − The output depends only on the current state.

**Acceptability by DFA and NDFA**

A string is accepted by a DFA/NDFA iff the DFA/NDFA starting at the initial state ends in an accepting state (any of the final states) after reading the string wholly.

A string S is accepted by a DFA/NDFA (Q, ∑, δ, q0, F), iff

**δ\*(q0, S) ∈ F**

The language **L** accepted by DFA/NDFA is

**{S | S ∈ ∑\* and δ\*(q0, S) ∈ F}**

A string S′ is not accepted by a DFA/NDFA (Q, ∑, δ, q0, F), iff

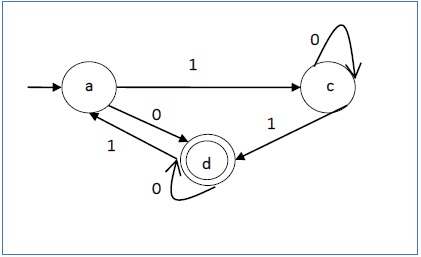
**δ\*(q0, S′) ∉ F**

The language L′ not accepted by DFA/NDFA (Complement of accepted language L) is

**{S | S ∈ ∑\* and δ\*(q0, S) ∉ F}**

**Example**

Let us consider the DFA shown in Figure 1.3. From the DFA, the acceptable strings can be derived.



Strings accepted by the above DFA: {0, 00, 11, 010, 101, ...........}

Strings not accepted by the above DFA: {1, 011, 111, ........}

# NDFA to DFA Conversion

## Problem Statement

Let **X = (Qx, ∑, δx, q0, Fx)** be an NDFA which accepts the language L(X). We have to design an equivalent DFA **Y = (Qy, ∑, δy, q0, Fy)** such that **L(Y) = L(X)**. The following procedure converts the NDFA to its equivalent DFA −

## Algorithm

**Input** − An NDFA

**Output** − An equivalent DFA

**Step 1** − Create state table from the given NDFA.

**Step 2** − Create a blank state table under possible input alphabets for the equivalent DFA.

**Step 3** − Mark the start state of the DFA by q0 (Same as the NDFA).

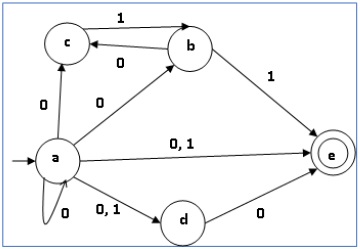
**Step 4** − Find out the combination of States {Q0, Q1,... , Qn} for each possible input alphabet.

**Step 5** − Each time we generate a new DFA state under the input alphabet columns, we have to apply step 4 again, otherwise go to step 6.

**Step 6** − The states which contain any of the final states of the NDFA are the final states of the equivalent DFA.

## Example

Let us consider the NDFA shown in the figure below.

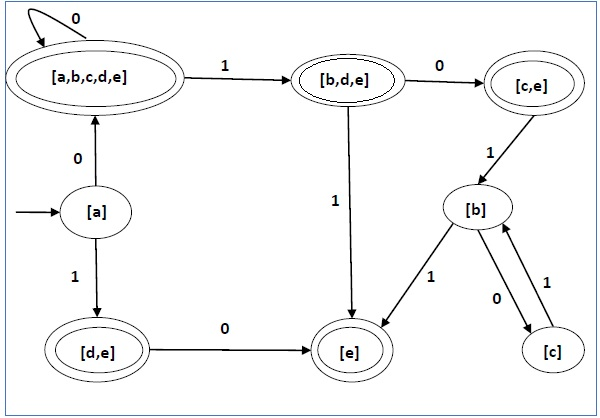


|  |  |  |
| --- | --- | --- |
| **q** | **δ(q,0)** | **δ(q,1)** |
| a | {a,b,c,d,e} | {d,e} |
| b | {c} | {e} |
| c | ∅ | {b} |
| d | {e} | ∅ |
| e | ∅ | ∅ |

Using the above algorithm, we find its equivalent DFA. The state table of the DFA is shown in below.

|  |  |  |
| --- | --- | --- |
| **q** | **δ(q,0)** | **δ(q,1)** |
| [a] | [a,b,c,d,e] | [d,e] |
| [a,b,c,d,e] | [a,b,c,d,e] | [b,d,e] |
| [d,e] | [e] | ∅ |
| [b,d,e] | [c,e] | [e] |
| [e] | ∅ | ∅ |
| [c, e] | ∅ | [b] |
| [b] | [c] | [e] |
| [c] | ∅ | [b] |

The state diagram of the DFA is as follows −



DFA Minimization:

**DFA Minimization using Equivalence Theorem**

If X and Y are two states in a DFA, we can combine these two states into {X, Y} if they are not distinguishable. Two states are distinguishable, if there is at least one string S, such that one of δ (X, S) and δ (Y, S) is accepting and another is not accepting. Hence, a DFA is minimal if and only if all the states are distinguishable.

**Algorithm 3**

**Step 1** − All the states **Q** are divided in two partitions − **final states** and **non-final states** and are denoted by **P0**. All the states in a partition are 0th equivalent. Take a counter **k** and initialize it with 0.

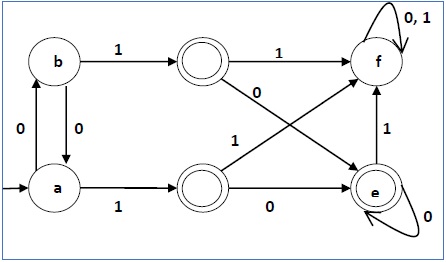
**Step 2** − Increment k by 1. For each partition in Pk, divide the states in Pk into two partitions if they are k-distinguishable. Two states within this partition X and Y are k-distinguishable if there is an input **S** such that **δ(X, S)** and **δ(Y, S)** are (k-1)-distinguishable.

**Step 3** − If Pk ≠ Pk-1, repeat Step 2, otherwise go to Step 4.

**Step 4** − Combine kth equivalent sets and make them the new states of the reduced DFA.

**Example**

Let us consider the following DFA −



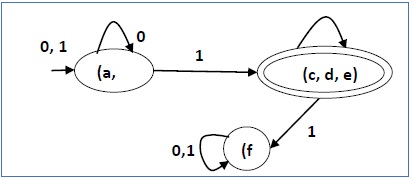
|  |  |  |
| --- | --- | --- |
| **q** | **δ(q,0)** | **δ(q,1)** |
| a | b | c |
| b | a | d |
| c | e | f |
| d | e | f |
| e | e | f |
| f | f | f |

Let us apply the above algorithm to the above DFA −

* P0 = {(c,d,e), (a,b,f)}
* P1 = {(c,d,e), (a,b),(f)}
* P2 = {(c,d,e), (a,b),(f)}

Hence, P1 = P2.

There are three states in the reduced DFA. The reduced DFA is as follows −



|  |  |  |
| --- | --- | --- |
| **Q** | **δ(q,0)** | **δ(q,1)** |
| (a, b) | (a, b) | (c,d,e) |
| (c,d,e) | (c,d,e) | (f) |
| (f) | (f) | (f) |

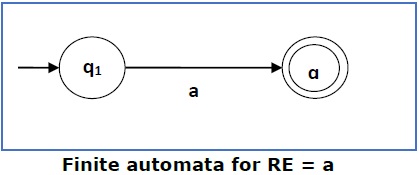
**Question: 3(b) 2015 for the given regular expression , draw its NFA and then convert NFA to the equivalent DFA using subset construction method.s**

# Construction of an FA from an RE

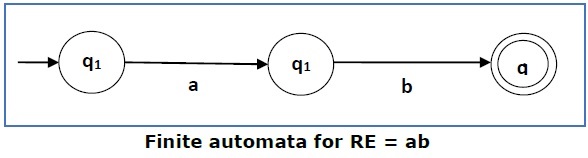
We can use Thompson's Construction to find out a Finite Automaton from a Regular Expression. We will reduce the regular expression into smallest regular expressions and converting these to NFA and finally to DFA.

Some basic RA expressions are the following −

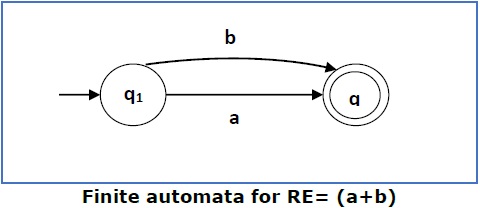
***Case 1*** − For a regular expression ‘a’, we can construct the following FA −



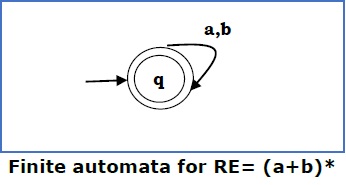
***Case 2*** − For a regular expression ‘ab’, we can construct the following FA −



***Case 3*** − For a regular expression (a+b), we can construct the following FA −



***Case 4*** − For a regular expression (a+b)\*, we can construct the following FA −



**Method**

**Step 1** Construct an NFA with Null moves from the given regular expression.

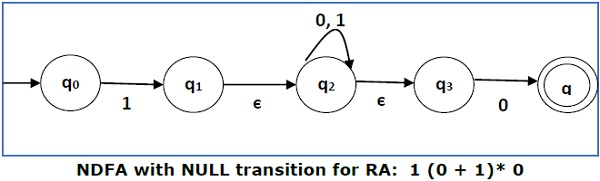
**Step 2** Remove Null transition from the NFA and convert it into its equivalent DFA.

**Problem**

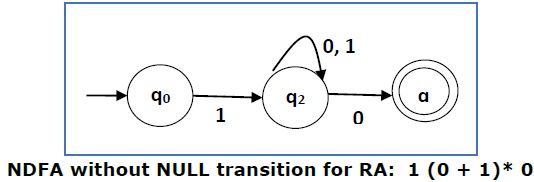
Convert the following RA into its equivalent DFA − 1 (0 + 1)\* 0

***Solution***

We will concatenate three expressions "1", "(0 + 1)\*" and "0"



Now we will remove the **ε** transitions. After we remove the **ε** transitions from the NDFA, we get the following −



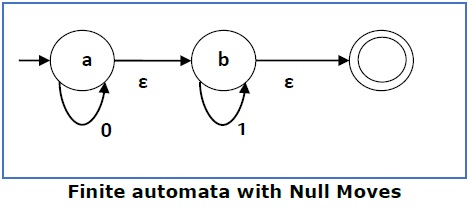
It is an NDFA corresponding to the RE − 1 (0 + 1)\* 0. If you want to convert it into a DFA, simply apply the method of converting NDFA to DFA discussed in Chapter 1.

**Finite Automata with Null Moves (NFA-ε)**

A Finite Automaton with null moves (FA-ε) does transit not only after giving input from the alphabet set but also without any input symbol. This transition without input is called a **null move**.

An NFA-ε is represented formally by a 5-tuple (Q, ∑, δ, q0, F), consisting of

* **Q** − a finite set of states
* **∑** − a finite set of input symbols
* **δ** − a transition function δ : Q × (∑ ∪ {ε}) → 2Q
* **q0** − an initial state q0 ∈ Q
* **F** − a set of final state/states of Q (F⊆Q).



The above **(FA-ε)** accepts a string set − {0, 1, 01}

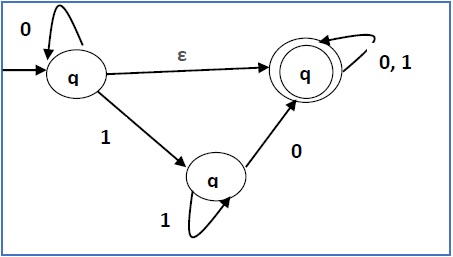
**Removal of Null Moves from Finite Automata**

If in an NDFA, there is ϵ-move between vertex X to vertex Y, we can remove it using the following steps −

* Find all the outgoing edges from Y.
* Copy all these edges starting from X without changing the edge labels.
* If X is an initial state, make Y also an initial state.
* If Y is a final state, make X also a final state.

**Problem**

Convert the following NFA-ε to NFA without Null move.



***Solution***

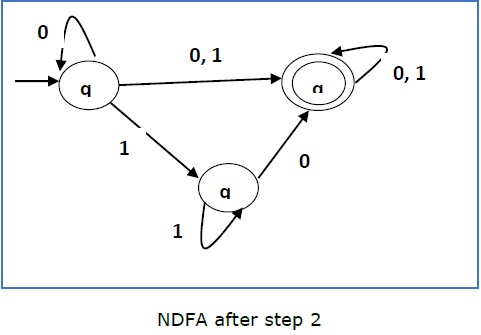
**Step 1** −

Here the ε transition is between **q1** and **q2**, so let **q1** is **X** and **qf** is **Y**.

Here the outgoing edges from qf is to qf for inputs 0 and 1.

**Step 2** −

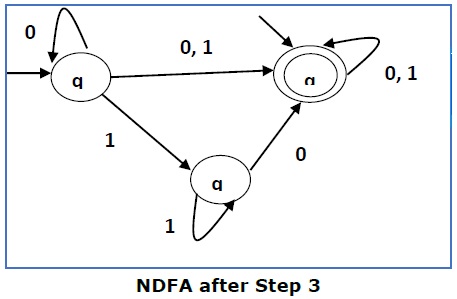
Now we will Copy all these edges from q1 without changing the edges from qf and get the following FA −



**Step 3** −

Here q1 is an initial state, so we make qf also an initial state.

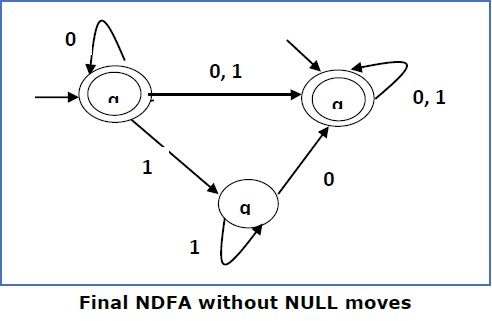
So the FA becomes −



**Step 4** −

Here qf is a final state, so we make q1 also a final state.

So the FA becomes −



**Phase-2 Syntax Analysis**

Syntax analysis or parsing is the second phase of a compiler. In this chapter, we shall learn the basic concepts used in the construction of a parser.

We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.

CFG, on the other hand, is a superset of Regular Grammar, as depicted below:



**Question 4(b):2016 Define context free grammer which describes the same language as the regular expression**

**Context Free Grammar:**

***Definition*** – A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple **(N, T, P, S)** where

* **N** is a set of non-terminal symbols.
* **T** is a set of terminals where **N ∩ T = NULL.**
* **P** is a set of rules, **P: N → (N ∪ T)\***, i.e., the left-hand side of the production rule **P** does have any right context or left context.
* **S** is the start symbol.

**Example**

* The grammar ({A}, {a, b, c}, P, A), P : A → aA, A → abc.
* The grammar ({S, a, b}, {a, b}, P, S), P: S → aSa, S → bSb, S → ε
* The grammar ({S, F}, {0, 1}, P, S), P: S → 00S | 11F, F → 00F | ε

**In Details:**

A context-free grammar has four components:

* A set of **non-terminals** (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
* A set of tokens, known as **terminal symbols** (Σ). Terminals are the basic symbols from which strings are formed.
* A set of **productions** (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a **non-terminal** called the left side of the production, an arrow, and a sequence of tokens and/or **on- terminals**, called the right side of the production.
* One of the non-terminals is designated as the start symbol (S); from where the production begins.

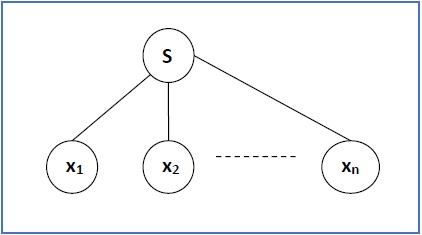
**Generation of Derivation Tree**

A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information a string derived from a context-free grammar.

**Representation Technique**

* **Root vertex** − Must be labeled by the start symbol.
* **Vertex** − Labeled by a non-terminal symbol.
* **Leaves** − Labeled by a terminal symbol or ε.

If S → x1x2 …… xn is a production rule in a CFG, then the parse tree / derivation tree will be as follows −



There are two different approaches to draw a derivation tree −

**Top-down Approach −**

* Starts with the starting symbol **S**
* Goes down to tree leaves using productions

**Bottom-up Approach −**

* Starts from tree leaves
* Proceeds upward to the root which is the starting symbol **S**

**Derivation or Yield of a Tree**

The derivation or the yield of a parse tree is the final string obtained by concatenating the labels of the leaves of the tree from left to right, ignoring the Nulls. However, if all the leaves are Null, derivation is Null.

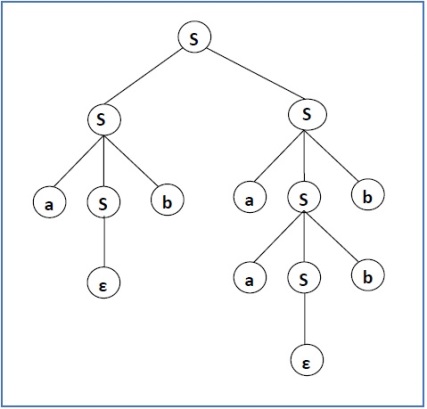
**Example**

Let a CFG {N,T,P,S} be

N = {S}, T = {a, b}, Starting symbol = S, P = S → SS | aSb | ε

One derivation from the above CFG is “abaabb”

S → SS → aSbS → abS → abaSb → abaaSbb → abaabb



**Sentential Form and Partial Derivation Tree**

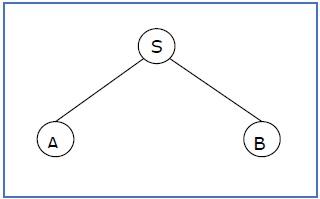
A partial derivation tree is a sub-tree of a derivation tree/parse tree such that either all of its children are in the sub-tree or none of them are in the sub-tree.

**Example**

If in any CFG the productions are −

S → AB, A → aaA | ε, B → Bb| ε

the partial derivation tree can be the following −



If a partial derivation tree contains the root S, it is called a **sentential form**. The above sub-tree is also in sentential form.

**Question: 6(a) 2015 what is the difference between left-most and right most derivation? Give example. Drive the string (id+id)/(id+id) for both the derivation methods following framer rule**

**Leftmost and Rightmost Derivation of a String**

* **Leftmost derivation** − A leftmost derivation is obtained by applying production to the leftmost variable in each step.
* **Rightmost derivation** − A rightmost derivation is obtained by applying production to the rightmost variable in each step.

**Example**

Let any set of production rules in a CFG be

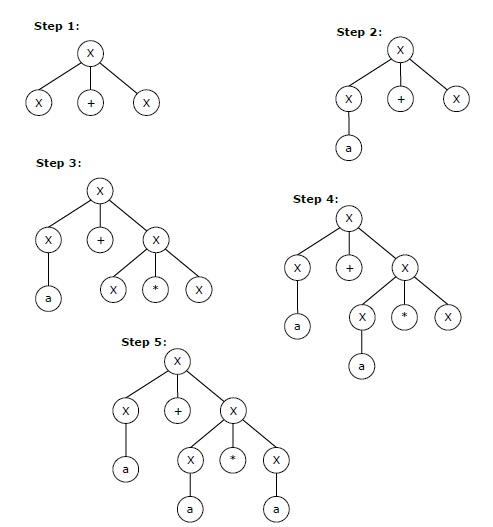
X → X+X | X\*X |X| a

over an alphabet {a}.

The leftmost derivation for the string **"a+a\*a"** may be −

X → X+X → a+X → a + X\*X → a+a\*X → a+a\*a

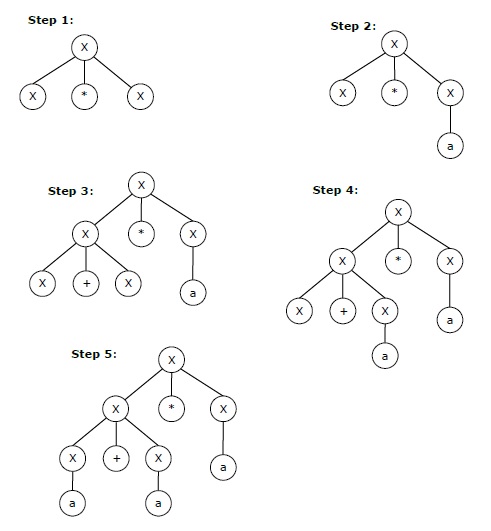
The stepwise derivation of the above string is shown as below −



The rightmost derivation for the above string **"a+a\*a"** may be −

X → X\*X → X\*a → X+X\*a → X+a\*a → a+a\*a

The stepwise derivation of the above string is shown as below −



**Left and Right Recursive Grammars**

In a context-free grammar **G**, if there is a production in the form **X → Xa** where **X** is a non-terminal and **‘a’** is a string of terminals, it is called a **left recursive production**. The grammar having a left recursive production is called a **left recursive grammar**.

And if in a context-free grammar **G**, if there is a production is in the form **X → aX** where **X** is a non-terminal and **‘a’** is a string of terminals, it is called a **right recursive production**. The grammar having a right recursive production is called a **right recursive grammar**.

**Question 7(1):2016 what do you mean by ambiguous grammer?**

**Question: 5(a) 2017 what do you mean by ambiguous grammer? Disambiguous the following CFG**

**Grammer.**

Answer:

Ambiguity in Grammer:

If a context free grammar **G** has more than one derivation tree for some string **w ∈ L(G)**, it is called an **ambiguous grammar**. There exist multiple right-most or left-most derivations for some string generated from that grammar.

## Problem

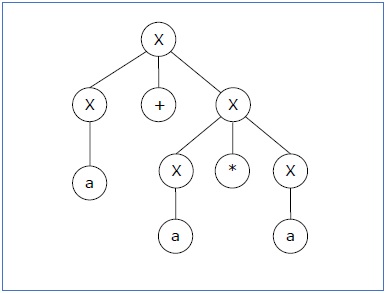
Check whether the grammar G with production rules − X → X+X | X\*X |X| a is ambiguous or not.

## Solution

Let’s find out the derivation tree for the string "a+a\*a". It has two leftmost derivations.

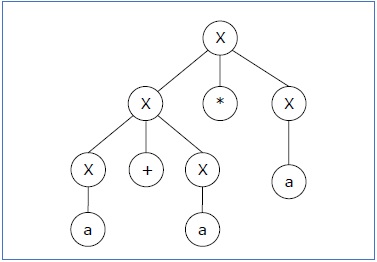
**Derivation 1** − X → X+X → a +X → a+ X\*X → a+a\*X → a+a\*a

**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.

A CFG is in Chomsky Normal Form if the Productions are in the following forms −

* A → a
* A → BC
* S → ε

where A, B, and C are non-terminals and **a** is terminal.

**Algorithm to Convert into Chomsky Normal Form −**

**Step 1** − If the start symbol **S** occurs on some right side, create a new start symbol **S’** and a new production **S’→ S**.

**Step 2** − Remove Null productions. (Using the Null production removal algorithm discussed earlier)

**Step 3** − Remove unit productions. (Using the Unit production removal algorithm discussed earlier)

**Step 4** − Replace each production **A → B1…Bn** where **n > 2** with **A → B1C** where **C → B2 …Bn**. Repeat this step for all productions having two or more symbols in the right side.

**Step 5** − If the right side of any production is in the form **A → aB** where a is a terminal and **A, B** are non-terminal, then the production is replaced by **A → XB** and **X → a**. Repeat this step for every production which is in the form **A → aB**.

**Problem**

Convert the following CFG into CNF

S → ASA | aB, A → B | S, B → b | ε

**Solution**

**(1)** Since **S** appears in R.H.S, we add a new state **S0** and **S0→S** is added to the production set and it becomes −

S0→S, S→ ASA | aB, A → B | S, B → b | ∈

**(2)** Now we will remove the null productions −

B → ∈ and A → ∈

After removing B → ε, the production set becomes −

S0→S, S→ ASA | aB | a, A → B | S | ∈, B → b

After removing A → ∈, the production set becomes −

S0→S, S→ ASA | aB | a | AS | SA | S, A → B | S, B → b

**(3)** Now we will remove the unit productions.

After removing S → S, the production set becomes −

S0→S, S→ ASA | aB | a | AS | SA, A → B | S, B → b

After removing S0→ S, the production set becomes −

S0→ ASA | aB | a | AS | SA, S→ ASA | aB | a | AS | SA

A → B | S, B → b

After removing A→ B, the production set becomes −

S0 → ASA | aB | a | AS | SA, S→ ASA | aB | a | AS | SA

A → S | b

B → b

After removing A→ S, the production set becomes −

S0 → ASA | aB | a | AS | SA, S→ ASA | aB | a | AS | SA

A → b |ASA | aB | a | AS | SA, B → b

**(4)** Now we will find out more than two variables in the R.H.S

Here, S0→ ASA, S → ASA, A→ ASA violates two Non-terminals in R.H.S.

Hence we will apply step 4 and step 5 to get the following final production set which is in CNF −

S0→ AX | aB | a | AS | SA

S→ AX | aB | a | AS | SA

A → b |AX | aB | a | AS | SA

B → b

X → SA

**(5)** We have to change the productions S0→ aB, S→ aB, A→ aB

And the final production set becomes −

S0→ AX | YB | a | AS | SA

S→ AX | YB | a | AS | SA

A → b A → b |AX | YB | a | AS | SA

B → b

X → SA

Y → a