Chapter 5

Properties of Regular Languages

Inside This Chapter

5.1. Introduction

This chapter explores the properties of regular languages. Our first tool for this exploration is a way to prove that certain languages are not regular. One important kind of fact about the regular languages is called a "closure property." These properties let us build recognize for languages that are constructed from other languages by certain operations. As an example, the intersection of two regular languages is also regular. Thus, given automata that recognize two different regular languages, we can construct mechanically or automation that recognizes exactly the intersection of these two languages.

Some other important facts about regular languages are called "decision properties." Our study of these properties gives us algorithms for answering important questions about automata.

5.2. Proving Languages not to be Regular

We have established that the class of Languages known as the regular languages for at least four different descriptions. They are the languages accepted by DFA's, NFA's, and by ϵ-NFA's; they are also the languages defined by regular expression.

Not every language is regular. In this section, we shall introduce a powerful technique, known as the "Pumping Lemma", for showing certain language not to be regular.

5.2.1. The Pumping Lemma for Regular Languages

Pumping Lemma is a powerful tool for proving certain language non-regular. It's also useful in the development of algorithm to answer certain questions concerning, finite automata, such as weather the language accepted by a given FA is finite or infinite.

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

(i) $y \neq \in$

(ii) $| xy | \leq n$

(iii) For all $i \geq 0$, the string xy^iz is also in L.

That is we always find a nonempty string y not too far from the beginning of w that can be "pumped"; that is, repeating y any number of times, keeps the resulting string in the language.

Proof. Suppose L is regular. Then $L = L(M)$ for some DFA, M. Suppose M has n states. Now, consider any string w of length n or more, say $w = a_1 a_2 \ldots a_m$, where $m \geq n$ and each a_i is an input symbol, for $i = 0, 1, 2, \ldots, m$ define state p_i to be $\hat{\delta}(q_0, a_1 a_2 \ldots a_i)$, where δ is the transition function of M, and q_0 is the start state of M. That is p_i is the state, M is in after reading the first i symbols of w.

By the pigeonhole principle, it is not possible for the $n + 1$ different p_i's for $i = 0, 1, \ldots, n$ to be distinct, since there are only n different states. Thus we can find two different integers i and j, with $0 \leq i < j \leq n$, such that $p_i = p_j$. Now we can break $w = xyz$ as follows :

1. $x = a_1 a_2 \ldots a_i$.
2. $y = a_{i+1} a_{i+2} \ldots a_j$.
3. $z = a_{j+1} a_{j+2} \ldots a_m$.

That is x takes us to p_i once; y takes us from p_i back to p_i (since p_i is also p_j), and z is balance of w. The relationship among the strings and states are discussed in Fig. 5.1.

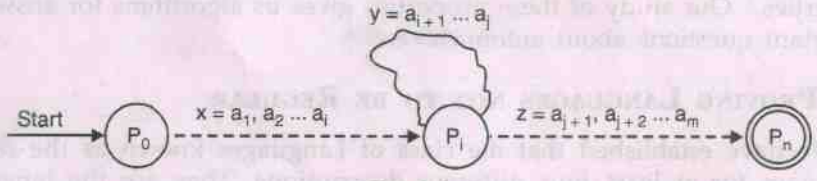


Fig. 5.1.

Note that x may be empty, in the case that $i = 0$. Also, z may be empty if $j = n = m$. However y cannot be empty, since i is strictly less than j.

Now consider what happens if the automation M receives xy^iz for any $i \geq 0$. If $i = 0$, then automation M goes from the start state q_0 to P_i on input x. Since p_i is also p_j, it must be that M goes from p_i to the accepting state in Fig. 5.1 on input z. Thus, in accepts xz.

If $i > 0$, then M goes from p_0 to p_i on input string x, circles from p_i to p_i, i times on input y_i, and then goes to the accepting state on input z. Thus, for any $i \geq 0$, xy^iz is also accepted by M; that is, xy^iz is in L.

We can also understand these transactions as follows :

For $i = 0$

$$\hat{\delta}(p_0, x) = \hat{\delta}(p_i, x) = p_i$$

$$\hat{\delta}(p_i, z) = P_n$$

For $i > 0$

$$\hat{\delta}(q_0, x) = \hat{\delta}(p_0, x) = p_i$$

$$\hat{\delta}(p_i, y) = p_i,$$

In general $$\hat{\delta}(p_i, y^i_{i\geq 0}) = p_i$$

$$\hat{\delta}(p_i, z) = p_n$$

5.2.2. Application of the Pumping Lemma

The pumping lemma is extremely useful in proving that certain sets are non-regular. The general methodology in its application is an "adversary argument" of the following :

1. Select the language L, you wish to prove non-regular.
2. The "adversary" picks n, the constant mentioned in the pumping lemma. Once the adversary has picked n, he may not change it.
3. Select a string z in L. Your choice may depend implicitly on the value of n chosen.
4. The adversary breaks z into u, v and w, subject to the constants that $| uv | \leq n$ and $| v | \geq 1$.
5. You achieve a contradiction to the pumping lemma by showing, for any u, v and w determined by the adversary that there exit i for which uv^iw is not in L. It may then concluded and L is not regular. Your selection of i may depend on, n, u, v and w.

Now let us discuss some example to see the application of pumping lemma.

Example 5.1. Prove that Language $L = \{a^nb^n$ for $n = 0, 1, 2, 3, \ldots\}$ is not regular.

Solution. Let us see how we could apply the pumping lemma directly to this case.

The pumping lemma says that there must be strings x, y and z such that all words of the form xy^nz are in L.

How do we break this into three pieces x, y and z ?

Case 1. If middle part y is made off entirely of a's, as

$$x\ aaaa\ \ldots\ z$$

How if we pump it as $xyyz$, $xyyyz$ then number of a's increases, but in language $L = \{a^nb^n$ for $n = 0, 1, 2, 3, \ldots\}$ a's and b's are equal so it is not allowed.

Case 2. If middle part y is made off entirely of b's as

x *bbbbb* ... z

For the same reason, it is also not allowed.

Case 3. y part is made of some positive number of a's and some positive number of b's. This would mean that y contain the substring ab.

x ... *aaaaabbbbb* ... z

Then $xyyz$ would have two copies of the substring ab. But every word in L contains substring ab exactly once. Therefore $xyyz$ cannot be a word in L.

This proves that the pumping lemma cannot apply to L and therefore L is not regular.

Example 5.2. Prove that $L = \{a^n b\ a^n$ for $n = 0, 1, 2, ...\}$ is not regular.

Solution. If this language were regular then there would exist three strings x, y and z such that xyz and $xyyz$ were both words in this language. We can show that this is impossible.

Observation 1. If the y string contained the b, then $xyyz$ would contain two b's which is not allowed, according to the language L.

Observation 2. If the y string is all a's than the b in the middle of the word xyz is in the x-side or z-side. In either case, $xyyz$ has increased the number of a's either infront of the b or after b, but not both.

Conclusion. Therefore, $xyyz$ does not have its b in the middle and is not in the form $a^n ba^n$. This language cannot be pumped and is therefore not regular.

Example 5.3. Prove that $L = \{a^n b^n\ ab^{n+1}$ for $n = 1, 2, 3, ...\}$ is not regular.

Solution. We are going to show that this language too is not regular by showing that if xyz is in this language for any three strings x, y and z, then $xyyz$ is not in this language :

Observation 1. For every word in this language, if we know the total number of a's, we can calculate the exact number of b's (twice the total number of a's – 1).

And conversely, if we know the total number of b's, we can uniquely calculate the number of a's (add 1 and divide by 2). So no two different words have the same number of a's and b's.

Observation 2. All words in this language have exactly two substrings two substrings equal to ab and one equal to ba.

Observation 3. If xyz and $xyyz$ are both in this language, then y cannot contain either substring ab or the substring ba because then $xyyz$ would have too many.

Conclusion 1. It must be a solid dump of a's or solid dump of b's.

Conclusion 2. If y is solid a's, then xyz and $xyyz$ are different words with same total b's, violating observation 1. If y is solid b's, then xyz and $xyyz$ are different words with the same number of a's violating observation 1.

Conclusion 3. It is impossible for both xyz and $xyyz$ to be in this language for any string x, y and z. Therefore, the language is unpumpable and not regular.

Example 5.4. Prove that language which contains set of strings of balanced parentheses is not regular.

Solution. Let us suppose language is regular, and n be a constant according to pumping lemma. We can divide each string in three parts xyz as $| xy | \leq n$ and $y \neq \in$. So xy^iz must be in L for $i \geq 0$.

Observation. It is clear that language will contains the strings like (( ) ( ) ( )), ( ) ( ) ( ), (((( )))) ....

Let us select a string $w = (((\ldots ( ) \ldots .)))$, where there are n left parenthesis by n right parenthesis. This string is in L, and is of length at least n. So we can divide w in three parts $w = xyz$ as $| xy | \leq n$ and $y \neq \in$.

Conclusion. Not let us choose $i = 0$. The resulting string xz is not in L, reason is very clear.

Since $w = xyz \; | xy | < n$.

It means y will be made of only left parenthesis. So xz is not in L since it has fewer left parenthesis then right parenthesis. This contradict the pumping lemma, so our original assumption, that L was regular, must have been incorrect.

Example 5.5. Prove that $L = \{0^n1^m2^n, n, m \geq 0\}$ is not regular.

Solution. We assume that L is regular and try to obtain a contradiction.

Observation. Suppose L is regular, let n be constant provided by the pumping lemma. Let w be the string 0^n2^n. (Here $m = 0$ which is allowed). This string is in L, and is of length at least $2n$.

So we can write $w = xyz, \; |xy| \leq n$, $y \neq \in$

Clearly xy will contain only 0's since in $w = 0^n2^n$ first n placed are occupied by the 0's only. So y will also made of 0's only.

Conclusion. According to pumping lemma xy^iz is also in L for $i \geq 0$. Let us choose $i = 0$. The resulting string xz, is not in L since it has fewer 0's then 1's. This contradicts the pumping lemma, so our original assumption that L was regular, must have been incorrect.

Example 5.6. Prove that language $L = \{0^n1^m \mid n \leq m\}$ is not regular.

Solution. Let us analyse the language first, $L = \{0^n1^m \mid n \leq m\}$ means that it contains set of strings, like 01, 011, 0111, 0011, 00111, ...

We will adopt same approach as we assume L is regular then try to obtain a contradiction to pumping lemma.

Let L be regular, and p be the constant provided by the pumping lemma. Let w be the string of L

$$w = 0^p1^p.$$

The string w is in L and is of length at least p.

Let us divide w in three parts $w = xyz$, $| xy | \leq p$ and y never empty.

Observation. Since $| xy | < p$, y consists entirely of 0's and since y is not $\in$ there is at least one 0 in y. If $xyz \in L$ then according to pumping lemma xy^kz must be in L for all $k \geq 0$.

Conclusion. So let us choose k to be 2. The resulting string $xyyz$, is not in L, since it has more 0's than 1's. This contradicts the pumping lemma, so our original assumption, that L was regular, must have been incorrect.

Example 5.7. Prove that $L = \{0^n 1^{2n} \mid n \geq 1\}$ is not regular.

Solution. Let us assume that L is regular and p be the constant provided by the pumping lemma. Let w be the string of L

$$w = 0^p 1^{2p}$$

Observation. The string w is in L, and its length at least p. So w can be written as xyz with $|\ xy\ | \leq p$ and y not $\in$ according to pumping lemma if xyz is in L then $xy^k z$ is in L for $k \geq 0$.

Conclusion. Since $|\ xy\ | \leq p$, y consists entirely of 0's, and since y is not $\in$ there is at least one 0 in y. So now let us choose k to be $2p + 1$. (Note that we are allowed to have our k depend on the p. The value p was given to us by pumping lemma.)

$$xyz \in L$$
$$\Rightarrow \quad xy^k z \in L$$
Now $$xy^{2p+1} z \notin L$$

Since the number of 0's is at least $2p + 1$ but the number of 1's is exactly $2p$.

This contradicts the pumping lemma, so our original assumption, that L was regular, must have been incorrect.

Example 5.8. Prove that language $L = \{0^n \mid n$ is a perfect square$\}$ is not regular.

Solution. Let us suppose L be regular, let p be a constant provided by the pumping lemma. Let w be the string O^{p^2}. This string is in L and is of length at least p. So w can be written as xyz with $|\ xy\ | \leq p$ and y not $\in$, and each $xy^i z$ also in L. Since $|\ xy\ | \leq p$, y consists of no more than p 0's, and since y is not $\in$ so there is atleast one '0' in y.

So now let us choose k to be 2. We claim that the resulting string $xyyz$ is not in L.

Now let us see the proof of this claim :

We want to show that the length of $xyyz$ is not a perfect square. In fact what we do is show that it must lie strictly in between 2 consecutive perfect squares, namely, p^2 and $(p-1)^2$.

So let q stand for the length of y. Then the length of $xyyz$ is $p^2 + q$. So it suffices to show that $p^2 < p^2 + q < (p+1)^2$. The first in equality holds since $q > 0$ (since y was not $\in$). For second inequality, we compute : $(p+1)^2 = p^2 + 2p + 1$, and this greater than $p^2 + q$ since $q < p$ (since $|\ xy\ | \leq p$). The claim is established, so $xyyz$ is not in L.

This contradicts the pumping lemma, so our original assumption, that L was regular, must have been incorrect.

Example 5.9. Prove that $L = \{0^n \mid n$ is perfect cube$\}$ is not regular.

Solution. We will apply same approach that is we assume that L is regular and try to obtain contradiction by applying pumping lemma.

Observation. Let us assume L is regular, let p be a constant provided by the pumping lemma. Let w be the string O^{p^3}. This string is in L, and is of length at least p. So w can be written as xyz with $| xy | \leq p$ and y not $\in$.

Conclusion. Pumping lemma say that if $xyz \in L$ then xy^kz also in L. y contains at least one 0 in y since y is not $\in$.

Let us choose $K = 2$. We claim that resulting string $xyyz$ is not a perfect cube.

Now let us see the proof for this claim :

We want to show that the length of $xyyz$ is not a perfect cube. Let q be the length of y, then the length of $xyyz$ is $p^3 + q$. So it suffices to show that $p^3 < p^3 + q < (p + 1)^3$. The first inequality holds since $q > 0$ (since y was not $\in$). For second inequality, we compute $(p + 1)^3 = p^3 + 3p^2 + 3p + 1$, and this is greater than $p^3 + q$ since $q < p$ (since $| xy | \leq p$). The claim is established, so $xyyz$ is not in L.

This contradicts the pumping lemma, so our original assumption, that L was regular, must have been incorrect.

Example 5.10. Let $\Sigma = \{a, b\}$. **Show that**

$$L = \{w \in \Sigma^* : n_a(w) < n_b(w)\}$$

is not regular.

Solution. Let us assume that L is regular and p is a constant provided by the pumping lemma. Let $w = a^p b^{p+1}$ be a string in L.

Observation. Let us write w as xyz where $| xy | \leq p$ and y not $\in$. w is in L and y consists of no more than p a's since $| xy | \leq p$. According to pumping lemma xy^kz will be in L for $k \geq 0$.

Conclusion. Let us choose $k = 2$ then $xyyz$ is not in L, since as we pump y by two then number of a's increases which violates the condition $n_a(w) < n_b(w)$.

So we got a contradiction for the pumping lemma so our assumption that L is regular must have been incorrect.

Example 5.11. Prove that set of all strings begining with a nonempty string of the form ww, is not regular. It is given that $\Sigma = \{0, 1\}$.

Solution. Let us assume that given language L is regular. Because the regular language over any alphabet are closed under intersection (we will discuss this theorem in Section 5.3).

So clearly language $L' = L \cap L$ is regular.

Now let us assume that n be constant given by the pumping lemma for language L'.

Let $$w = 10^n\ 10^n \in L'$$

By the pumping lemma, we can write $w = xyz$ such that $xy \leq n$, $y \neq \in$, and for all $k \in N$, $xy^kz \in L'$. Now let us consider xy^2z (that is choose $k = 2$).

Conclusion. If y contains a 1, then xy^2z certain more than two 1's and hence is not in L'.

If y contains no 1's then $xy^2z = 10^{n+1}10^n$ for $i > 1$, and hence is not in L (since $y \neq \in$).

In either case, we obtain a contradiction. We conclude that L is not regular.

Example 5.12. Prove that language L, which is the set of all strings beginning with a string w of length at least 3 such that $w = w^R$, is not regular. It is given that alphabet $\Sigma = \{0, 1\}$.

Solution. Let us assume L is not regular. Because the set of regular languages over any alphabet is closed under intersection, the language $L' = L \cap L$ is regular, (let say 001* 01*00).

Let n be the constant guaranteed by the pumping lemma for L'. Let $w = 001^{n+1}01^{n+1}00 \in L'$. By the pumping lemma, we can write $w = xyz$ such that $xy \leq n$, $y \neq \in$ and for all $k \in N$, $xy^kz \in L'$.

Conclusion. Because every string in L' has exactly five 0s, y can contain only 0's, for otherwise, xy^0z would not be in L'. Therefore $xy^iz = 001^{n+i+1} + 01^{n+1}$ for some $i > 0$. Clearly xy^2z contains no prefix of length at least three that is palindrome so $xy^2z \notin L'$—a contraction. Therefore, L is not regular.

Example 5.13. Prove that language $L = \{0^{2^K} \mid K \in N\}$ is not regular.

Solution. By contradiction.

Let us assume that L is regular. Let n be constant provided by the pumping lemma.

Let $w = 0^{2^i}$ for some $i \in N$ such that $2^i > n$. Clearly $w \in L$, by the pumping lemma we can write $w = xyz$ such that $| xy | \leq n$, $y \neq \in$, and for all k, $k \in N$, $xy^kz \in L$.

Clearly, $0 < | y | < 2^i$, by adding 2^i to each term, we obtained

$$2^i < | y | + 2^i < 2^{i+1}$$

Thus
$$xy^2z = 0^{2^{i+|y|}}$$
$$\notin L$$

a contradiction. Therefore L is not regular.

Example 5.14. Prove that language $L = \{0^K\}$ K is prime number $\}$ is not regular.

Solution. By contradiction. Let us assume that L is regular, let n be constant guaranteed by pumping lemma. Let $w = 0^p$ where p is some prime number so smaller then n. Clearly w is in L.

By pumping lemma we can write $w = xyz$ such that $| xy | < n$, $y \neq \in$ and for all $k \in N$ $xy^kz \in L$. Then

$$xy^{p+1}z = 0^{p+p|y|}$$
$$= 0^{P(1+|y|)}$$
$$\notin L \text{ because } | y | > 0$$

a contradiction. Therefore, L is not regular.

Example 5.15. Prove that $L = \{x \in (0, 1)^* \mid x = x^R\}$ is not regular.

Solution. We will apply same approach of proving by contradiction.

Let us assume that L is regular and n be a constant guaranteed by pumping lemma.

Let $w = 0^n\ 10^n \in L$.

By the pumping lemma we can write

$w = xyz$ such that $|\ xy\ | \leq n$, $y \notin \in$ and for all $k \in N$ $xy^kz \in L$.

Because $|\ xy\ | \leq n$ we have

$$xy^2z = 0^{n+|\ y\ |}10^n \quad \text{For } k = 2$$

$$\notin L \text{ because } |\ y\ | > 0$$

So it is contradiction, hence L is not regular.

5.3. Closure Properties of Regular Languages

In this section, we shall prove several theorems of the form "if certain languages are regular, and a language L is formed from them by certain operations, then L is also regular. These theorems are often called closure properties of the regular languages, since they show that the class of regular languages is closed under the operation mentioned. Here is a summary of the principle closure properties of regular languages.

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

Theorem 5.1. If L_1 and L_2 are regular languages, then so are $L_1 \cup L_2$, L_1L_2 and L_1^*.

We can say that family of regular language is closed under union, concatenation, and star-closure.

Proof. If L_1 and L_2 are regular, then there exist regular expression r_1 and r_2 such that $L_1 = L(r_1)$ and $L_2 = L(r_2)$.

By the definition, $r_1 + r_2$, r_1r_2 and r_1^* are regular expression denoting the languages $L_1 \cup L_2$, L_1L_2, and L_1^*, respectively. Thus, closure under union, concatenation and star-closure is immediate.

Theorem 5.2. If L is regular then complement of L that is $\overline{L}$ is also regular. In other words, the set of regular languages is closed under complementation.

Proof. To show closure under complementation, let $M = (Q, \Sigma, \delta, q_0, F)$ be a *dfa* that accepts L, then *dfa*

$$\overline{M} = (Q, \Sigma, \delta, q_0, Q - F)$$

accepts $\overline{L}$.

We can justify it as if M accepts the language L then some of the states of this FA are final states and, must likely, some are not. Let us reverse the final

status of each state, that is if it was a final state, make it a non-final state, and if it was a non-final state, make it a final state. If an input string formally ended in a non-final state, it now ends in a final state and vice versa. This new machine we have built accepts all input strings that were not accepted by the original FA (all the words in $\bar{L}$) and rejects all the input strings that the FA used to accept (the words in L). Therefore in machine accepts exactly the language $\bar{L}$. So $\bar{L}$ is also regular language.

Example 5.16. An FA that accepts only the strings *aba* and *abb* is shown adjoining figure, find FA for the complement of L *i.e.* accepted every string other then *aba* and *abb*.

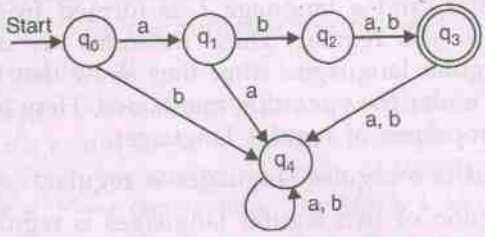


Fig. 5.2.

Solution. Apply Theorem 5.2. to find out the required FA which accepts strings other then *aba* and *abb* that is every non-final state will be final state and every final state will be non-final.

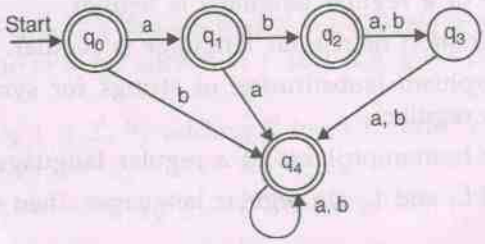


Fig. 5.3. DFA for the complements of *dfa* in figure 5.2.

Theorem 5.3. If L_1 and L_2 are regular languages, then $L_1 \cap L_2$ is also regular language. In other words, the set of the regular language is closed under intersection.

Proof. By Demorgon's law for sets of any kind (regular languages or not):

$$L_1 \cap L_2 = \overline{(\bar{L}_1 + \bar{L}_2)}$$

This can be justify by the Venn diagrams as follows :

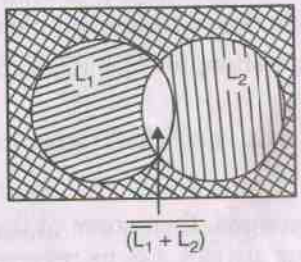


Fig. 5.4.

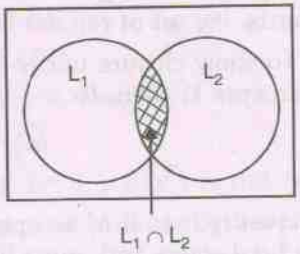


Fig. 5.5

So it is clear from Fig. 5.4 and Fig. 5.5 that

$$L_1 \cap L_2 = \overline{(\overline{L_1} + \overline{L_2})}$$

Because L_1 and L_2 are regular so $\overline{L_1}$ and $\overline{L_2}$ are regular (from Theorem 5.2). So is $\overline{L_1} + \overline{L_2}$ (by Theorem 5.1). And because $\overline{L_1} + \overline{L_2}$ is regular then so is $\overline{(\overline{L_1} + \overline{L_2})}$ which means $L_1 \cap L_2$ is regular.

Theorem 5.4. If L and m are regular languages, then so is $L - m$.

Proof. Observe that $L - m = L \cap \overline{m}$. Here $\overline{m}$ is also regular since m is regular (according to Theorem 5.2) and $L \cap \overline{m}$ is regular according to the Theorem 5.3. Therefore $L - m$ is regular.

5.3.1. Reversal

The reversal of a string $a_1\, a_2\, a_3 \ldots a_n$ is the string written backwards, that is $a^n\, a^{n-1} \ldots a_1$. We use w^R for the reversal of string w. Thus, 1101^R is 1011 and $\epsilon^R = \epsilon$.

TIPS

The reversal of a language L, written L^R, is the language consisting of the reversal of all its strings. For instance, if L = {001, 10, 111}, then L^R = {100, 01, 111}. Reversal is another operation that preserve regular languages; that is if L is regular language, so is L^R. There are two simple proofs, one based on automata and one based on regular expression.

Given a language L that is $L(A)$ for some finite automation, perhaps with non-determinism and ϵ-transitions, we may construct automation for L^R by :

1. Reverse all the arcs in the transition diagram for A.
2. Make the start of A be the only accepting state for the new automation.
3. Create a new start state p_0 with transition on ϵ to all the accepting states of A.

The result is an automation that simulates A "in reverse", and therefore accepts a string w if and only if A accepts w^R.

Theorem 5.5. If L is a regular language, so is L^R.

Proof. Let us assume that L is defined by the regular expression r. The proof is a structural induction on the size of r. We show that there is another regular expression r^R such that $L(r^R) = (L(r))^R$; that is the language of r^R is the reversal of the language of r.

Basis. If r is ϵ, ϕ, or a, for some symbol a, then r^R is the same as r. That is, we know $\{\epsilon\}^R = \{\epsilon\}$, $\phi^R = \phi$ and $\{a\}^R = \{a\}$.

Induction. There are three cases, depending on the form of r.

1. $r = r_1 + r_2$. Then $r^R = r_1^R + r_2^R$. The justification is that the reversal of the union of two languages is obtained by computing the reversal of the two languages and taking the union of those languages.
2. $r = r_1 \,.\, r_2$. Then $r^R = r_2^R r_1^R$. Note that we reverse the order of the two languages, as reversing the language themselves.
3. $r = r_1^*$. Then $r^R = (r_1^R)^*$. The justification is that any string w in $L(r)$ can be written as $w_1, w_2, \ldots, w_n$, where each w_i is in $L(r)$. But

$$w^R = w_n^R w_{n-1}^R \ldots w_1^R.$$

Each w_i^R is in $L(r)^R$, so w^R is in $(r_1^R)^*$.

Example 5.17. Let L be defined by the regular expression $(0 + 1)0^*$, then find L^R.

Solution. $L = (0 + 1)\ 0^*$

$L^R = (0^*)^R\ (0 + 1)^R$, by the rule for the concatenation.

5.3.2. Homomorphism

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

Suppose Σ and Σ' are alphabets. Then function

$$h : \Sigma \longrightarrow \Sigma'$$

is called a "homomorphism".

In other words, a homomorphism is a substitution in which a single letter is replaced with a string. The domain of the function h is extended to strings in an obvious fashion, if

$$w = x_1 x_2 \ldots x_n,$$

then $$h(w) = h(x_1)\ h(x_2)\ h(x_3) \ldots h(x_n)$$

If L is a language on Σ, then its homomorphic image is defined as

$$h(L) = \{h(w) : w \in L\}.$$

Example 5.18. Let $\Sigma = \{0, 1\}$ and $\Sigma' = \{0, 1, 2\}$ and defined n by

$$h(0) = 01$$
$$h(1) = 112$$

Find $h(010)$ and homomorphic image of $L = \{00, 010\}$.

Solution. Given $\Sigma = \{0, 1\}$, $\Sigma' = \{0, 1, 2\}$

h is defined as : $h(0) = 01$

$h(1) = 112$

So $h(010) = 0111201$.

The homomorphic image of $L = \{00, 010\}$ is the language $h(L) = \{0101, 0111201\}$.

If we have a regular expression r for a language L, then a regular expression for $h(L)$ can be obtained by simply applying the homomorphism to each Σ symbol of r.

Example 5.19. $\Sigma = \{0, 1\}$, and $\Sigma' = \{1, 2, 3\}$. Define h by

$$h(0) = 3122$$
$$h(1)\ 132$$

if L is regular language denoted by

$$r = (0 + 1^*)\ (00)^*$$

the find out the regular expression for language $h(L)$.

Solution. Given $\Sigma = \{0, 1\}$

$$\Sigma' = \{1, 2, 3\}$$

and h is defined as $h(0) = 3122$

$$h(1) = 132$$

$r = (0 + 1^*)\ (00)^*$; where r is regular expression for language L.

Let regular expression for the language $h(L)$ be r', then

$$r' = (h(0) + h(1)^*)\ (h(0)\ h(0))^*$$

Let us put $h(0)$ and $h(1)$ in r'.

$$r' = (3122 + (132)^*)\ (3122\ 3122)^*$$

denotes then regular language $h(L)$.

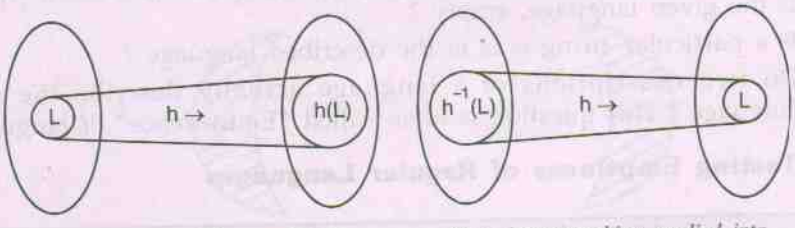
Theorem 5.6. Let h be a homomorphism. If L is a regular, then its homomorphic image $h(L)$ is also regular. The family of regular languages is therefore closed under arbitrary homomorphisms.

Proof. Let L be regular language denoted by some regular expression r. We find $h(r)$ by substituting $h(a)$ for each symbol $a \in \Sigma$ or r. It can be shown directly by an appeal to the definition of a regular expression that the result is a regular expression. It is equally easy to see that expression denotes $h(L)$. All we need to do is to show that for every $w \in L(r)$. The corresponding $h(w)$ is in $L(h(r))$ and conversely that for every x in $L(h(r))$ there is a w in L, such that $x = h(w)$. Leaving the detail as an exercise, we claim that $h(L)$ is regular.

5.3.3. Inverse Homomorphism

Definition Homomorphism may also be applied "backwards," and in this made they also preserve regular language. That is, suppose h is a homomorphism from some alphabet Σ to strings in another (possible the same) alphabet Σ'. Let L be a language over alphabet Σ'. Then $h^{-1}(L)$, read "h inverse of L," is the set of strings w in Σ^* such that $h(w)$ is in L.

Figure 5.6 suggests the effect of a homomorphism on a language L in part (a), and the effect of an inverse homomorphism in part (b).



(a) A homomorphism applied in forward direction.

(b) A homomorphism applied into inverse direction.

Fig. 5.6.

Theorem 5.7. If n is a homomorphism from alphabet Σ to alphabet Σ', and L is a regular over Σ', then $h^{-1}(L)$ is also regular language.

Proof. The proof starts with a DFA A for L. We construct from A and h a DFA for $h^{-1}(L)$ using the plan suggested by Fig. 5.7. This DFA uses the states of A but translates the input symbol according to h before deciding on the next state.

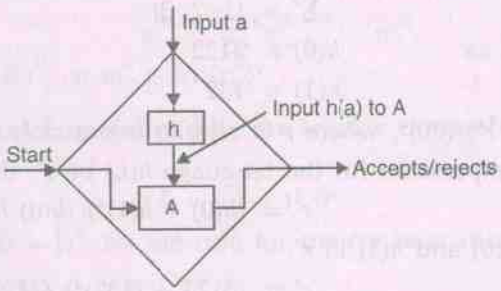


Fig. 5.7.

The DFA for $h^{-1}(L)$ applies to its input, and then simulates the DFA for L.

Formally, let L be $L(A)$, where DFA $A = (Q, \Sigma', \delta, q_0, F)$. Define a DFA $B = (Q, \Sigma, \delta', q_0, F)$, where transition function δ' is constructed by the rule $\delta'(q, a) = \hat{\delta}(q, h(a))$. That is, the transition B makes on input a is the result of the sequence of transitions that a makes on the string of symbols $h(a)$. Remember that $h(a)$ could be ϵ, it could be one symbol, or it could be many symbols, but δ is properly defined to take care of all these cases.

It is an easy induction on $| w |$ to show that $\hat{\delta}'(q_0, w) = \hat{\delta}(q_0, h(w))$. Since the accepting states of A and B are the same, B accepts w if and only if A accepts $h(w)$. Put another way, B accepts exactly those strings what are in $h^{-1}(L)$.

5.4. Decision Properties of Regular Languages

In this section we consider how one answers important questions about regular languages. First, we must consider what it means to ask a question about a language. The typical language is infinite, so you cannot present the strings of the language to someone and ask a question that requires them to inspect the infinite set of strings. Rather, we present a language by giving one of the finite representations for it that we have developed : a DFA, an NFA, an ϵ-NFA or a regular expression.

Of course the language so described will be regular and in fact there is no way at all to represent completely arbitrary languages. Let us consider some of the fundamental questions about languages :

1. Is the given language, empty ?
2. Is a particular string w is in the described language ?
3. Do two descriptions of a language actually describe the same language ? This question is often called "Equivalence" of languages.

5.4.1. Testing Emptiness of Regular Languages

TIPS

If our representation is any kind of finite automation, the emptiness question is whether there is any path whatsoever from the start state to some accepting state, if so, language is nonempty, while if the accepting states are all separated from the start state, then the language is empty.

The start state is surely reachable from the start state, if there is an arc from q to p with any label (an input symbol, or ϵ if the automation is an ϵ-NFA) then p is reachable. In that manner we can compute the set of reachable

states. If any accepting state is among them, we answer "no" (the language of the automation is not empty), and otherwise we answer "yes". Note that the reachability calculation takes no more time that $0(n^2)$ if the automation has n states, and in fact it is no worse than proportional to the number of arcs in the automation is transition diagram, which could be less than n^2 and cannot be more than $0(n^2)$.

If we are given a regular expression representing the language L, rather than an automation, we could convert the expression to an ϵ-NFA and proceed as above. Since the automation that results from a regular expression of length n has at most $0(n)$ states and transitions, the algorithm takes $0(n)$ time.

5.4.2. Testing Membership in a Regular Language

The next question of importance is, given a string w and a regular language L, is w in L. While w is represented explicitly, L is represented by an automation or regular expression.

If L is represented by a DFA, the algorithm is simple. Simulate the DFA processing the string of input symbol w, beginining in the start state. If the DFA ends in an accepting state, the answer is "yes"; otherwise the answer is "no". If $| w | = n$, and the DFA is represented by a suitable data structure, such as two-dimensional array that is the transition table, then each transition requires constant time, and the entire test takes $0(n)$ time.

Theorem 5.8. Whether two regular sets are identical is solvable.

Proof. Let us take two finite automate (M_1 and M_2) and examine a picture of the sets they accept.

If the intersection is the same as both sets then indeed they are identical. Or, on the other hand, if the areas outside the intersection are empty then both sets ($s(M_1)$ and $s(M_2)$) are identical.

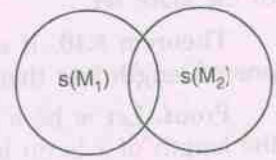


Fig. 5.8.

Let us examine these outside areas.

The picture in Fig. 5.9 (a) represents the set accepted by M_1 and rejected by M_2 while that in Fig. 5.9 (b) is the set which M_2 accepts and M_1 rejects.

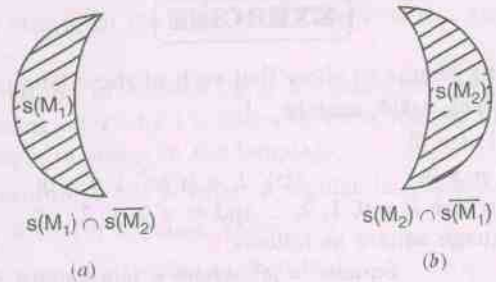


Fig. 5.9.

If these two areas (or sets) are empty then sets accepted by M_1 and M_2 are exactly the same. This means that the equivalence problem for $s(M_1)$ and $s(M_2)$ is exactly same as the emptiness problem for:

$$\left| s(M_1) \cap s(\bar{M}_2) \right| \cup \left| s(M_2) \cap s(\bar{M}_1) \right|$$

So, if we can solve the emptiness problem for the above set, then we can solve the equivalence problem for $s(M_1)$ and $s(M_2)$. Since the regular sets are closed under union, complement and intersection; the above set is a regular set. And we know that emptiness for the class of regular sets is solvable. Hence it proved.

Theorem 5.9. Whether a regular set is finite is solvable.

Proof. We know that if a finite automation accepts any strings at all then some will be of length less than the size of the state set. (This does not help directly but it gives a hint as to what we need for this theorem.)

If we were to find a string accepted by finite automation m which was longer than or equal to the size of it's state set, we could use the inflation aspect of the pumping lemma to show that machine m must accept on infinite number of strings. This means that :

"a finite automation accepts only strings of length less than the size of its set of states, if and only if it accepts a finite set."

Thus, to solve the finiteness problem for $M = (Q, \Sigma, \delta, q_0, F)$, we need to determine whether or not :

$L(M) - \mid$ strings of length $< \mid Q \mid \mid = \phi$

A question now arises as to how many inputs we must examine in order to tell it M will accept on input longer than or equal to the size of its state set. The answer is that we only need to consider input strings upto twice the size of the state set.

Theorem 5.10. If a finite automation accepts any strings, it will accepts one of length less than the size of its state set.

Proof. Let m be a finite automation which accepts the string x and that the length of x is no less than the size of M's state set. Assume further that M accepts no string shorter than x. This is the opposite of our theorem.)

Immediately the pumping lemma asserts that there are strings u, v and w such that $uvw = x$, $v \neq \in$ and $uw \in L(M)$. Since $v \neq \in$, uw is shorter than $uvw = x$. Thus M accepts shorter strings than x and the theorem follows :

EXERCISE

1. Use the pumping lemma to show that each of these languages is non-regular :
   (*i*) $\{a^n\ b^{n+1}\} = \{abb, aabbb, aaabbbb \ldots\}$
   (*ii*) $L = \{a^n b^n a^n \mid n > 0\}$
   (*iii*) $L = \{a^n b^{2n} \mid n > 0\}$ (*iv*) $L = \{a^n b a^n \mid n > 0\}$
   (*v*) $L = \{a^n b^n a^m$ where $n = 0, 1, 2, \ldots$ and $m = 0, 1, 2, \ldots\}$
2. Define the language square as follows :
   Square $= \{a^n$ where n is a square, $n > 0\}$
   $= \{a, aaaa, aaaaaaaaa \ldots\}$
   Using the pumping lemma to prove that square is non-regular.
3. Prove that $L = \{a^n b^n$ where n is square and $n > 0\}$ is non-regular.
4. Define the language double prime as follows
   Double prime $= \{a^p b^p$ where p is any prime$\}$
   prove that double prime is non-regular.

5. Define the language double factorial as follows :

$$DF = \{a^{n!}\, b^{n!} \mid n > 0\}$$

prove that DF is non-regular.

6. Let L_1, L_2, L_3 ... be an infinite sequence of regular languages.
   (*i*) Let L be the infinite union of all these language taken together. Is L necessarily regular ?
   (*ii*) Is the infinite intersection of all these languages necessarily regular ?

7. Consider the following language :

$$\bar{p} = \{a^n \text{ where } n \text{ is not prime}\}$$
$$= \{\in, a, aaaa, aaaaaa, aaaaaaaaa, ...\}$$

prove that $\bar{p}$ is non-regular.

8. Consider the language L defined as

$$L = \{a^n \text{ where } n \text{ is any integer with an even number of digits in base 10}\}$$
$$= \{\in, a^{10}, a^{11}, a^{12}, ...\}$$

prove that L is non-regular.

9. (*i*) Show that if we add a finite set of words to a regular language, the result is a regular language.
   (*ii*) Show that if we subtract a finite set of words from a regular language, the result is a regular language.

10. Prove that following languages are not regular.
    (*i*) $L = \{a^n b^j a^k : k \geq n + 1\}$ (*ii*) $L = \{a^i b^j a^k : k \neq i + j\}$
    (*iii*) $L = \{a^i b^j a^k : i = j \text{ or } j = k\}$ (*iv*) $L = \{a^i b^j : i \leq 1\}$
    (*v*) $L = \{w : n_a(w) \neq n_b(w)\}$

11. Prove that following language regular :
    (*i*) $L = \{xyy^R z : x, z, y \in \{0, 1\}^+\}$
    (*ii*) $L = \{xyy^R z : x, z, y \in \{0, 1\}^+, \mid x \mid \geq \mid z \mid\}$

12. Suppose that we know that $L_1 \cup L_2$ and L_1 are regular. Can we conclude from this that L_2 is regular.

13. Prove that following languages over alphabet {0, 1} are not regular.
    (*i*) The set of strings of 0's and 1's, beginning with a 1, such that when interpreted as an integer, that integer is prime.
    (*ii*) The set of strings of the form $w\,1n$, where w is a string of 0's and 1's of length n.
    (*iii*) The set of strings of 0's and 1's of the form $w\bar{w}$, where $\bar{w}$ is formed from w by replacing all 0's by 1's, and vice-versa that is, $\overline{011} = 100$, and 011 100 is an example of string in the language.

14. Given an algorithm to tell whether a regular language L is infinite.

15. Let A be any FA with N states. Then :
    (*i*) If A accepts an input string w such that
    $N \leq$ length $(w) < 2N$, then A accepts an infinte language.
    (*ii*) If A accepts many words, then A concepts some word w such that $N \leq$ length $(w) < 2N$.

*****

Chapter 6

Context-Free Grammars and Languages

Inside This Chapter

6.1. GRAMMARS

All of us know that a grammar is nothing but a set of rules to define valid sentences in any languages. In this chapter we introduce the context-free grammars, which generates context-free languages. Context-free languages have great practical significance in defining programming languages and in simplifying the translation for programming languages.

Initially Linguists were trying to define precisely valid sentences and to give structural description for these sentences. They tried to define rules for natural languages like Hindi, English etc. Noam Chomsky gave a mathematical model for the grammars in 1956. Although it was useless to describe natural languages but it becomes very useful for computer languages.

The original motivation for grammars was the description of natural languages. We can write rules for the grammar of natural languages as follows :

<sentence> → <noun phrase> <verb phrase>

<noun phrase> → <article> <noun>

According to above set of rules "The girl eats" is valid sentence.

TIPS

It can be easily observed that, some words in the grammar works as terminator for the sentence, these symbols are called terminal symbol. Rest of the symbols of the vocabulary are called non-terminals. So it is clear that terminals and non-terminals forms vocabulary for any language.

Let us now formalise the idea of a grammar and how it is used. There are four important components in a grammatical description of a language :

1. There is a finite set of symbols that form the strings of the language being defined. We call these alphabets terminal symbols, represented by V_t.
2. There is a finite set of variables, also called sometimes non-terminals or syntactic categories. Each variable represents a language; *i.e.*, a set of strings, represented by V_n.
3. One of the variable represents the language being defined; it is called the start symbol. Generally it is denoted by S.
4. There is a finite set of rules or productions that represents the recursive definition of a langauge. Each production consists of :
   (*a*) A variable that is being (partially) defined by the production. This variable is often called the head of the production.
   (*b*) The production symbol $\rightarrow$
   (*c*) A string of zero or more terminals and variables. This string, called the body of the production, represents one way to form strings in the language of the variable of the head. So, we leave terminals unchanged and substitute for each variable of the body any string that is known to be in the language of that variable.

After this discussion, we are able to define context free grammar.

6.2. Context-Free Grammars

Now let us formalize the concept of context free grammar, as we discussed intuitive notions for context free grammar in previous section.

Definition Mathmatically context-free grammar is defined as follows :

"A grammar $G = (V_n, V_t, P, S)$ is said to be context-free"

where V_n : A finite set of non-terminals, generally represented by capital letters, $A, B, C, D, \ldots$

V_t : A finite set of terminals, generally represented by small letters, like, $a, b, c, d, e, f, \ldots$

S : Starting non-terminal, called start symbol of the grammar. S belongs to V_n.

P : Set of rules or productions in CFG.

G is context-free and all productions in P have the form,

$$\alpha \rightarrow \beta$$

where $\alpha \in V_n$ and $\beta \in (V_t \cup V_n)^*$

Every regular grammar is context-free, so a regular language is also a context-free one. It is already proved by pumping lemma that language $\{a^n b^n / n > 0\}$ is not regular, but it is possible to design a context-free-grammar for these languages (we will design it in next section). **So now it is very clear that "the family of regular language is a proper subset of the family of context-free language".**

TIPS

Context-free grammars derive their name from the fact that the substitution of the variable on the left of a production can be made any time, such a variable appears in the sentential form. It does not depend on the symbols in the rest of the sentential form (the context). This feature is the consequence of allowing only a single variable on the left side of the production.

6.2.1. Derivations

We now define the notations to represent a derivation. First we define two notations $\underset{G}{\Rightarrow}$ and $\overset{*}{\underset{G}{\Rightarrow}}$. If $\alpha \to \beta$ is a production of P in CFG and a and b are strings in $(V_n \cup V_t)^*$, then

$$a\ \alpha\ b \underset{G}{\Rightarrow} a\ \beta\ b.$$

We say that the production $\alpha \to \beta$ is applied to the string $a\ \alpha\ b$ to obtain $a\ \beta\ b$ or we say that $a\ \alpha\ b$ directly drives $a\ \beta\ b$.

Now suppose $\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_m$ are string in $(V_n \cup V_t)^*$,

$$m \geq 1 \text{ and } \alpha_1 \underset{G}{\Rightarrow} \alpha_2, \alpha_2 \underset{G}{\Rightarrow} \alpha_3, \alpha_3 \underset{G}{\Rightarrow} \alpha_4, \ldots, \alpha_{m-1} \underset{G}{\Rightarrow} \alpha_m$$

Then we say that $\alpha_1 \overset{*}{\underset{G}{\Rightarrow}} \alpha_m$, *i.e.*, we say α_1, drives α_m in grammar G. If α drives β by exactly i steps, we say $\alpha \overset{i}{\underset{G}{\Rightarrow}} \beta$.

6.2.2. Language of Context-Free Grammar

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

$$L(G) = \left\{ w \text{ in } V_t \,/\, S \overset{*}{\underset{G}{\Rightarrow}} w \right\}$$

Definition If a language L is the language of some context-free grammar, then L is said to be context-free language, or CFL.

6.2.3. Sentential Forms

Derivations from the start symbol produce strings that have a special rule. We call these "Sentential forms". That is, if $G = (V_n, V_t, P, S)$ is a CFG, then any string α in $(V_n \cup V_t)^*$ such that $S \overset{*}{\Rightarrow} \alpha$ is a sentential form.

Note that the language $L(G)$ is those sentential forms that are in V_t^*, *i.e.*, they consists solely of terminals.

Now let us discuss some examples.

Example 6.1. Consider a grammar $G = (V_n, V_t, P, S)$ where $V_n = \{S\}$, $V_t = \{a, b\}$ and set of productions P is given by

$$P = \{S \to aSb$$
$$S \to ab$$
$$\}.$$

Here S is the only non-terminal which is the starting symbol for the grammar; 'a' and 'b' are terminals. There are two productions $S \to aSb$ and $S \to ab$. Now we will show how the strings a^2b^2 can be derived.

$$\begin{aligned} S &\Rightarrow aSb \\ &\Rightarrow aabb \\ &\Rightarrow a^2b^2 \end{aligned}$$

Here we need to apply the first production then second production.

By applying first production $n - 1$ times, followed by an application of second production, we get

$$\begin{aligned} S &\Rightarrow aSb \\ &\Rightarrow aaSbb \\ &\Rightarrow : \\ &\quad : \\ &\Rightarrow a^{n-1}Sb^{n-1} \\ &\Rightarrow a^nb^n \end{aligned}$$

Hence we can say that language for the above grammar is

$$L(G) = \{a^n\, b^n / \; n >= 1\}.$$

Example 6.2. Following is a CFG for the language

$$L = \{wcw^R / w \in (a, b)^*\}$$

Solution. Let G be CFG for language $L = \{wcw^R / w \in (a, b)^*\}$

$$G = (V_n, V_t, P, S)$$

Here

$$V_n = \{S\}$$

$$V_t = \{a, b, c\}$$

and P is given by

$$\begin{aligned} &S \to aSa \\ &S \to bSb \\ &S \to c \end{aligned}$$

Let us check that *abbcbba* can be drived from the given CFG.

$$\begin{aligned} S &\Rightarrow aSa && \text{(use the } S \to aSa\text{)} \\ &\Rightarrow abSba && \text{(use the } S \to bSb\text{)} \\ &\Rightarrow abbSbba && \text{(use the } S \to bSb\text{)} \\ &\Rightarrow abbcbba && \text{(use the } S \to c\text{)} \end{aligned}$$

So string *abbcbba* can be derived from given CFG.

6.2.4. Bacus Naur Form (BNF)

While linguists were studying CFG's computer scientists began to describe programming languages by notation called Bacus-Naur form or Bacus Normal Form, which is the CFG notation with minor changes in format and some shorthand. For example, the above grammar can be rewritten in BNF as

$$\begin{aligned} S &\to bA / aB \\ A &\to bAA / aS / a \\ B &\to aBB / bS / a \end{aligned}$$

Hence BNF is a shorthand notation for context free grammar.

Example 6.3. Write a CFG, which generates string of balanced parenthesis.

Solution. This grammar will accept the balanced right and left parenthesis. For example ( ) ( ) is acceptable, ((( ))) is also accepted.

Let us design the CFG for this

Let CFG be $G = (V_n, V_t, P, S)$ where

V_n = set of non-terminals = {S},

V_t = set of terminals = {(,)}

and set of production P is given by

$S \rightarrow SS$

$S \rightarrow (S)$

$S \rightarrow \in$

Now we see what this grammar generates :

$S \Rightarrow SS$

$\Rightarrow (S)\ S$

$\Rightarrow (S)\ SS$

$\Rightarrow (S)\ (S)\ S$

$\Rightarrow (S)\ (S)\ (S)$

$\Rightarrow (\ )\ (S)\ (S)$

$\Rightarrow (\ )\ (\ )\ (S)$

$\Rightarrow (\ )\ (\ )\ (\ )$

Thus the above grammar always generates balanced pairs of parenthesis.

Example 6.4. Write a CFG, which generates palindrome for binary numbers.

Solution. Grammar will generate palindrome for binary numbers, that is 00, 010, 11, 101, 11100111 ...

Let CFG be $G = (V_n, V_t, P, S)$

where V_n = set of non-terminal = {S}

V_t = set of terminals = {0, 1}

and production rule P is defined as

$S \rightarrow 0S0/1S1$

$S \rightarrow 0/1/\in$

Obviously this grammar generates palindrome for binary numbers, it can be seen by the following derivation

$S \Rightarrow 0S0$

$\Rightarrow 01S10$

$\Rightarrow 010S010$

$\Rightarrow 0101010$

which is a palindrome.

Example 6.5. Write a CFG for the regular expression

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

Solution. Let us analyse regular expression

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

Clearly regular expression is the set of strings which starts with any number of 0's, followed by a one and end with any combination of 0's and 1's.

Let CFG be $G = (V_n, V_t, P, S)$

where $V_n = \{S, A, B\}$

$V_t = \{0, 1\}$

and productions P are defined as

$$S \rightarrow A1B$$
$$A \rightarrow 0A/\in$$
$$B \rightarrow 0B/1B/\in$$

Let us see the derivation of the string 00101

$$S \Rightarrow A1B$$
$$\Rightarrow 0A10B$$
$$\Rightarrow 00A101B$$
$$\Rightarrow 00101$$

So clearly G is CFG for regular expression r.

Example 6.6. Write a CFG which generates strings having equal number of a's an b's.

Solution. Let CFG be $G = (V_n, V_t, P, S)$

$V_n = \{S\}$

$V_t = \{a, b\}$

where P is defined as

$$S \rightarrow aSbS/bSaS/\in$$

Let us derive a string

w = bbabaabbaa

$$S \Rightarrow b\underline{S}aS$$
$$\Rightarrow bb\underline{S}aSaS$$
$$\Rightarrow bbaSb\underline{S}aSaS$$
$$\Rightarrow bbaSba\underline{S}bSaSaS$$
$$\Rightarrow bba\underline{S}baaSbSbSaSaS$$
$$\Rightarrow bbabaa\underline{S}bSbSaSaS$$
$$\Rightarrow bbabaaba\underline{S}bSaSaS$$
$$\Rightarrow bbabaabb\underline{S}aSaS$$
$$\Rightarrow bbabaabba\underline{S}aS$$
$$\Rightarrow bbabaabbaa\underline{S}$$
$$\Rightarrow bbabaabbaa$$

Example 6.7. Design a CFG, which can generate string, having any combination of *a*'s and *b*'s, except null string.

Solution. Let CFG be $G = (V_n, V_t, P, S)$

$$V_n = \{S\}$$
$$V_t = \{a, b\}$$

Productions P are defined as

$(P_1$, say) $S \rightarrow aS$

$(P_2$, say) $S \rightarrow bS$

$(P_3$, say) $S \rightarrow a$

$(P_4$, say) $S \rightarrow b$

We can produce the word *baab* as follows :

$$S \Rightarrow bS \text{ (by } P_2)$$
$$\Rightarrow baS \text{ (by } P_1)$$
$$\Rightarrow baaS \text{ (by } P_1)$$
$$\Rightarrow baab \text{ (by } P_4)$$

The language generated by this CFG is the set of all possible strings of the letters *a* and *b* except for the null string, which we cannot generate.

We can generate any word by the following algorthim :

At the beginning, the working string is the start symbol S select a word to be generated. Read the letters of the desired word from left to right, one at a time. If an *a* is read that is not the last letter of the word, apply P_1 to the working string. If '*b*' is read that is not the last letter of the word, apply P_2 to the working string.

If the last letter is read and it is an *a*, apply P_3 to the working string. If the last letter is read and it is *b*, apply P_4 to the working string.

Productions 3 and 4 can be used only once and one of them can be used. For example, to generated *babb*, we apply in order productions P_2, P_1, P_2 and P_4 as below

$$S \Rightarrow bS \Rightarrow baS \Rightarrow babS \Rightarrow babb.$$

Example 6.8. Design CFG for regular expression

$$r = (a + b)^* \, aa \, (a + b)^*.$$

Solution. Let CFG be

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

Productions P are defined as

$S \rightarrow TaaT$ (say P_1)

$T \rightarrow aT$ (say P_2)

$T \rightarrow bT$ (say P_3)

$T \rightarrow \in$ (say P_4)

Last three productions, *i.e.*, P_2, P_3 and P_4 allows us to generate any word we want from terminal T. If the non-terminal T appears in any working string,

we can apply productions to turn in to any string we want. Therefore, the words generated from S have the form any substring aa any substring

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

which is the language of the all words with a double 'a' in them somewhere.

For example to generate baabaab, we can proceed as follows :

$$\begin{aligned} S &\Rightarrow TaaT \\ &\Rightarrow bTaaT \\ &\Rightarrow baTaaT \\ &\Rightarrow baaTaaT \\ &\Rightarrow baabTaaT \\ &\Rightarrow baab\in aaT \\ &\Rightarrow baabaaT \\ &\Rightarrow baabaabT \\ &\Rightarrow baabaab\in \\ &\Rightarrow baabaab \end{aligned}$$

There are other sequences that can also derive the word *baabaab*.

Example 6.9. Design a CFG for the regular expression $r = (a + b)^*$.

Solution. Let G be CFG

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

P are desined as

$$S \to aS$$
$$S \to bS$$
$$S \to a$$
$$S \to b$$
$$S \to \in$$

The word ab can be generated by the derivation

$$\begin{aligned} S &\Rightarrow aS \\ &\Rightarrow abS \\ &\Rightarrow ab\in \\ &\Rightarrow ab \end{aligned}$$

or by the derivation

$$\begin{aligned} S &\Rightarrow aS \\ &\Rightarrow ab. \end{aligned}$$

Example 6.10. The difference between even 'palindrome' and 'odd palindrome' (whose definitions is obvious) is that when we are finally ready to get rid of S in the even palindrome working string, we must replace

it with ∈. If we were force to replace it with an *a* or *b* instead, we would create a central letter and the result would be a grammar for odd palindrome as follows :

$$S \to aSa$$
$$S \to bSb$$
$$S \to a$$
$$S \to b$$

Solution. If we allow the option of turning the central S into either ∈ or a letter, we would have a grammar for the entire language palindrome :

$$S \to aSa$$
$$S \to bSb$$
$$S \to a$$
$$S \to b$$
$$S \to \in$$

The language $\{a^n b^n\}$ and palindrome are amazingly similar in grammatical structure while the first is nearly a regular expression and the other is for from it.

Example 6.11. Write a CFG for the language $L(G) = \{ww^r : w \in \{0, 1\}^*\}$.

Solution. Let grammar be G.

$$G = (V_n, V_t, P, S)$$
$$V_n = \{S\}$$
$$V_t = \{0, 1\}$$

Productions are defined as

$$S \to 0S0 \qquad (P_1)$$
$$S \to 1S1 \qquad (P_2)$$
$$S \to \in \qquad (P_3)$$

Let us derive a string $\underset{w}{001}\underset{w^r}{100}$ by above CFG.

$$S \Rightarrow 0S0 \quad \text{(by } P_1\text{)}$$
$$\Rightarrow 00S00 \quad \text{(by } P_1\text{)}$$
$$\Rightarrow 001S100 \quad \text{(by } P_2\text{)}$$
$$\Rightarrow 001100 \quad \text{(by } P_3\text{)}$$

Example 6.12. Design a CFG for the language

$$L(G) = \{ab\ (bbaa)^n\ bba\ (ba)^n : n \geq 0\}.$$

Solution. Let CFG be G

$$G = (V_n, V_t, P, S)$$
$$V_n = (S, X, Y)$$
$$V_t = \{a, b\}$$

Productions are defined as $S \to abX$

$$X \to bbYa$$
$$Y \to aaXb$$

$$Y \to \in$$

or we can also defined production as :

$$S \to abX$$
$$X \to bbaaXba$$
$$X \to bba$$

Let us derive a string $ab(bbaa)^2\ bba(ba)^2$

$$S \Rightarrow ab\underline{X}$$
$$\Rightarrow abbb\underline{Y}a$$
$$\Rightarrow abbbaa\underline{X}ba$$
$$\Rightarrow abbbaabb\underline{Y}aba$$
$$\Rightarrow abbbaabbaa\underline{X}baba$$
$$\Rightarrow ab(bbaa)^2bbYa(ba)^2$$
$$\Rightarrow ab(bbaa)^2bb\in a(ba)^2$$
$$\Rightarrow ab(bbaa)^2bba(ba)^2$$

or

$$S \Rightarrow abX$$
$$\Rightarrow abbbaaXba$$
$$\Rightarrow abbbaabbaaXbaba$$
$$\Rightarrow ab(bbaa)^2bba(ba)^2.$$

Example 6.13. Design a CFG for the language

$$L = \{a^n\ b^m : n \neq m\}.$$

Solution. If $n \neq m$ then there are only two cases are possible.

Case 1. $n > m$

Let us say langauge L on condition $n > m$ is

L_1 and $L_1 = \{a^n\ b^m : n > m\}$.

Let us say G_1 be the CFG for the language L_1,

$$G_1 = (V_n^1, V_t^1, P^1, S^1)$$
$$V_n^1 = \{S_1, A, S^1\}$$
$$V_t^1 = \{a, b\}$$

then productions P^1 are defined as

$$S^1 \to AS_1$$
$$S_1 \to aS_1b/\in$$
$$A \to aA/a$$

Case 2. $n < m$

Let us say L on condition $n < m$ is L_2

$$L_2 = \{0^n1^m : n < m\}$$

and

Let CFG for G_2 be G_2

$$G_2 = (V_n^2, V_t^2, P^2, S^2)$$

$$V_n^2 = \{S^2, S_2, B\}$$
$$V_t^2 = \{a, b\}$$

P^2 are defined as follows :

$$S^2 \to S_2B$$
$$S_2 \to aS_2b/\in$$
$$B \to bB/b$$

by the help (by the combining G_1 and G_2) of G_1 and G_2 we can write the CFG for L as

$$S \to S^1/S^2$$

where S is start symbol of CFG for L.

Example 6.14. Design a CFG for the language

$$L = \{0^n\ 1^n/n \geq 0) \cup (1^n\ 0^n/n \geq 0)\}.$$

Solution. We can assure L as

$$L = L_1 \cup L_2$$
$$L_1 = \{0^n\ 1^n/n \geq 0\}$$

Let say G_1 be CFG for the language L_1

$$G_1 = (V_n, V_t, P, S)$$
$$V_n = \{S_1\}$$
$$S = \{S_1\}$$
$$V_t = \{0, 1\}$$

and P is defined as

$$S_1 \to 0S_11/\in$$

and $L_2 = \{1^n0^1/n \geq 0\}$

Let say G_2 be CFG for language L_2

$$G_2 = (V_n, V_t, P, S)$$
$$V_n = \{S_2\}$$
$$S = \{S_2\}$$
$$V_t = \{0, 1\}$$

P is defined as

$$S_2 \to 1S_20/\in$$

Since $L = L_1 \cup L_2$, suppose G be CFG for language L, with starting non-terminal S.

Then productions in G will be

$$S \to S_1/S_2$$
$$S_1 \to 0S_11/\in$$
$$S_2 \to 1S_20/\in.$$

Example 6.15. Design CFG for $\Sigma = \{a, b\}$ that generates the set of

(a) all strings with exactly one a.

(b) all strings with at least one a.

(c) all strings with at least 3 a's.

Solution. (a) Let CFG be X G_1

$$G_1 = (V_n, V_t, P, S)$$
$$V_n = \{S, A\}$$
$$V_t = \{a, b\}$$

P is defined as

$$S \rightarrow AaA$$
$$A \rightarrow bA/\in$$

Let us derive a string *bbaab*

$$S \Rightarrow AaA$$
$$\Rightarrow bAaA$$
$$\Rightarrow bbAaA$$
$$\Rightarrow bbAabA$$
$$\Rightarrow bbAab\in$$
$$\Rightarrow bbAab$$

How we don't have any choice for derving second G so string *bbaab* cannot be derive from G_1.

(b) Let CFG be G_2

$$G_2 = (V_n, V_t, P, S)$$
$$V_n = \{S, A\}$$
$$V_t = \{a, b\}$$

P is defined as

$$S \rightarrow AaA$$
$$A \rightarrow aA/bA/\in$$

Let us derive string *baab*

$$S \Rightarrow AaA$$
$$\Rightarrow bAaA$$
$$\Rightarrow baAaA$$
$$\Rightarrow bae\in AaA$$
$$\Rightarrow baaA$$
$$\Rightarrow baabA$$
$$\Rightarrow baab\in$$
$$\Rightarrow baab$$

(c) Let CFG be G_3

and

$$G_3 = (V_n, V_t, P, S)$$
$$V_n = \{S, A\}$$
$$V_t = \{a, b\}$$

P is defined as

$$S \rightarrow AaAaAaA$$
$$A \rightarrow aA/bA/\in.$$

Example 6.16. Give the simple description of the language generated by the grammar with production

$$S \rightarrow aA$$
$$A \rightarrow bS$$
$$S \rightarrow \in.$$

Solution. Given CFG is

$$S \rightarrow aA \quad \text{(say } P_1\text{)}$$
$$A \rightarrow bS \quad \text{(say } P_1\text{)}$$
$$S \rightarrow \in \quad \text{(say } P_3\text{)}$$

From the production P_3, CFG can generate null string. Now let us see production P_1 (*i.e.* $S \rightarrow aA$), once we use it then we have to $A \rightarrow bS$. Now again we have to use either production P_1 ($S \rightarrow aA$) or production P_3 ($S \rightarrow \in$), so clearly the langauge of above CFG will be as follows :

$$L = \{(ab)^n / n \geq 0\}$$

Clearly language L is the set of strings. Starts with ab followed by any number of ab's. Null string ($\in$) also included in language L.

Example 6.17. Give the simple description of the language generated by the grammar with productions.

$$S \rightarrow SS$$
$$S \rightarrow (S)$$
$$S \rightarrow \in.$$

Solution. Given CFG is

$$S \rightarrow SS$$
$$S \rightarrow (S)$$
$$S \rightarrow (\in)$$

The language generated by the CFG is set of strings, contains balanced set of paranthesis that is for every left paranthesis there is a right paranthesis, and strings like ( ) ( ) ( ) ... or ((( ))) ( ) ( ) ((( ))) ... are the part of the language of CFG.

Example 6.18. Write the CFG for the language

$$L = \{a^n\ b^n\ c^m\ d^m / n > =1,\ m > =1\}.$$

Solution. Let CFG for the language L be G

$$G = (V_n, V_t, P, S)$$
$$V_n = \{S, X, Y\}$$
$$V_t = \{a, b\}$$

Productions P is defined as follows :

$$S \rightarrow XY \quad \text{(Production } P_1\text{)}$$
$$X \rightarrow aXb/ab \quad \text{(Production } P_2\text{)}$$
$$Y \rightarrow cYd/cd \quad \text{(Production } P_3\text{).}$$

When we use P_1 (*i.e.* $S \rightarrow XY$) then X can generate equal number of a's and b's with no a's follows when any b is encountered. Similarly Y can generate strings of equal number of c's and d's with no c's follows when any d is encountered.

Let us derive a string aaabbbcd that $n = 3$, $m = 1$

$$\begin{aligned} S &\Rightarrow XY \\ &\Rightarrow aXbY \\ &\Rightarrow aaXbbY \\ &\Rightarrow aaabbbY \\ &\Rightarrow aaabbbcd \end{aligned}$$

which required string.

Example 6.19. Design a CFG for the language

$$L = \{a^n\ b^m\ c^m\ d^n / n \geq 1,\ m >= 1\}.$$

Solution. Let us analyse the language L first, language L is the set of strings which contain equal number of a's and d's and equal number of b's and c's in between a's and d's. There is no 'a' followed by b, c and, there is no 'b' followed by c and, there is no c followed by d. Similarly there is no b, c, and d before any 'a', there is no 'c' and d before any 'b', finally there is no d before any c.

On the base of above description let us design a CFG for the language.

Let us assume CFG be G

$$\begin{aligned} G &= (V_n, V_t, P, S) \\ V_n &= \{S, A\} \\ V_t &= \{a, b, c, d\} \end{aligned}$$

and P is defined as

$$\begin{aligned} S &\rightarrow aSd && \text{(say } P_1\text{)} \\ S &\rightarrow aAd && \text{(say } P_2\text{)} \\ A &\rightarrow bAc && \text{(say } P_3\text{)} \\ A &\rightarrow bc && \text{(say } P_4\text{)} \end{aligned}$$

Let us derive a string *aabbbcccdd* from grammar G.

$$\begin{aligned} S &\Rightarrow aSd && \text{(by } P_1\text{)} \\ &\Rightarrow aaAdd && \text{(by } P_2\text{)} \\ &\Rightarrow aabAcdd && \text{(by } P_3\text{)} \\ &\Rightarrow aabbAccdd && \text{(by } P_3\text{)} \\ &\Rightarrow aabbbcccdd && \text{(by } P_4\text{)} \end{aligned}$$

which is required string.

Example 6.20. Design a CFG for the language $L = \{a^n\ b^{2n} / n >= 0\}$.

Solution. Let CFG be G for the language L

$$\begin{aligned} G &= (V_n, V_t, P, S) \\ V_n &= \{S\} \\ V_t &= \{a, b\} \end{aligned}$$

and P is defined as

$S \rightarrow aSbb/\in$

Let us derive a string *aaabbbbbb*

$S \Rightarrow aSbb$
$\Rightarrow aaSbbbb$
$\Rightarrow aaaSbbbbbb$
$\Rightarrow aaa\in bbbbbb$
$\Rightarrow aaabbbbbb$

which is required string.

Now let us derive another string *abbb*.

$S \Rightarrow aSbb$

From this derivation only it is clear that third b is not possible against a single 'a'.

Example 6.21. Write the CFG for the language $L = \{a^{2n}\ b^m / n > 0,\ m >= 0\}$.

Solution. Let us analyse the language first language L is the set of strings, if string starts with a then number of a's are even, followed by any number of times b's. There is no 'a' in any string after first 'b' is encountered similarly there is no 'b' before any 'a' in the string.

Let suppose CFG for the L is G

$G = (V_n, V_t, P, S)$
$V_n = \{S, A, B\}$
$V_t = \{a, b\}$

Productions P is defined as

$S \rightarrow aaAB$ (say P_1)
$A \rightarrow aaA/\in$ (say P_2)
$B \rightarrow bB/\in$ (say P_3)

Let us derive the string *aaaabbb*

$S \Rightarrow aaAB$ (by P_1)
$\Rightarrow aaaaAB$ (by P_2)
$\Rightarrow aaaa\in B$ (by P_2)
$\Rightarrow aaaabB$ (by P_3)
$\Rightarrow aaaabbB$ (by P_3)
$\Rightarrow aaaabb\in$ (by P_3)
$\Rightarrow aaaabb$

which is required string.

Example 6.21(*a*). Write CFG for

$L = \{(a^n\ b^n\ c^m\ d^m / n >= 1,\ m >= 1) \cup (a^n\ b^m\ c^m\ d^n / n >= 1,\ n >= 1\}$.

Solution. Let suppose $L = L_1 \cup L_2$

where $L_1 = \{a^n\ b^n\ c^m\ d^m / n >= 1,\ m >= 1\}$

Let CFG for L_1 is G_1

$G_1 = (V_n, V_t, P_1, S_1)$
$V_n = \{S_1, A, B\}$

$$V_t = \{a, b, c, d\}$$

and P_1 is defined as follows

$$S_1 \rightarrow AB$$
$$A \rightarrow aAb/ab$$
$$B \rightarrow cBd/cd$$
$$L_2 = \{a^n b^m c^m d^n / n >= 1, m >= 1\}$$

Let us assume that CFG for L_2 is G_2

and
$$G_2 = (V_n, V_t, P_2, S_2)$$
$$V_n = \{S_2, D, E\}$$
$$V_t = \{a, b, c, d\}$$

P_2 is defined as follows

$$S_2 \rightarrow aDd$$
$$D \rightarrow aDd/E$$
$$E \rightarrow bEc/bc$$

By the help of the G_1 and G_2, we can define CFG for the language L, since $L = L_1 \cup L_2$

so $G = G_1 \cup G_2$

Let suppose
$$G = (V_n, V_t, P, S)$$
$$V_n = \{S, S_1, S_2, A, B, D, E\}$$
$$V_t = \{a, b, c, d\}$$

and P is defined as (for $G = G_1 \cup G_2$)

$$S \rightarrow S_1/S_2$$
$$S_1 \rightarrow AB$$
$$A \rightarrow aAb/ab$$
$$B \rightarrow cBd/cd$$
$$S_2 \rightarrow DE$$
$$D \rightarrow aDd/E$$
$$E \rightarrow bEc/bc$$

Example 6.22. Write a CFG for language $L = \{0^i 1^j 2^k / k \leq i$ or $k \leq j\}$.

Solution. In the same fashion as we solve Example 6.21, we approach this problem.

Let us assume $L = L_1 \cup L_2$

where
$$L_1 = \{0^i 1^j 2^k / k \leq i\}$$

and
$$L_2 = \{0^i 1^j 2^k / k \leq j\}$$

Let as assume that CFG for L_1 is G_1

$$G_1 = (V_n, V_t, P_1, S_1)$$
$$V_n = \{S_1, X, Y\}$$
$$V_t = \{0, 1\}$$

and P_1 is defined as

$$S_1 \rightarrow 0S_1 2/0Y2$$
$$Y \rightarrow C/0Y/\epsilon$$
$$C \rightarrow 1C/\epsilon$$

Let us assume that CFG for L_2 is G_2

$$G_2 = (V_n, V_t, P_2, S_2)$$
$$V_n = \{S_2, A, B\}$$
$$V_t = \{0, 1\}$$

P_2 is defined as follows

$$S_2 \rightarrow AB$$
$$A \rightarrow 0A/\epsilon$$
$$B \rightarrow 1B2/1B/\epsilon$$

We can define CFG for L (since $L = L_1 \cup L_2$)

Let us assume that CFG for L is G

$$G = (V_n, V_t, P, S)$$
$$V_n = \{S_1, S_2, X, Y, Z, B\}$$
$$V_t = \{0, 1\}$$

P is defined as

$$S \rightarrow S_1/S_2$$
$$S_1 \rightarrow 0S_12/0Y2$$
$$Y \rightarrow C/0Y/\epsilon$$
$$C \rightarrow 1C/\epsilon$$
$$S_2 \rightarrow AB$$
$$A \rightarrow 0A/\epsilon$$
$$B \rightarrow 1B2/1B/\epsilon.$$

Example 6.23. Write the CFG for the langauge

$$L = \{0^i\ 1^j\ 2^k / i = j \text{ or } j = k\}.$$

Solution. Let us assume $L = L_1 \cup L_2$

where $L_1 = \{0^i\ 1^j\ 2^k / i = j\}$

and $L_2 = \{0^i\ 1^j\ 2^k / j = k\}$

Let us consider L_1 first, let CFG for L_1 be G_1

$$G_1 = (V_n, V_t, P_1, S_1)$$
$$V_n = \{S_1, A, B\}$$
$$V_t = \{0, 1, 2\}$$

P_1 is defined as follows

$$S_1 \rightarrow AB$$
$$A \rightarrow 0A1/\epsilon$$
$$B \rightarrow 2B/\epsilon$$

Now let us assume that CFG for language L_2 is G_2

$$G_2 = (V_n, V_t, P_2, S_2)$$
$$V_n = (S_2, C, D)$$
$$V_t = (0, 1, 2)$$

Productions are defined as follows :

$$S_2 \rightarrow CD$$
$$C \rightarrow 0C/\epsilon$$
$$D \rightarrow 1D2/\epsilon$$

By the help G_1 and G_2 we can define CFG for the language L, Let it is G.

$$G = (V_n, V_t, P, S)$$
$$V_n = \{S, S_1, S_2, A, B, C, D\}$$
$$V_t = \{0, 1, 2\}$$

Productions are defined as follows

$$S \rightarrow S_1 / S_2$$
$$S_1 \rightarrow AB$$
$$A \rightarrow 0A1/\in$$
$$B \rightarrow 2B/\in$$
$$S_2 \rightarrow CD$$
$$C \rightarrow 0C/\in$$
$$D \rightarrow 1D2/\in$$

which is required CFG.

Example 6.24. Find the CFG for the following language

$$L = \{a^n\ b^{2n}\ c^m / n,\ m >= 0\}.$$

Solution. Let us analyse the language first, L is language, which contain a set of strings such that every string may start from a or c but not by b.

If string starts with 'a' then number of a's must follow b's and the, number of b's are twice then the number of a's. If string does start with a then it starts with c, followed by any number of c's. There should be no a after any b or any c, and no b after any c. Similarly there should be no b or c before 'a' and no c before any b.

On the basic of above discussion, let us assume that CFG for the languageis G.

$$G = (V_n, V_t, P, S)$$
$$V_n = \{S, A, B\}$$
$$V_t = \{a, b, c\}$$

P is defined as follows

$$S \rightarrow AB$$
$$A \rightarrow aAbb/\in$$
$$B \rightarrow cB/\in$$

which is required CFG.

Example 6.25. Write the CFG for the language

$$L = \{a^n\ b^m\ c^{2m} / n,\ m >= 0\}.$$

Solution. This problem is very similar to Example 6.25.

Let us assume CFG be G

$$G = (V_n, V_t, P, S)$$
$$V_n = (S, A, B)$$
$$V_t = (a, b, c)$$

P is defined as follows

$$S \to AB$$
$$A \to aA/\in$$
$$B \to bBcc/\in$$

which is required grammar.

6.2.5. Left most and Right most Derivations

In order to restrict the number of choices of replacement of variables, if at each step we replace the "left most" variable by one of its production bodies such a derivation is called a "left most derivation", and we indicate that a derivation is left most by using relation $\underset{lm}{\Rightarrow}$ and $\overset{*}{\underset{lm}{\Rightarrow}}$ for one or many steps, respectively.

Similarly, it is possible that at each step the "right most" variable is replaced by one of its bodies. If so, we call the derivation "right most" and use symbols $\underset{rm}{\Rightarrow}$ and $\overset{*}{\underset{rm}{\Rightarrow}}$ to indicate one or many right most derivation steps, respectively.

Example 6.26. (*i*) Write a CFG for solving simple expression, such + and *.

(*ii*) Also write CFG for regular expression

$$r = (a + b)\ (a + b + 0 + 1)^*.$$

(*iii*) Derive the string (which is defined in part (*ii*)) a^* (a + b00) by applying left most derivation and right most derivation.

Solution. (*i*) We need two variables in this grammar. One, which we call E, represents the expression, it is the start symbol and represents the language of expressions we are defining. The other variable, I, reprsents the identifiers.

Let CFG be $G = (V_n, V_t, P, S)$

$V_n = \{E, I\}$

$V_t = \{+, *, (,)\}$

P is defined as

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

Here I can derive any terminal symbol as

$$I \to \text{terminal}.$$

(*ii*) Given regular epression is

$(a + b)\ (a + b + 0 + 1)^*$

Let CFG be $G = (V_n, V_t, P, S)$

$V_n = \{E, I\}$

$V_t = \{+, *, (,), a, b, 0, 1\}$

P is defined as follows : $E \to I$

$E \to E * E$

$E \to E + E$

$E \to (E)$

$I \to a$

$I \to b$

$I \to Ia$

$I \to Ib$

$I \to I0$

$I \to I1$

$I \to 0$

$I \to 1.$

(*iii*) String is $a * (a + b00)$, and we want to derive it by using left most and right most derivations, from CFG defined in part (*ii*)

By using leftmost derivation :

$$E \underset{lm}{\Rightarrow} \underline{E} * E \underset{lm}{\Rightarrow} \underline{I} * E \underset{lm}{\Rightarrow} a * \underline{E} \underset{lm}{\Rightarrow} a * (\underline{E}) \underset{lm}{\Rightarrow} a * (\underline{E} * E) \underset{lm}{\Rightarrow} a * (\underline{I} + E) \underset{lm}{\Rightarrow}$$

$$a * (a + \underline{E}) \underset{lm}{\Rightarrow} a * (a + \underline{I}\,0) \underset{lm}{\Rightarrow} a * (a + \underline{I}\,00) \underset{lm}{\Rightarrow}$$

$$\underset{lm}{\Rightarrow} a * (a + b00)$$

which is required string.

By using right most derivation :

$$E \underset{rm}{\Rightarrow} E * \underline{E} \underset{rm}{\Rightarrow} E * (\underline{E}) \underset{rm}{\Rightarrow} E * (E + \underline{E}) \underset{rm}{\Rightarrow} E * (E + \underline{I}) \underset{rm}{\Rightarrow} E * (E + \underline{I}0) \underset{rm}{\Rightarrow} E * (E + \underline{I}00) \underset{rm}{\Rightarrow}$$

$$E * (\underline{E} + b00) \underset{rm}{\Rightarrow} E * (\underline{I} + b00) \underset{rm}{\Rightarrow} \underline{E} * (a + b00) \underset{rm}{\Rightarrow}$$

$$\underset{rm}{\Rightarrow} \underline{I} * (a + b00) \underset{rm}{\Rightarrow} b * (a + b00)$$

which is required string.

6.3. Parse Tree

There is a tree representation for derivation that has proved extremely useful. It is the second way of showing derivations, independent of the order in which productions are used, is also called "derivation tree". "A parse tree is an odered tree in which nodes are labeled with the left sides of productions and in which the children of a node represent its corresponding right sides".

6.3.1. Definition

Let $G = (V_n, V_t, P, S)$ be a context-free grammar. An ordered tree for this CFG, G is a derivation tree if and only if it has the following properties.

(*a*) The root is labeled by the starting non-terminal of the CFG that is S.

(*b*) Every leaf of the orderd ordered tree has a label from $V_t \cup \{\in\}$.

(*c*) Every Interior node of ordered tree has a label from V_n.

(*d*) Let us assume that a vertex has label $X \in V_n$, and its children are labeled (from left to right) $y_1, y_1, y_3, \ldots, y_n$, then production must contain a production of the form

$$X \rightarrow y_1, y_2, \ldots, y_n.$$

(*e*) A leaf labeled $\in$ has no siblings, that is, vertex with a child labeled $\in$ can have no other children. Clearly if the leaf is labeled $\in$, then it must be the only child of its parent.

6.3.2. The Yield of Parse Tree

If we look at the leaves of any parse tree and can concate them from the left, we get a string, called the yield of the tree, which is always a string that is derived from the root will be proved shortly of special importance are those parse trees such that :

(*a*) The yield is a terminal string. That is all leaves are labeled either with a terminal or with $\in$.

(*b*) The root is labeled by the start symbol.

Now let us see some examples based on the parse tree.

Example 6.27. Consider the CFG

$$S \rightarrow XX$$

$$X \rightarrow XXX/bX/Xb/a$$

Find the parse tree for the string *bbaaaab*.

Solution. Given CFG is

$$S \rightarrow XX$$

$$X \rightarrow XXX/bX/Xb/a$$

and string is $w = bbaaaab$

We begin with S and apply the production $S \rightarrow XX$.

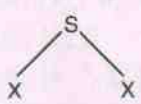


Fig. 6.1.

To the left hand X, let us apply the production $X \rightarrow bX$. To the right hand X, let us apply $X \rightarrow XXX$.

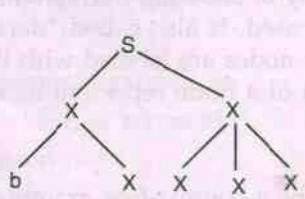


Fig. 6.2.

The b that we have on the bottom line is a terminal, so it does not descend further. In the terminology of trees, it is called a "terminal node". Let the

four A's left to right, undergo the production $A \to bA$, $A \to a$, $A \to a$, and $A \to Ab$, respectively. We now have

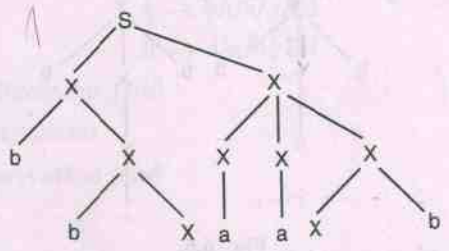


Fig. 6.3.

Let us finish off the generation of a word with the production $X \to a$ and $X \to a$:

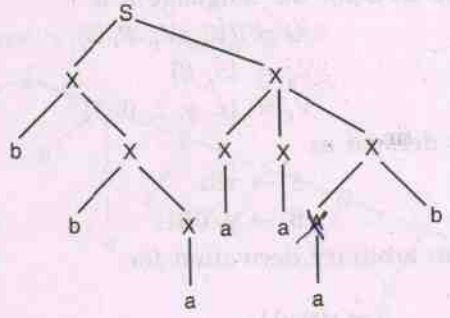


Fig. 6.4.

Reading from left to right, we see the word we have produced is *bbaaaab*.

These tree diagrams are called "syntax trees", "parse trees", "generation" trees, "production trees" or derivation trees.

Example 6.28. Consider the grammar G, with production

$$S \to aXY$$
$$X \to bYb$$
$$Y \to X/\epsilon.$$

Solution. Let us see a partial derivation tree for G, this tree gives *abbbb* as yield.

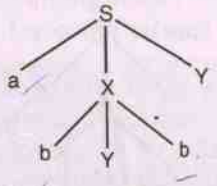


Fig. 6.5.

Now extend this partial derivative tree for the string *abbbb*.

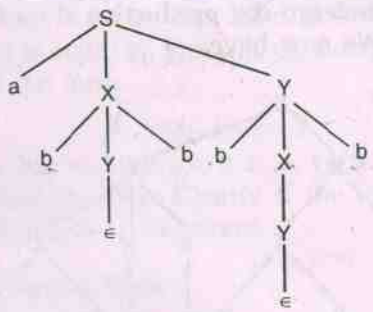


Fig. 6.6.

the yield is $ab \in bb \in b$ that is *abbbb*.

Example 6.29. Write the CFG for the language $L = \{x\ 0^n y 1^n z / n > =0\}$, and give the parse tree for the string $x000\ y11z$.

Solution. Let the CFG for the language L is

$$G = (V_n, V_t, P, S)$$

where

$$V_n = \{S, B\}$$
$$V_t = \{x.\ y,\ z,\ 0,\ 1\}$$

and production P is defined as

$$S \rightarrow xBz$$
$$B \rightarrow y/0B1.$$

Now construct an arbitrary derivation for

$$S \overset{*}{\underset{G}{\Rightarrow}} x00\ y11z.$$

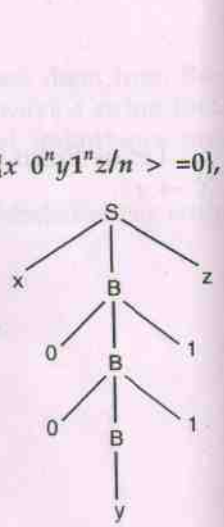


Fig. 6.7.

One possible derivation using the above grammar is

$$S \Rightarrow x\underline{B}z \Rightarrow x0B1z \Rightarrow x00\underline{B}11z \Rightarrow x00y11z$$

Now we are able to design the parse tree for the string $x00y11z$ from the above CFG. Parse tree is shown in Fig. 6.7.

Example 6.30. Consider the CFG 'G' whose productions are

$$S \rightarrow aAS/a$$
$$A \rightarrow SbA/SS/ba.$$

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

Solution. $S \Rightarrow a\underline{A}S \Rightarrow a\underline{S}bAS \Rightarrow aab\underline{A}S \Rightarrow aabbaS \Rightarrow a^2b^2a\underline{S} \Rightarrow a^2b^2a^2$

The derivation tree is given in Fig. 6.8.

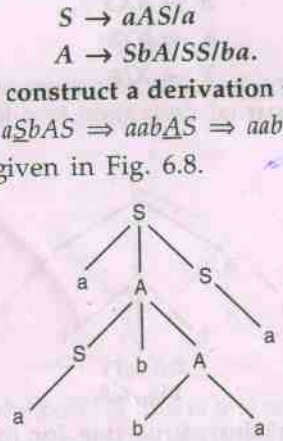


Fig. 6.8.

Derivation tree with yield *aabbaa*.

Example 6.31. Let G be CFG

$$S \to bB/aA,$$
$$A \to b/bS/aAA$$
$$B \to a/aS/bBB.$$

For the string *bbaababa* find

(*i*) left most derivation

(*ii*) rightmost derivation and

(*iii*) parse tree.

Solution. (*i*) Left most derivation for string w = *bbaababa* is

$S \Rightarrow b\underline{B} \Rightarrow bb\underline{B}B \Rightarrow bba\underline{B} \Rightarrow bbaa\underline{S} \Rightarrow b^2a^2b\underline{B} \Rightarrow b^2a^2ba\underline{S} \Rightarrow b^2a^2bab\underline{B} \Rightarrow \underline{b}^2a^2baba$

(*ii*) The right most derivation is

$S \Rightarrow b\underline{B} \Rightarrow bbB\underline{B} \Rightarrow bb\underline{B}aS \Rightarrow bb\underline{B}abB \Rightarrow b^2Baba\underline{S} \Rightarrow b^2Bababb\underline{B} \Rightarrow b^2\underline{B}ababa \Rightarrow b^2a^2baba$

The derivation tree is following in Fig. 6.9.

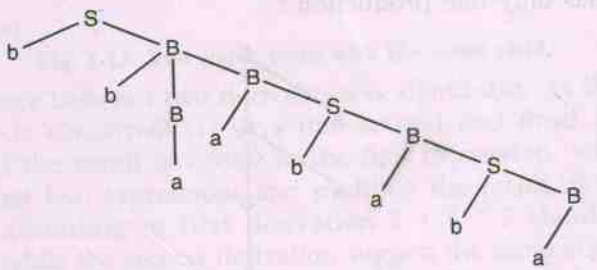


Fig. 6.9.

Yield is *bbaababa*.

6.4. Ambiguity in Grammars and Languages

Grammars can be used to put structure on programs and documents. The assumption was that a grammar uniquely determines a structure for each string in its language. However not every grammar does provide unique structure.

When a grammar fails to provide unique structures, it is sometimes possible to redesign the grammar to make the structure unique for each string in the language. Unfortunately sometimes we cannot do so. That is, there are some CFL's that are "inherently ambiguous"; every grammar for the language puts more than one structure for some strings in the language.

Definition A CFG is called ambiguous if for at least one word in the lanauge that it generates, there are two possible derivations of the word that correspond to different syntax trees. If a CFG is not ambiguous, it is called unambiguous.

Example 6.32. The language of all non null strings of *a*'s can be defined by a CFG as follows :

$$S \to aS/Sa/a$$

Solution. In this case, the word a^3 can be generated by four different tree's :

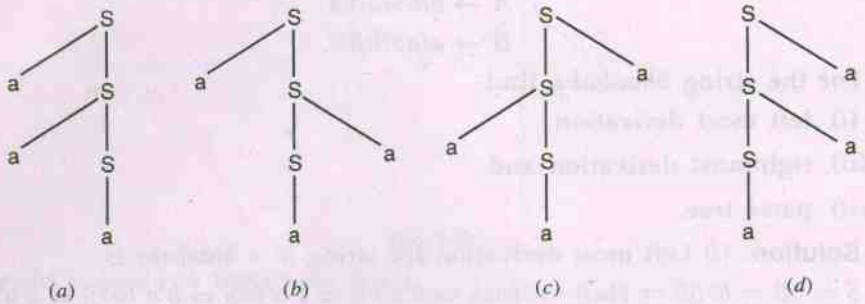


Fig. 6.10.

The CFG is therefore ambiguous.

However, the same language can also be defined by the CFG

$$S \rightarrow aS/a$$

for which a^3 has only one production :

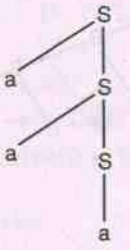


Fig. 6.11.

Example 6.33. The CFG $S \rightarrow aSb/SS/\in$ is ambiguous. The sentence *aabb* has the two derivation trees shown in Fig. 6.12 (*a*) and 6.12 (*b*).

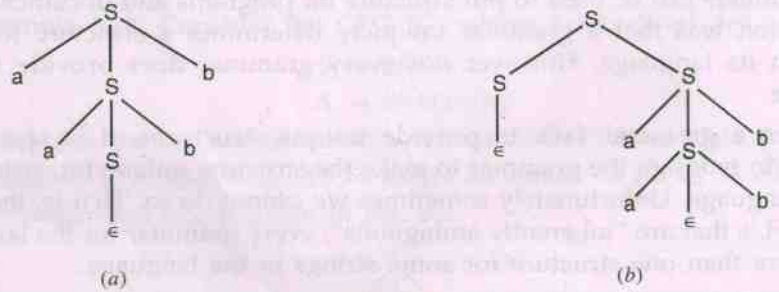


Fig. 6.12.

Example 6.34. Consider the grammar

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

$I \to b$
$I \to Ia$
$I \to Ib$
$I \to I0$
$I \to I1$

For instance, consider the sentential form $E + E * E$. It has two derivation from E :

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

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

Figure 6.13 shows the two parse trees, which we should note are distinct trees.

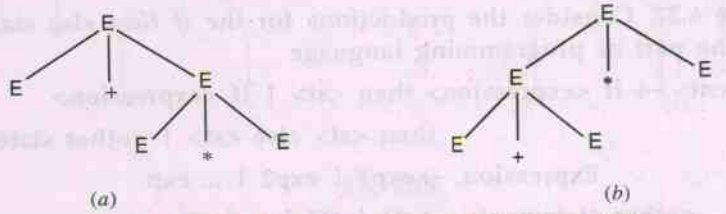


Fig. 6.13. Two parse trees with the same yield.

The difference between two derivations is significant. As the structure of the expression is concerned (1) says that second and third expression are multiplied, and the result is added to the first expression, while derivation (2) adds the first two expressions and multiply the result by the third one. For example, according to first derivation 2 + 5 * 3 should be grouped 2 + (5 * 3) = 17, while the second derivation suggest the same expression should be grouped (2 + 5) * 3 = 21. Obviously, the first of these, and not the second, matches our notation of correct grouping of arithmatic expressions.

Example 6.35. Using the same grammar as in Example 6.34, we find that strng $a + b \times b$ has many different derivations trees, as follows.

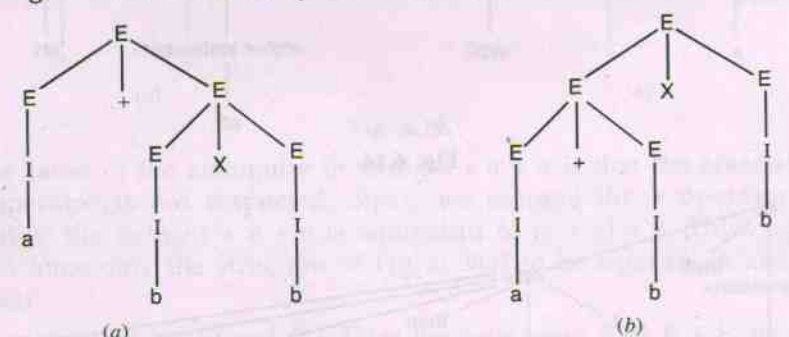
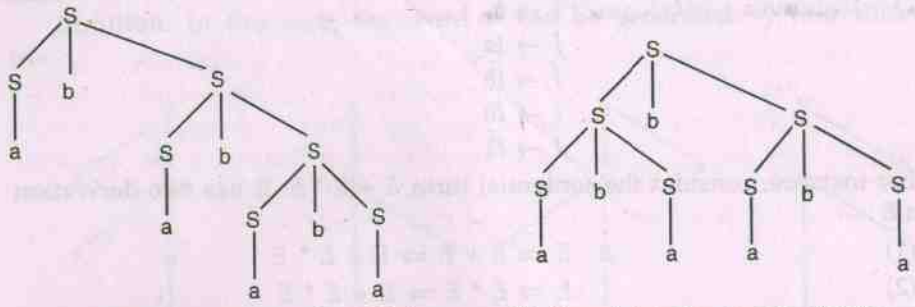


Fig. 6.14.

Example 6.36. If CFG a is $S \to SbS/a$, show that G is ambiguous.

Solution. To prove that G is ambiguous, we have to find a string, $w \in L(G)$ which is ambiguous.

Consider the $w = abababa \in L(G)$. Then we get two derivation tree for w. Thus grammar G is ambiguous.



(*a*) First derivation tree for *abababa*. (*b*) Second derivation tree for *abababa*.

Fig. 6.15.

Example 6.37. Consider the productions for the *if then else* statement which are the part of programming language

<statement> → if <expression> then <st> | if <expression> then <st> else <st> | <other statement>

Expression →exp1 | exp2 | ... exp.

<other statement> →st1 | st2 | ... | st*n*.

Prove that grammar is ambiguous.

Solution. Consider the statement

if exp1 then if exp2 then st1 else st2

There are two different parse trees as shown in Fig. 6.16 and Fig. 6.17.

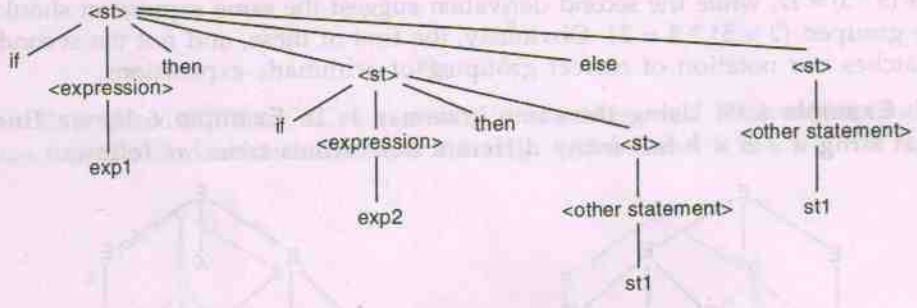


Fig. 6.16.

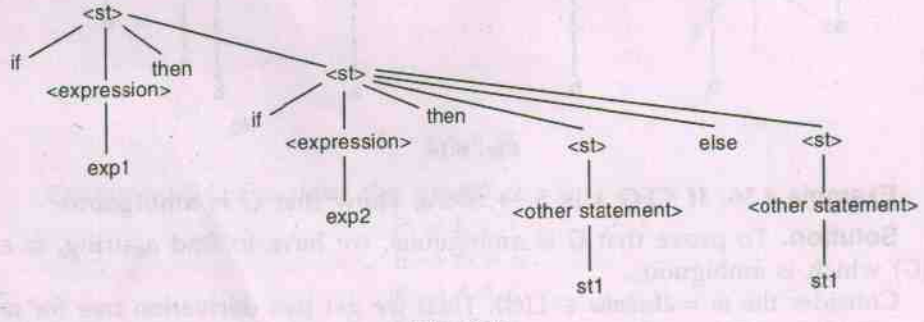


Fig. 6.17.

Hence grammar is ambiguous.

6.4.1. Removing Ambiguity from Grammars

If a context free grammar is ambiguous, it is often possible and usually desirable to find an equivalent unambiguous CFG. Although some CFGs are **"inherently ambiguous"** in the sense that they cannot be produced except by ambiguous grammars. Ambiguity is usually the property of the grammar rather than the language. Let us consider the grammar for algebraic expressions and for the string $a + a^*a$ the two parse tree are possible.

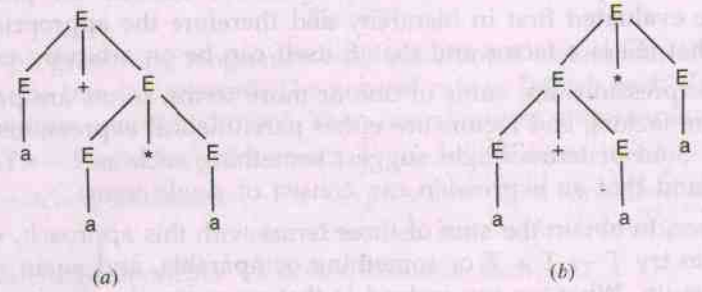


Fig. 6.18.

The first cause of ambiguity in the grammar is that the precedence of the operator + and * is not respected. Both parsing trees in Fig. 6.18 are valid. We need to force only the structure of Fig. 6.18(a) to be legal in an ambiguous grammar in which the * operator has higher precedence then + because $a + a^*a$ is equivalent to $a + (a^*a)$ and $a^*a + a$ is equivalent to $(a^*a) + a$.

Again, the parse trees for $a + a + a$ are

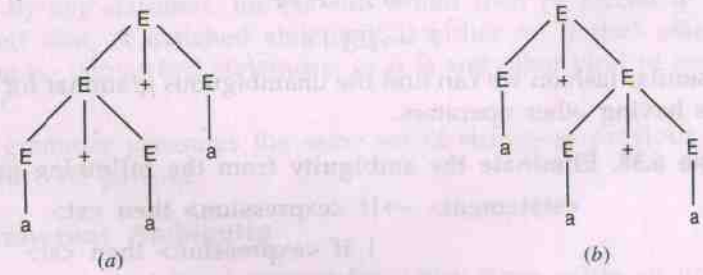


Fig. 6.19.

The cause of the ambiguity in string $a + a + a$ is that the associativity it of + operator is not respected. Since, we assume the + operator is left associative the string $a + a + a$ is equivalent to $(a + a) + a$. Thus, again we need to force only the structure of Fig. 6.19(a) to be legal in an ambiguous grammar.

Because we do not want $E + E$ (as we have seen, $E \rightarrow E + E$, by itself is enough to produce ambiguity) we will not think of expressions involving + as being sums of other expressions. They are obviously sums of something: Let we use the word **terms** to stand for this things that are added *i.e.* combined using + to create expressions. The corresponding variable in the grammar will be T. Expressions can also be products; because the two expressions $a + b * c$ and $a * b + c$ are both sums, it is more appropriate to say that terms can be products. Let us say that "factors" are the things that are multiplied

(combined using *) to produce terms. The corresponding variable in the grammar will be F.

Since the multiplication have higher precedence over addition. We can say that Expressions are sums of one or more terms, and terms are products of one or more factors.

Now, we must deal with parentheses. We might say that (E) could be an expression, a term or a factor. However, evaluation of a parenthetical expression takes precedence over any operators outside the parentheses. Factors are evaluated first in hierarchy and therefore the appropriate choice is to say that (E) is a factor and that E itself can be an arbitrary expression.

Now, expressions are sums of one or more terms, terms are products of one or more factors, and factors are either parenthetical expressions or single identifiers. Sum of terms might suggest something such as $E \rightarrow T + T \mid T$. (keep in mind that an expression can consist of single term)

However, to obtain the sum of three terms with this approach, we would be forced to try $T \rightarrow T + T$ or something comparable, and again we would have ambiguity. What we say instead is that an expression is either a single term or the sum of a term and another experssion. The only question is whether we want $E \rightarrow E + T$ or $E \rightarrow T + E$. Since + operator is left associative, we would probably choose $E \rightarrow E + T$ as more appropriate. Similarly, we choose the production $T \rightarrow T * F$ rather than $T \rightarrow F * T$.

The resulting unambiguous grammar is

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

In the similar fashion we can find the unambiguous grammar for algebraic expressions having other operators.

Example 6.38. Eliminate the ambiguity from the following grammar

<statement> →if <expression> then <st>
| if <expression> then <st>
else <st> | <other statement>

<other statement> →$S_1 \mid S_2 \mid S_3 \mid \ldots S_n$

<expression> →$E_1 \mid E_2 \mid E_3 \ldots \mid E_n$

Solution. The Grammar is ambiguous since the string "if E_1 then if E_2 then S_1 else S_2" has two parse trees.

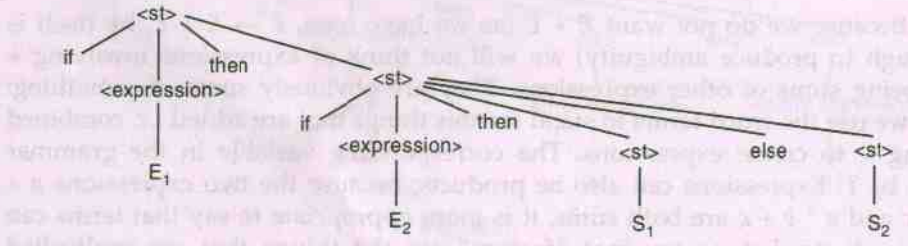


Fig. 6.20.

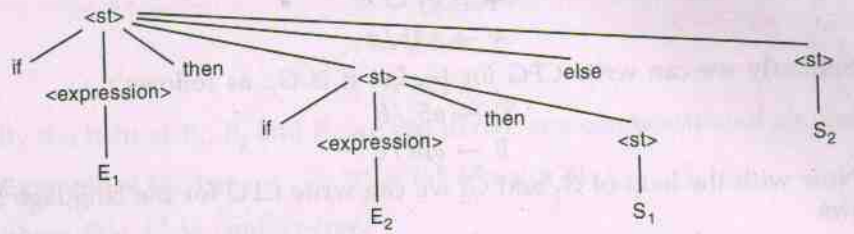


Fig. 6.21.

In all programming languages with conditional statements of this form, the first parse tree is preferred. The general rule is "Match each else with the closest previous unmatched then". Then unambiguous grammar is as follows :

<st> → <matched-statement> | <unmatched-statement>

<matched-statement> → if <expression> then <matched-statement>
else <matched statement> | <other statement>

<unmatched-statement> → if <expression> then <st>
| if <expressions> then <matched-statement>
else <unmatched-statement>

<expression> → $E_1 \mid E_2 \mid \ldots E_n$

<unmatched statement> → $S_1 \mid S_2 \ldots S_n$

<matched statement> → $S_1 \mid S_2 \ldots \mid S_n$

The idea in this grammar is that a statement appearing between a *then* and an *else* must be "matched" *i.e.* it must not end with an unmatched *then* followed by *any statement*, for the *else* would then be forced to match this unmatched *then*. A matched statement is either an if then else statement containing no unmatched statements or it is any other kind of unconditional statement.

This grammar generates the same set of strings as previous one, but it allows only one parsing.

6.4.2. Inherent Ambiguity

"If L is a context-free language for which three exists an unambiguous grammar, then L is said to be unambiguous.

Definition If every grammar that generates L is ambiguous, then the language is said to be "inherently ambiguous".

Let us consider an example for it :

Example 6.39. Show that

$$L = \{a^n\, b^n\, c^m\} \cup \{a^n\, b^m\, c^m\}$$

with n and m non-negative, is an inherently ambiguous context-free language.

Solution. Let us say $L = L_1 \cup L_2$

where $L_1 = \{a^n\, b^n\, c^m\}$

and $L_2 = \{a^n\, b^m\, c^m\}$

Let us write CFG for L_1, let it is G_1, as follows :

$$S_1 \rightarrow S_1\ c/A$$
$$A \rightarrow aAb/\in$$

Similarly we can write CFG for L_2, Let it is G_2, as follows :

$$S_2 \rightarrow aS_2/B$$
$$B \rightarrow bBc/\in$$

Now with the help of G_1 and G_2 we can write CFG for the language L, as follows

$$S \rightarrow S_1/S_2$$

where S is starting non-terminal for CFG of language L.

The grammar is ambiguous since the string $a^n\ b^n\ c^m$ has two distinct derivation, one starting with $S \Rightarrow S_1$, and another with $S \Rightarrow S_2$. It does of course not follows that L is in herently ambiguous as there might exist some other non-ambiguous grammar for it. But in some way L_1 and L_2 have some conflicting requirements, the first putting a restriction on the number of a's and b's, while the second does the same for b's and c's. A few tries will quickly convince us of the impossiblity of combining these requirements in a single set of rules that cover the case $n = m$ uniquely.

Now let us discuss some more examples about these concepts.

Example 6.40. Design a CFG for the language $L = \{a^n\ b^m : n \leq m + 3\}$.

Solution. Let us solve this problem for $n = m + 3$. Then add more b's.

Let us assume CFG for above concept is $G = (V_n, V_t, P, S)$

where $V_n = \{S, A, B\}$

$V_t = \{a, b, \in\}$

and productions are defined as :

$$S \rightarrow aaaA$$
$$A \rightarrow aAb/B$$
$$B \rightarrow Bb/\in$$

But when we analyse this solution, then it is found incomplete since it creates at least three a's. To take care of these cases $n = 1, 2$ we add

$$S \rightarrow \in/aA/aaA$$

So finally solution is

$$S \rightarrow \in/aA/aaA$$
$$S \rightarrow aaaA$$
$$A \rightarrow aAb/B$$
$$B \rightarrow Bb/\in$$

Example 6.41. Find the context-free grammar for the set of all regular expression on the alphabet $\{a, b\}$.

Solution. Let us analyse the regular expression on the alphabet $\{a, b\}$ regular expression by any combination of a's and b's is $r = (a + b)^*$.

Let CFG for regular expression is $G = (V_n, V_t, P, S)$

$$V_n = \{S, A, B\}$$
$$V_t = \{a, b, \in\}$$

and P is defined as

$S \rightarrow AB/BA$ (P_1 say)

$A \rightarrow aA/\in$ (P_2 say)

$B \rightarrow bB/\in$ (P_3 say)

By the help of P_1, P_2 and P_3 we can derive any combination of a's and b's.

Example 6.42. Let $L = \{a^n b^n : n \geq 0\}$

Show that L^2 is context-free.

Solution. $L = \{a^n b^n : n \geq 0\}$

Let CFG for L is $G = (V_n, V_t, P, S)$

$V_n = \{s\}$

$V_t = \{a, b, \in\}$

P is defined as $S \rightarrow aSb$

$S \rightarrow \in$

Clearly L is context-free language, since there exist a CFG for given language.

Now consider a grammar, which is CFG and production of this CFG are defined as follows.

$S_1 \rightarrow SS$

$S \rightarrow aSb$

$S \rightarrow \in$

Now, when carefully analyse these production then we find that, it is the CFG for the language L^2. So L^2 is context free language, provided L is context free.

Example 6.43. Show that the following grammar is ambiguous.

$S \rightarrow AB/aaB$

$A \rightarrow a/Aa$

$B \rightarrow b.$

Solution. Given CFG is

$S \rightarrow AB/aaB$

$A \rightarrow a/Aa$

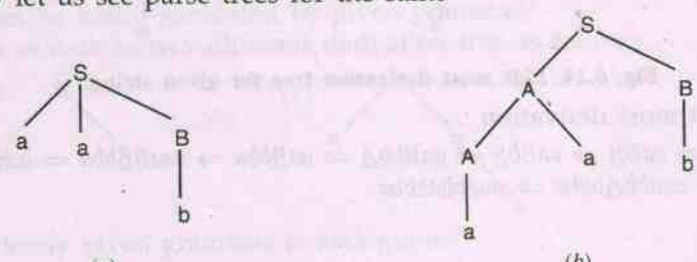
$B \rightarrow b$

The string $w = aab$ has two distinct leftmost derivations, as follows :

(1) $S \Rightarrow aa\underline{B} \Rightarrow aab$

(2) $S \Rightarrow \underline{A}B \Rightarrow \underline{A}aB \Rightarrow aa\underline{B} \Rightarrow aab$

Now let us see parse trees for the same



(a) (b)

Fig. 6.22.

Example 6.44. Find a derivation free of $a * b + a * b$ given that $a * b + a * b$ is in $L(G)$, where G is given by

$$S \to S + S/S * S$$
$$S \to a/b.$$

Solution. Given grammar is

$$S \to S + S/S * S$$
$$S \to a/b$$

and string is $w = a * b + a * b$

$S \Rightarrow \underline{S} + S \Rightarrow \underline{S} * S + S \Rightarrow a * \underline{S} + S \Rightarrow a + b + \underline{S} \Rightarrow a * b + \underline{S} * S$
$\Rightarrow a * b + a * \underline{S} \Rightarrow a * b + a * b$

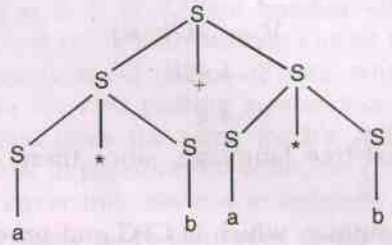


Fig. 6.23. Derivation tree for Example 6.44.

Example 6.45. Consider the grammar

$$S \to aB/bA$$
$$A \to aS/bAA/a$$
$$B \to bS/aBB/b$$

For the string *aaabbabbba* find

(*i*) The left most derivation and left most derivation tree.

(*ii*) The right most derivation and right most derivation tree.

Solution. Given string is *aaabbabbba*.

(*i*) Left most derivation

$S \Rightarrow a\underline{B} \Rightarrow aa\underline{B}B \Rightarrow aaa\underline{B}BB \Rightarrow aaab\underline{B}B \Rightarrow aaabb\underline{B} \Rightarrow aaabba\underline{B}B \Rightarrow aaabbab\underline{B} \Rightarrow$
$aaabbabbS \Rightarrow aaabbabbb\underline{A} \Rightarrow aaabbabbba.$

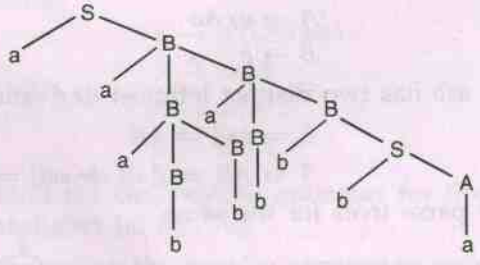


Fig. 6.24. Left most derivation tree for given string.

(*ii*) Right most derivation :

$S \Rightarrow a\underline{B} \Rightarrow aaB\underline{B} \Rightarrow aaBb\underline{S} \Rightarrow aaBbb\underline{A} \Rightarrow aa\underline{B}bba \Rightarrow aaaB\underline{B}bba \Rightarrow aaa\underline{B}bbba \Rightarrow$
$aaab\underline{S}bbba \Rightarrow aaabb\underline{A}bbba \Rightarrow aaabbabbba$

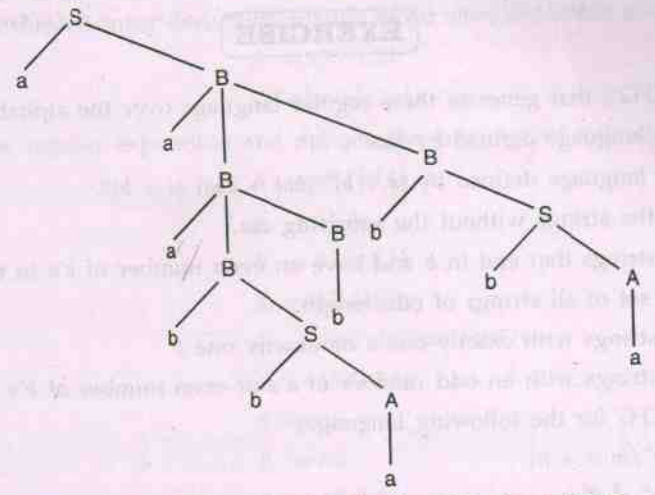


Fig. 6.25. Right most derivation tree for given string.

Example 6.46. Show that the grammar

$$S \rightarrow a/abSb/aAb$$
$$A \rightarrow bS/aAAb$$

is ambiguous.

Solution. Given grammar

$S \rightarrow a$ (P_1 say)
$S \rightarrow abSb$ (P_2 say)
$S \rightarrow aAb$ (P_3 say)
$A \rightarrow bS$ (P_4 say)
$A \rightarrow aAAb$ (P_5 say)

Let us consider a string $w = abab$, it has two different derivations, as follows :

(*i*) $S \Rightarrow abSb \Rightarrow abab$ (by using P_2 and P_1)

(*ii*) $S \Rightarrow aAb \Rightarrow abSb \Rightarrow abab$ (by using P_3, P_4 and P_1)

Example 6.47. Show that grammar

$$S \rightarrow aB/ab$$
$$A \rightarrow aAB/a$$
$$B \rightarrow ABb/b$$

is ambiguous.

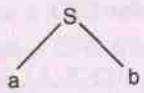
Solution. Let us consider a string

$$w = ab$$

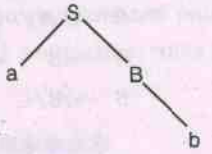
which can be easily generated by given grammar.

Now $w = ab$ as two different derivation tree as follows :

(*a*)



(*b*)



So clearly given grammar is ambiguous.

EXERCISE

1. Find CFG's that generate these regular language over the alphabet $\Sigma = \{a, b\}$.
   (*i*) The language defined by $(aaa + b)^*$.
   (*ii*) The language defined by $(a + b)^* (bbb + aaa) (a + b)^*$.
   (*iii*) All the strings without the substring *aaa*.
   (*iv*) All strings that end in *b* and have an even number of *b*'s in total.
   (*v*) The set of all strings of odd length.
   (*vi*) All strings with exactly one a or exactly one *b*.
   (*vii*) All strings with an odd number of *a*'s or even number of *b*'s.
2. Write CFG for the following languages
   (*i*) $\{a^m b^n / m >= n\}$
   (*ii*) $\{a^m b^n c^p d^q / m + n = p + q\}$
   (*iii*) $\{u\ a\ w\ b / \mu, w \in \{a, b\}^*, |\mu| = |w|\}$
3. Consider the grammar

$$G = (V_n, V_t, P, S)$$

where

$$V_n = \{E, I\}$$
$$V_t = \{a, b, c, +, *, (,)\},$$
$$S = E, \text{ and } P \text{ is given by}$$
$$E \to I$$
$$E \to E + E$$
$$E \to E * E$$
$$E \to (E)$$
$$I \to a/b/c$$

   show that this grammar is ambiguous by deriving two parse tree for string $a + b * c$.
4. Consider the grammar with production

$$S \to aaB$$
$$A \to bBb/\in$$
$$B \to Aa$$

   Show that the string *aabbabba* is not accepted by this grammar.
5. The following grammar generates prefix expression with operands x and y and binary operators +, −, and *

$$E \to +EE/*EE/-EE/x/y.$$

   Find the leftmost and rightmost derivations for the string $+ * - xyxy$.
6. (*i*) Find the left most derivation for the word *abba* in the grammar

$$S \to AA$$
$$A \to aB$$
$$B \to bB/\in$$

(*ii*) Find the left most derivation for the word *abbabaabbbabbab* in the CFG

$$S \rightarrow SSS/aXb$$
$$X \rightarrow ba/bba/abb$$

7. Find the regular expression and defined the language of following CFG's.

(*i*) $S \rightarrow aX/bS/a/b$
$X \rightarrow aX/a$

(*ii*) $S \rightarrow bS/aX/b$
$X \rightarrow bX/aS/a$

(*iii*) $S \rightarrow aaS/abS/baS/bbS/\in$

(*iv*) $S \rightarrow aB/bA/\in$
$A \rightarrow aS$
$B \rightarrow bS$

(*v*) $S \rightarrow aB/bA$
$A \rightarrow aB/a$
$B \rightarrow bA/b$

8. Starting with the alphabet

$$\Sigma = \{a, b, (,), +, *\}$$

Find a CFG that generates all regular expressions.

9. Give the derivation tree for $(((a + b)(c)) + a + b$, using the following grammar

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

10. Show that the language $L = \{wwR : w \in \{a, b\}^*\}$ is not inherently ambiguous.

11. Write the CFG for the following language (with $n \geq 0$, $m \geq 0$)

(*i*) $L = \{a^n b^m : n \neq m - 1\}$

(*ii*) $L = \{w \in \{a, b\}^* : n_a(w) = 2n_b(w) + 1\}$.

12. Find the CFG for the following language (with $n \geq 0$, $m \geq 0$).

(*i*) $L = \{a^n b^m c^k, k \neq n + m\}$

(*ii*) $L = \{a^n b^n c^k : k \geq 3\}$

(*iii*) $L = \{w \in \{a, b, c\}^* : n_a(w) + n_b(w) \neq n_c(w)\}$

13. Define what one right mean by properly rested parenthesis structures involving two kind of parenthesis, say ( ) and [ ]. Intuitively, properly rested strings in this situation are ([ ]), ([[ ]]) [( )], but not ([ )] or (( ]]. Using your definition, give a CFG for generating all properly nested parentheses.

14. Find a CFG that generate all the production rules for CFG with $T = \{a, b\}$ and $V = \{A, B, C\}$.

*****

Chapter 7

Simplified Context-Free Grammar and It's Normal Form

Inside This Chapter

7.1. Introduction

We have already seen, that definition of a context grammar imposes no restriction whatsoever on the right side of a production. Eliminating rules of the form $X \to \in$ and $X \to Y$ made the arguments simpler. In many instances, it is desirable to place even more restriction on the grammar. Due to this, we need to look at method for transforming an arbitrary context free grammar into an equivalent one that satisfies certain restriction on its form. In this chapter, we are going to study several transformation and substitutions that will be useful in subsequent discussion.

We will also see normal forms for context free grammar. These are the form of context free grammars, which are broad enough so that any grammar has an equivalent normal form version. We will study two of the most useful of these, the Chomsky Normal Form (CNF) and the Greibach Normal Form (GNF).

7.2. Reduction of Context Free Grammar

There are several ways in which one can restrict the format of context free grammar without reducing the language generation power of context free grammar. Let L be a non-empty context free language, then it can be generated by a context-free grammar G with the following properties :

(*a*) We must eliminate useless symbols, those variables or terminals that do not appears in any derivation of a terminal string from the start symbol.

(*b*) We must eliminate $\in$-productions, those of the form $X \to \in$ for some variable X.

(*c*) We must eliminate unit production, those of the form $X \to Y$ for variables X and Y.

7.2.1. Eliminating Use Less Symbols

Here we are going to identified those symbols, which do not play any role in the derivation of any string w in $L(G)$ (these symbols are called use less symbols) and then eliminate the identified production, which contains useless symbols, from the context free grammar.

A symbols Y in a context-free grammar is useful it and only if :

(*a*) $Y \overset{*}{\Rightarrow} w$, where $w \in L(G)$ and w in V_t^*, that is Y leads to a string of terminals. Here Y is said to be "generating".

(*b*) If there is a derivation $S \overset{*}{\Rightarrow} \alpha Y \beta \overset{*}{\Rightarrow} w$, $w \in L(G)$, for same α and β, then Y is said to be reachable.

Surely a symbol that is useful will be both generating and reachable. If we eliminate the symbols that are not generating first, and then eliminate from the remaining grammar those symbols that are not reachable. Then, after this process, context free grammar will have only useful symbols.

There for reduction of a given grammar G, involves following steps :

(*a*) Identified non-generating symbols in given CFG and eliminate those productions which contains non-generating symbols.

(*b*) Identified non-reachable symbols in grammar and eliminate those productions which contains non-reachable symbols.

Let us consider examples :

Example 7.1. Consider a CFG

$$S \to AB/a$$
$$A \to b$$

Identified and eliminate useless symbols.

Solution. Given CFG is

$$S \to AB/a$$
$$A \to b$$

Here by observing the given CFG, it becomes very clear that B is non-generating symbol. Since A derives b, S derives a but B is not deriving any string w in the V_t^*.

So we can eliminate $S \to AB$ from the context free grammar, now CFG becomes

$$S \to a$$
$$A \to b$$

Here A is non-reachable symbol, since it can not be reached by starting non- terminal S.

S

↘

a

So we can eliminate $A \rightarrow b$, now grammar is

$$S \rightarrow a$$

which is reduced grammar.

Example 7.2. Remove the useless symbol from the given context free grammar :

$$S \rightarrow aB/bX$$
$$A \rightarrow BAd/bSX/a$$
$$B \rightarrow aSB/bBX$$
$$X \rightarrow SBD/aBx/ad.$$

Solution. First we choose those non-terminals which are deriving to the strings of terminals. The non-terminals A and X directly deriving to the strings of terminal a and ad respectively. Hence they are useful symbols. Since $S \rightarrow bX$ and X is useful symbol, hence S is also useful symbol. But B does not derive any string w in V_t^* so clearly B is a non-generating symbol, so eliminate those productions which contains B. Now grammar becomes :

$$S \rightarrow bX$$
$$A \rightarrow bSX/a$$
$$X \rightarrow ad$$

Again here A is non-reachable, since A can not be reached from S (starting non-terminal), it can be understand by following parse tree in Fig. 7.1.

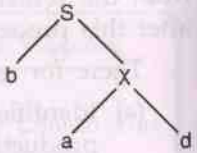


Fig. 7.1.

So eliminate $A \rightarrow bSX/a$, then reduced CFG is

$$S \rightarrow bX$$
$$X \rightarrow ad.$$

Example 7.3. Consider the following grammar and obtain an equivalent grammar containing no useless grammar symbol.

$$A \rightarrow xyz/Xyzz$$
$$X \rightarrow Xz/xYz$$
$$Y \rightarrow yYy/Xz$$
$$Z \rightarrow Zy/z.$$

Solution. Since $A \rightarrow xyz$ and $Z \rightarrow z$, hence A and Z are directly deriving to the string of terminals. Hence they are useful symbols.

Hence X and Y do not lead to a string of terminals, that is, X and Y are useless symbols. Therefore by eliminating these productions we get :

$$A \rightarrow xyz$$
$$Z \rightarrow Zy/z$$

Now since A is the starting non-terminal and right side of A does not contain Z, it means Z is not reachable. Hence after eliminating $Z \rightarrow Zy/z$, grammar becomes

$$A \rightarrow xyz.$$

which is a grammar containing no useless symbols.

Example 7.4. Find the reduced grammar that is equivalent to the CFG given below :

$$S \rightarrow aC/SB$$
$$A \rightarrow bSCa$$
$$B \rightarrow aSB/bBC$$
$$C \rightarrow aBC/ad.$$

Solution. Since $C \rightarrow ad$, therefore, C is generating symbol. Since $S \rightarrow aC$, therefore S is also a useful symbol according to production $A \rightarrow bSCa$, A is also generating. Right side of $B \rightarrow aSB$ and $B \rightarrow bBC$ contains B, and B is not terminating so B is not a generating symbol so we can eliminate those productions and grammar becomes

$$S \rightarrow aC$$
$$A \rightarrow bSCa$$
$$C \rightarrow ad$$

Now since S is the start symbol and right side of S does not contain A, hence A is not reachable as :

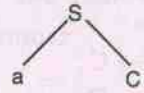


Fig. 7.2.

So by eliminating A we get

$$S \rightarrow aC$$
$$C \rightarrow ad$$

which is reduced grammar equivalent to the given grammar, containing no useless symbol.

7.2.2. Removal of Unit Production

Let us first define unit production :

A production of the form

Non-terminal $\rightarrow$ One non-terminal

that is a production of the form $A \rightarrow B$ (where A and B, both are non-terminals) is called unit production. Unit production increase the cost of derivation in a grammar. Following algorthim can be used to eliminate the unit production.

Algorthim : Removal of unit productions $\Rightarrow$

While (there exist a unit production, $A \rightarrow B$)

{

select a unit production $A \rightarrow B$, such that there exist a production $B \rightarrow \alpha$, where α is a terminal.

For (every non-unit production, $B \rightarrow \alpha$)

Add production $A \rightarrow \alpha$ to the grammar

Eliminate $A \rightarrow B$ from the grammar.

}.

Now let us apply this algorthim on the given CFG's.

Example 7.5. Consider the context free grammar G.

$$S \to AB$$
$$A \to a$$
$$B \to C/b$$
$$C \to D$$
$$D \to E$$
$$E \to a$$

remove the unit production.

Solution. Given CFG

$$S \to AB$$
$$A \to a$$
$$B \to C/b$$
$$C \to D$$
$$D \to E$$
$$E \to a$$

Contain three unit productions

$$B \to C$$
$$C \to D$$
$$D \to E$$

Now to remove unit production $B \to C$, we see if there exists a production whose left side has C and right side contains a terminal (*i.e.* $C \to a$), but there is no such productions in G. Similar things holds for production $C \to D$. Now we try to remove unit production $D \to E$, because there is a production $E \to a$. Therefore, eliminate $D \to E$ and introduce $D \to a$, grammar becomes

$$S \to AB$$
$$A \to a$$
$$B \to C/b$$
$$C \to D$$
$$D \to a$$
$$E \to a$$

Now we can remove $C \to D$ by using $D \to a$, we get

$$S \to AB$$
$$A \to a$$
$$B \to C/b$$
$$C \to a$$
$$D \to a$$
$$E \to a$$

Similarly, we can remove $B \to C$ by using $C \to a$, we obtain

$$S \to AB$$
$$A \to a$$
$$B \to a/b$$
$$C \to a$$
$$D \to a$$
$$E \to a$$

Now it can be easily seen that productions $C \to a$, $D \to a$ $E \to a$ are useless because if we start deriving from S, these productions will never be used. Hence eliminating them gives,

$$S \to AB$$
$$A \to a$$
$$B \to a/b$$

which is completely reduced grammar.

Example 7.6. Consider the following unambiguous expression grammar

$$I \to a/b/Ia/Ib/I0/I1$$
$$F \to I/(E)$$
$$T \to F/T*F$$
$$E \to T/E + T$$

Identified unit production and remove them.

Solution. In the given grammar

$$F \to I$$
$$T \to F$$
$$E \to T$$

are three unit production.

Unit production $F \to I$ can be removed by the help of

$$I \to a/b/Ia/Ib/I0/I1$$

and grammar becomes

$$I \to a/b/Ia/Ib/I0/I1$$
$$F \to (E)/a/b/Ia/Ib/I0/I1$$
$$T \to F/T*F$$
$$E \to T/E + T$$

Now we can eliminate $T \to F$ by the help of

$$F \to (E)/a/b/Ia/Ib/I0/I1$$

and now grammar becomes

$$I \to a/b/Ia/Ib/I0/I1$$
$$F \to (E)/a/b/Ia/Ib/I0/I1$$
$$T \to T*F/(E)/a/b/Ia/Ib/I0/I1$$
$$E \to T/E + T$$

Now let us remove $E \to T$ by the help

$$T \to T*F/(E)/a/b/Ia/Ib/I0/I1$$

and finally, unit production free grammar is

$$I \to a/b/Ia/Ib/I0/I1$$
$$F \to (E)/a/b/Ia/Ib/I0/I1$$
$$T \to T*F/(E)/a/b/Ia/Ib/I0/I1$$
$$E \to E + T/T * F/(E)/a/b/Ia/Ib/I0/I1$$

Example 7.7. Identified and remove the unit productions from following grammar

$$S \to A/bb$$
$$A \to B/b$$
$$B \to S/a$$

Solution. Here unit productions are

$$S \to A$$
$$A \to B$$
$$B \to S$$

We list all unit productions and sequences of unit productions.Then we will handle one unit production or sequence at a time and try to prove generating and reachable for each non-terminal involve in the unit productions, as follows :

$S \to A$	generate $S \to b$
$S \to A \to B$	generate $S \to a$
$A \to B$	generate $A \to a$
$A \to B \to S$	generate $A \to bb$
$B \to S$	generate $B \to bb$
$B \to S \to A$	generate $B \to b$

The new CFG becomes

$$S \to bb/b/a$$
$$A \to b/a/bb$$
$$B \to a/bb/b$$

which hold no unit productions.

7.2.3. Removal of ∈-productions and Nullable Non-terminal

Any context-free language in which ∈ is a word, must have some ∈-productions in its grammar since otherwise we could never derive the word ∈ from S. The statement is obvious, but it should be given some justification, mathematically, this is easy; we observe that ∈-productions are only productions that shorten the working string. If we begin with the string S and apply only non-∈-productions, we never develop a word of length zero.

We now form our attention to the eliminate the productions of the form $A \to \in$, which are called ∈-productions. Surely if ∈ is in $L(G)$, we can not eliminate all ∈-productions from G, but if ∈ is not in $L(G)$, we can eliminate all ∈-productions from G.

"In a given CFC, we call a non-terminal N Nullable if

There is a production $N \to \in$ or

There is a derivation that starts at N and leads to ∈ :

$N \Rightarrow ... \Rightarrow \in$"

To eliminate ∈-productions from a grammar G we use the following technique :

"If $A \to \in$ is a production to be eliminated then we look for all productions, whose right side contains A, and replace each occurrence of A in each of these

productions to obtain the non ϵ-productions. Now these resultant non ϵ-production must be added to the grammar to keep the language generated, the same."

Let us apply this concept :

Example 7.8. Consider the following grammar

$$S \to aA$$
$$A \to b/\epsilon$$

remove the Nullable non-terminal.

Solution. Given CFG is $S \to aA$
$A \to b/\epsilon$

Here $A \to \epsilon$ is ϵ-production, so let us apply above described concept, put ϵ in place of A, at the right side of productions and add the resulted productions to the grammar.

Here is only one productions $S \to aA$ whose right side contains A. So, replacing A by ϵ in this production, we get $S \to a$; now add this new productions **to keep the language generated by this grammar same.** Therefore, the ϵ-free grammar, equivalent to above grammar is

$$S \to aA$$
$$S \to a$$
$$A \to b$$

or

$$S \to aA/a$$
$$A \to b$$

there is no ϵ-production in the above context-free grammar.

Example 7.9. Consider the following grammar G

$$S \to ABAC$$
$$A \to aA/\epsilon$$
$$B \to bB/\epsilon$$
$$C \to c$$

remove the ϵ-production from the above grammar.

Solution. Here we have two ϵ-production to remove $A \to \epsilon$ and $B \to \epsilon$. To eliminate $A \to \epsilon$ from the grammar, the non ϵ-productions to be added are obtained as follows :

List of productions whose right side contain A are :

$$S \to ABAC$$
$$A \to aA$$

So on replacing each occurrence of A by ϵ (one by one), we get four new productions to be added to the grammar :

$$S \to ABC/BAC/BC$$
$$A \to a$$

add these productions to the grammar and eliminate $A \to \epsilon$, to obtain the following grammar

$$S \to ABAC/ABC/BAC/BC$$
$$A \to aA/a$$
$$B \to bB/\epsilon$$
$$C \to c$$

Now to eliminate $B \to \in$, list all productions whose right side contains B, these are :

$$S \to ABAC/BAC/ABC/BC$$
$$B \to bB$$

Replace occurrence of B in each of these productions to obtain non $\in$-productions to be added to the grammar, we get following new productions.

$$S \to AAC/AC/C$$
$$B \to b$$

Add these productions to the grammar and eliminate $B \to \in$ from the grammar, to obtain

$$S \to ABAC/BAC/ABC/BC/AAC/AC/C$$
$$A \to aA/a$$
$$B \to bB/b$$
$$C \to c$$

which is the grammar without any $\in$-production.

Example 7.10. Consider the following grammar and remove $\in$-productions

$$S \to aSa$$
$$S \to bSb/\in$$

Solution. Here $S \to \in$ is the $\in$-production, so let us replace occurrence of S by $\in$ in the following productions

$$S \to aSa$$
$$S \to bSb$$

We will get following new productions

$$S \to aa/bb$$

add these productions in the grammar and remove $S \to \in$ from the grammar

$$S \to aSa/aa/bb/bSb$$

It is $\in$-production free grammar.

Example 7.11. Consider the following grammar

$$S \to a/Xb/aYa$$
$$X \to Y/\in$$
$$Y \to b/X$$

eliminate $\in$-productions.

Solution. Here $X \to \in$ is $\in$-productions. So this problem can also solved in similar fashion and grammar without $\in$-productions will be

$$S \to a/Xb/aYa/b/aa$$
$$X \to Y$$
$$Y \to b/X.$$

Example 7.12. Design a CFG for regular expression

$r = (a + b)^* bb (a + b)^*$, which is free from $\in$-productions.

Solution. Let CFG G be required context free grammar with $\in$-productions

$$S \to XY$$
$$X \to Zb$$

$$Y \to bW$$
$$Z \to AB$$
$$W \to Z$$
$$A \to aA/bA/\in$$
$$B \to Ba/Bb/\in$$

Finally here $A \to \in$ and $B \to \in$ are $\in$-productions and A, B are nullable non-terminals.

The CFG without $\in$-production can be achieved from G in similar fashion, as above discussed. Let it be G', then G' is

$$S \to XY$$
$$X \to Zb/b$$
$$Y \to bW/b$$
$$Z \to AB/A/B$$
$$W \to Z$$
$$A \to aA/bA/a/b$$
$$B \to Ba/Bb/a/b$$

Because $\in$ was not generated by G, the new CFG G' generates exactly the same language.

7.3. Chomsky Normal Form

Definition If a CFG has only production of the form

Non-terminals → string of exactly two non-terminals or of the form

Non-terminals → one terminal

is said to be chomsky normal form or CNF.

7.3.1. Conversion to Chomsky Normal Form

"Conversion of the grammar to CNF can be understood by the following example"

Let us consider a grammar

$G = (\{S, A, B\}, \{a, b\}, P, S)$, where S is the start symbol and P is given by

$$S \to bA/aB$$
$$A \to bAA/aS/a$$
$$B \to aBB/bS/a$$

Now let us find an equivalent chomsky normal form of it.

As we know that right side in CNF either contains two non-terminals or one terminal. So it is clear that in the first production above, we have to replace terminal 'b' by a non-terminal say C_b and 'a' by C_a, hence grammar becomes

$$S \to C_bA/C_aB$$
$$A \to C_bAA/C_aS/a$$
$$B \to C_aBB/C_bS/b$$
$$C_a \to a$$
$$C_b \to b$$

Now first rule is in CNF. In second rule C_bAA is not in CNF. So we can replace AA by a non-terminal say D and similarly BB by E, we obtain

$$S \rightarrow C_bA/C_aB$$
$$A \rightarrow C_bD/C_aS/a$$
$$B \rightarrow C_aE/C_bS/b$$
$$C_a \rightarrow a$$
$$C_b \rightarrow b$$
$$D \rightarrow AA$$
$$E \rightarrow BB$$

The above grammar is equivalent Chomsky normal form of the given grammar.

Example 7.13. Change the following grammar into CNF

$$S \rightarrow 1A/0B$$
$$A \rightarrow 1AA/0S/0$$
$$B \rightarrow 0BB/1$$

Solution. We know that in CNF, either two non-terminal or one terminal is on the right side.

Therefore replace AA by C_a and BB by C_b, 1 by C_1, 0 by C_0, we get

$$S \rightarrow C_1A/C_0B$$
$$A \rightarrow C_1C_a/C_0S/0$$
$$B \rightarrow C_0C_b/1$$
$$C_1 \rightarrow 1$$
$$C_0 \rightarrow 0$$
$$C_a \rightarrow AA$$
$$C_b \rightarrow BB$$

while the CNF form.

Example 7.14. Change the following grammar into CNF

$$G = (\{S\}, \{a, b, c\}, \{S \rightarrow a/b/CSS\}, S).$$

Solution. Here productions are

$$S \rightarrow a$$
$$S \rightarrow b$$
$$S \rightarrow CSS$$

Replace SS by D, we get

$$S \rightarrow a$$
$$S \rightarrow b$$
$$S \rightarrow CD$$
$$D \rightarrow SS$$

which is in CNF form.

Example 7.15. Change the following grammar into CNF

$$S \rightarrow abSb/a/aAb$$
$$A \rightarrow bS/aAAb.$$

Solution. Replace aA by B_a and Ab by B_b, we get

$$S \rightarrow abSb/a/aB_b$$
$$A \rightarrow bS/B_aB_b$$
$$B_a \rightarrow aA$$
$$B_b \rightarrow Ab$$

Now replace ab by C and Sb by D, we get

$$S \rightarrow CD/a/aB_b$$
$$A \rightarrow bS/B_aB_b$$
$$B_a \rightarrow aA$$
$$B_b \rightarrow aA$$
$$B_b \rightarrow Ab$$
$$C \rightarrow ab$$
$$D \rightarrow Sb$$

Now replace a by X_a and b by X_b, we get

$$S \rightarrow CD/a/X_aB_b$$
$$A \rightarrow X_bS/B_aB_b$$
$$B_a \rightarrow X_aA$$
$$B_b \rightarrow X_aB$$
$$C \rightarrow AX_b$$
$$D \rightarrow SX_b$$
$$X_a \rightarrow a$$
$$X_b \rightarrow b.$$

It is required CNF form.

Example 7.16. Convert CFG which is given below into CNF form.

$$S \rightarrow bA/aB$$
$$A \rightarrow bAA/aS/a$$
$$B \rightarrow aBB/bS/b.$$

Solution. Let us replace b by C_b and a by C_a, then CFG becomes

$$S \rightarrow C_bA/C_aB$$
$$A \rightarrow C_bAA/C_aS/a$$
$$B \rightarrow C_aBB/C_bS/b$$
$$C_b \rightarrow b$$
$$C_a \rightarrow a$$

Now let us replace C_bA by D and C_aB by E then grammar becomes as follows :

$$S \rightarrow C_bA/C_aB$$
$$A \rightarrow DA/C_aS/a$$
$$B \rightarrow EB/C_bS/b$$

$$C_b \to b$$
$$C_a \to a$$
$$D \to C_bA$$
$$E \to C_aB$$

Now every production of the grammar is in the CNF form.

Example 7.17. Design a CFG for the language $L = \{a^{4n} \mid n \geq 1\}$ and convert that CFG into CNF form.

Solution. Let CFG for the language $L = \{a^{4n}/n \geq 1\}$ is G

$$G = (V_n, V_t, P, S)$$

where

$$V_n = \{S\}$$
$$V_t = \{a\}$$

and P is defined as

$$S \to aaaaS/aaaa$$

Now let us convert this CFG into CNF form, replace each a by A then CFG becomes

$$S \to AAAAS/AAAA$$
$$A \to a$$

or we can also write

$$S \to AAAAS$$
$$S \to AAAA$$
$$A \to a$$

then CNF form will be

$$S \to AR_1$$
$$R_1 \to AR_2$$
$$R_2 \to AR_3$$
$$R_3 \to AS$$
$$S \to AR_4$$
$$R_4 \to AR_5$$
$$R_5 \to AA$$
$$A \to a.$$

7.4. Greibach Normal Form

Definition For every context free language L without ϵ, there exist a grammar in which every production is of the form $A \to aV$, where 'A' is a variable, 'a' is exactly one terminal and 'V' is the string of none or more variables, clearly $V \in V_n^*$.

"In other words if every production of the context free grammar is of the form $A \to aV/a$, then it is in Greibach Normal Form."

Greibach normal form will be used to construct a push down automata that recognize the language generated by a context free grammar.

To convert a grammar to GNF we start with a production in which the left side has a higher numbered variable than first variable in the right side and make replacements in right side.

Generally we see $A \to A\alpha/\beta$ type production in CFG, we can replace it by following two production :

$$A \to \beta A'$$
$$A' \to \alpha A'/\in, \text{ here } A' \text{ is new non-terminal.}$$

By this way we actually remove left-recursion from the grammar.

In general if a grammar contains productions :

$$A \to A\alpha_1/A\alpha_2/A\alpha_3/.../A\alpha_m/\beta_1/\beta_2/\beta_3/.../\beta_n$$

Then the left recursion can be eliminated by adding the following productions in place of the above productions.

$$A \to \beta_1 A'/\beta_2 A'/\beta_3 A'.../\beta_n A'$$
$$A' \to \alpha_1 A'/\alpha_2 A'/.../\alpha_m A'/\in$$

7.4.1. Conversion of CFG into GNF

We know that every CFL 'L' without $\in$ can be generated by a grammar for which every productions are in Greibach Normal form.

We can convert CFG into GNF by following the number of steps discussed below :

Step 1 : Every CFG must be in CNF form, if not then convert it into CNF. Now rename all variables (V_n) by $A_1, A_2, ..., A_n$ where A_1 is start symbol.

Step 2 : Now modify the grammar so that every production are in following form :

$$\boxed{\begin{aligned} A_i &\to a\gamma \\ A_i &\to A_j\gamma \quad \text{where } j > i \quad \text{and } \gamma \in V_n^* \end{aligned}}$$

or

To get CNF in above discussed format, we should follow the method :

Start with A_1 and proceed to A_n. Suppose upto A_{m-1} the above condition $1 \le i \le m-1$, $A_i \to A_j\gamma$ then $j > i$ is satisfied and for A_m we have production $A_m \to A_j\gamma$ such that $j < m$. To convert A_m production so that it satisfy the above condition we apply the substitute rule.

(that is if P contains productions like $A \to \alpha_1\beta\alpha_2$ and

$$B \to \beta_1 \mid \beta_2 \mid \beta_3 \ldots \beta_n$$

then we can write $A \to \alpha_1\beta_1\alpha_2 \mid \alpha_2\beta_2\alpha_2 \mid \ldots \alpha_1\beta_n\alpha_2)$

Here we can generate new productions by substituting A_j, the right hand side of each A_j production *i.e.* if $A_2 \to A_3A_1 \mid a$ and $A_3 \to A_2A_1 \mid b$ then use A_2 production in R.H.S. of A_3. Now it becomes $A_3 \to A_3A_1 \mid aA_1 \mid b$. By repeating the process at most $m-1$ times, we obtain productions of the form

$$A_m \to A_p\gamma \text{ where } p \ge m.$$

If $m = p$, we have $A_m \to A_m\gamma$, in left recursive form, which are then replaced by method of elimination of left recursion introducing a new variable B_m.

By repeating the step 2 for each variable in G, we have only productions of the form

$$A_i \to A_j\gamma \quad \text{where } j > i$$
$$A_i \to a\gamma \quad a \in V_t$$
$$B_i \to \gamma \quad \text{where } \gamma \in (V_n \cup \{B_1B_2 \ldots B_{i-1}\})^*$$

Step 3. Since A_n is the highest numbered variable then productions are of the form $A_n \to a\gamma$. Now the leftmost symbol of right side of any production for A_{m-1} must be either terminal or A_m (since all productions have been converted $A_i \to A_j$, $j > i$ or $A_i \to a\gamma$ in step 2) replace A_m on right side of production of A_{m-1} by replacement rule.

Now, the leftmost symbol of the right hand side of productions for A_{m-1}, is terminal. Repeat this process for A_{m-2}, $A_{m-3} \ldots A_1$. Now right hand side of each production for an A_i starts with a terminal symbol.

Step 4. In step 2; when we have productions of the form $A_i \to A_i\gamma$, we eliminate left recursion and each time when we eliminate left recursion, we produced new variables B_i. Since in first step we have already converted the grammar into CNF and on elimination method of left recursion (if $A \to A\alpha \mid \beta$ we convert it into $A \to \beta B \mid \beta$, $B \to \alpha B \mid \alpha$), we cannot find new variable B_i in right hand side as leftmost symbol of right hand side. Thus, after step 3 every production of $A_i (1 \le i \le n)$ is in the form $A_i \to a\gamma$ or $A_i \to A_j A_k$. Again, in production $B_i (B_i \to \alpha, B_i \to \alpha B_i)$, all B_i productions have right hand side beginning with terminals or variable A_i's. So, no B_i production can start with another B_j. Finally, we apply the substitution rule and get the grammar which is in GNF.

In the above steps, it is clear to see that we start with grammar in CNF and then we apply substitution rule and elimination of left recession.

Example 7.18. Convert the grammar.

$$S \to AB \mid BC$$
$$A \to aB \mid bA \mid a$$
$$B \to bB \mid cC \mid b$$
$$C \to c$$

into GNF.

Solution. Here the production $S \to AB \mid BC$ is not in GNF. On applying the substitution rule we immediately get equivalent grammar.

$$S \to aBB \mid bAB \mid aB \mid bBC \mid cCC \mid bC$$

which is in GNF.

Example 7.19. Convert the grammar.

$$S \to abaSa \mid aba$$

into GNF.

Solution. If we introduce new variables A and B and productions as $A \to a$, $B \to b$ and substitute into given grammar as

$$S \to aBASA \mid aBA$$
$$A \to a, B \to b$$

which is in GNF.

Example 7.20. Convert the grammar

$$S \rightarrow AB$$
$$A \rightarrow BS \mid a$$
$$B \rightarrow SA \mid b$$

into GNF.

Solution. Step 1 : The given grammar is in CNF. Let us change the name of variables as A_1, A_2, A_3 for S, A, B respectively. Now the given productions are in the form (with A_1 as start symbol)

$$A_1 \rightarrow A_2A_3$$
$$A_2 \rightarrow A_3A_1 \mid a$$
$$A_3 \rightarrow A_1A_2 \mid b.$$

Step 2 : In this step we need productions must be in the form that the R.H.S. of productions must starts with terminal or with higher numbered variable. The productions.

$$A_1 \rightarrow A_2A_3$$
$$A_2 \rightarrow A_3A_1 \mid a$$
$$A_3 \rightarrow b$$

are in required form. But $A_3 \rightarrow A_1A_2$ is not. Now we want to convert $A_3 \rightarrow A_1A_2$ into required form.

By applying substitution in R.H.S. for A_1, we get $A_3 \rightarrow A_2A_3A_2$

Again applying substitution rule for A_2, we get

$$A_3 \rightarrow A_3A_1A_3A_2 \mid aA_3A_2$$

Thus, we get :

$$A_1 \rightarrow A_2A_3$$
$$A_2 \rightarrow A_3A_1 \mid a$$
$$A_3 \rightarrow A_3A_1A_3A_2 \mid aA_3A_2 \mid b$$

Eliminating the left recursion from A_3 production, we get

$$A_3 \rightarrow aA_3A_2B_3 \mid bB_3 \mid aA_3A_2 \mid b$$

and

$$B_3 \rightarrow A_1A_3A_2B_3 \mid A_1A_3A_2$$

where B_3 is a new variable introduced.

Thus, we have set of productions as :

$$A_1 \rightarrow A_2A_3$$
$$A_2 \rightarrow A_3A_1 \mid a$$
$$A_3 \rightarrow aA_3A_2B_3 \mid bB_3 \mid aA_3A_2 \mid b$$
$$B_3 \rightarrow A_1A_3A_2B_3 \mid A_1A_3A_2$$

Step 3 : Now all the A_3 productions start with terminals. With the help of substitution rule we can replace A_3 in R.H.S. of A_2 production. We get

$$A_2 \rightarrow aA_3A_2B_3A_1 \mid bB_3A_1 \mid aA_3A_2A_1 \mid bA_1 \mid a.$$

Now, all A_2 productions start with terminal. Again we replace A_2 from $A_1 \rightarrow A_2A_3$, we have

$$A_1 \rightarrow aA_3A_2B_3A_1A_3 \mid bB_3A_1A_3 \mid aA_3A_2A_1A_3 \mid bA_1A_3 \mid aA_3$$

The resulting set of productions are the following :

$$\begin{aligned}
A_1 &\rightarrow aA_3A_2B_3A_1A_3 \mid bB_3A_1A_3 \mid aA_3A_2A_1A_3 \mid bA_1A_3 \mid aA_3 \\
A_2 &\rightarrow aA_3A_2B_3A_1 \mid bB_3A_1 \mid aA_3A_2A_1 \mid bA_1 \mid a \\
A_3 &\rightarrow aA_3A_2B_3 \mid bB_3 \mid aA_3A_2 \mid b \\
B_3 &\rightarrow A_1A_3A_2B_3 \mid A_1A_3A_2.
\end{aligned}$$

Step 4 : In step 3 only B productions are not in GNF. Applying substitution rule in production

$$B_3 \rightarrow A_1A_3A_2B_3$$

We get

$$\begin{aligned}
B_3 \rightarrow\ & aA_3A_2B_3A_1A_3A_3A_2B_3 \mid bB_3A_1A_3A_3A_2B_3 \mid aA_3A_2A_1A_3A_3A_2B_3 \\
& \mid bA_1A_3A_3A_2B_3 \mid aA_3A_3A_2B_3
\end{aligned}$$

And from $B_3 \rightarrow A_1A_2A_3$, we get

$$\begin{aligned}
B_3 \rightarrow\ & aA_3A_2B_3A_1A_3A_3A_2B_3 \mid bB_3A_1A_3A_3A_2B_3 \mid aA_3A_2A_1A_3A_3A_2A_3 \\
& \mid bA_1A_3A_3A_2B_3 \mid aA_3A_3A_2
\end{aligned}$$

The final sets of productions are

$$\begin{aligned}
A_1 &\rightarrow aA_3A_2B_3A_1A_3 \mid bB_3A_1A_3 \mid aA_3A_2A_1A_3 \mid bA_1A_3 \mid aA_3 \\
A_2 &\rightarrow aA_3A_2B_3A_1 \mid aB_3A_1 \mid aA_3A_2A_1 \mid bA_1 \mid a \\
A_3 &\rightarrow aA_3A_2B_3 \mid bB_3 \mid aA_3A_2 \mid \mathrm{b} \\
B_3 &\rightarrow aA_3A_2B_3A_1A_3A_2A_3B_3 \mid bB_3A_1A_3A_2A_3B_3 \mid aA_3A_2A_1A_3A_2A_3B_3 \\
&\quad \mid bA_1A_3A_2A_3B_3 \mid aA_3A_2A_3B_3 \mid aA_3A_2B_3A_1A_3A_2A_3 \\
&\quad \mid bB_3A_1A_3A_2A_3 \mid aA_2A_3A_1A_3A_2A_3 \mid bA_1A_3A_2A_3 \mid aA_3A_2A_3
\end{aligned}$$

In which all productions are in GNF.

Example 7.21. Consider the following grammar and write its equivalent GNF

$$\begin{aligned}
S &\rightarrow AB \\
A &\rightarrow aA/bB/b \\
B &\rightarrow b.
\end{aligned}$$

Solution. We know that in the GNF each production should be of the $A \rightarrow aV/a$ form where $V \in V_n^*$.

We can replace A in $S \rightarrow AB$ by aA, or bB or b.

$$\begin{aligned}
S &\rightarrow aAB/bBB/bB \\
A &\rightarrow aA/bB/b \\
B &\rightarrow b
\end{aligned}$$

which is in Greibach normal form.

Example 7.22. Convert the following grammar

$$S \to abSb/aa$$

into GNF.

Solution. Here we can use a method similar to the one introduced in the construction of Chomsky normal form. We introduce new variables A and B that are essentially synonyms for a and b, respectively, substituting for the terminals with their associated variables leads to the equivalent grammar.

$$S \to aBSB/aA$$
$$A \to a$$
$$B \to b$$

which is in GNF.

Example 7.23. Convert the following grammar into GNF

$$S \to aAS$$
$$S \to a$$
$$A \to SbA$$
$$A \to SS$$
$$A \to ba.$$

Solution. Let us introduced new non-terminals as B, C and D, then GNF will be

$$A \to aBC$$
$$A \to aBS$$
$$A \to aC$$
$$A \to aS$$
$$A \to bD$$
$$B \to aBCS$$
$$B \to aBSS$$
$$B \to aBSS$$
$$B \to aCS$$
$$B \to aSS$$
$$B \to bDS$$
$$C \to bA$$
$$S \to aB$$
$$S \to a$$
$$D \to a.$$

7.5. REGULAR GRAMMARS

A regular grammar may be left linear or right linear.

Definition "If all production of a CFG are of the form $A \to wB$ or $A \to w$, where A and B are variables and $w \in V_t^*$, then we say that grammar is right liner".

Definition "If all the production of a CFG are of the form $A \to Bw$ or $A \to w$, we call it left linear."

A right or left linear grammar is called a regular grammar.

Example 7.24. Write the left linear and right linear grammar for the regular expression

$$r = 0(10)^*.$$

Solution. Given regular expression $r = 0(10)^*$, it means any string which starts with 0 followed by any number of 10's is in regular expression.

Left Linear. Let grammar be $G = (V, V_t, P, S)$

here
$$V_n = \{S\}$$
$$V_t = \{0, 1\}$$

and P is defined as

$$S \to S10/0$$

Right Linear. Let grammar be $G' = (V_n, V_t, P, S)$

where
$$V_n = \{S, A\}$$
$$V_t = \{0, 1\}$$

and P is defined as

$$S \to 0A$$
$$A \to 10\ A/\in.$$

Example 7.25. Write the right linear and left linear regular grammar for the regular expression $r = (ab)^* a$.

Solution. Let us analyse the language of this regular expression, clearly it contains set of strings in which every string start with any number of ab's and end with single a.

Right Linear. Let grammar be $G_1 = (\{S\}, \{a, b\}, S, P)$

where P_1 is defined as follows :

$$S \to abS/a$$

Left Linear. Let grammar be $G_2 = (\{S, S_1, S_2\}, \{a, b\}, S, P_2)$

with production

$$S \to S_1 a$$
$$S_1 \to S_1 ab/\in$$

is left-linear. Both G_1 and G_2 are regular grammars.

7.5.1. Equivalence of Regular Grammars and Finite Automata

The regular grammars characterize the regular sets, in the sense that a language is regular if and only if it has a left-linear grammar or if and only if it has a right linear grammar. These results are proved in the next two theorems.

Theorem 7.1. If L has a regular grammar, then L is a regular set ?

Proof. First, suppose $L = L(G)$ for some right-linear grammar

$$G = (V_n, V_t, P, S).$$

Let us construct an NFA with ϵ-moves, $M = (Q, V_t, \delta, \{S\}, \{\epsilon\})$ that simulates derivates derivation in G.

Consists of the symbols $[\alpha]$ such that α is S or a suffix of some right hand side of a production in P, we define δ by :

(*a*) If A is a variable, then $\delta([A], \epsilon) = \{[\alpha]/A \to \alpha$ is a production$\}$.

(*b*) If a is in V_t and α in $V_t^* \cup V_t^* V$, then

$$\delta([a\alpha], a) = \{[\alpha]\}.$$

Then it is easy to show that $\delta([S], w)$ contains $[\alpha]$ if and only if $S \overset{*}{\Rightarrow} xA \Rightarrow xy\alpha$, where $A \to y\alpha$ is a production and $xy = w$, or if $\alpha = S$ and $w = \epsilon$. As $[\epsilon]$ is the unique final state, M accepts w if and only if $S \overset{*}{\Rightarrow} xA \Rightarrow w$. But since every derivation of a terminal string has atleast on step, we see that on accepts w if and only if G generates w. Hence every right-linear grammar generates a regular set.

Theorem 7.2. If L is a regular set, then L is generated by some left-linear grammar and by some right-linear grammar.

Proof. Let $L = L(M)$ for DFA $M = (Q, \Sigma, \delta, q_0, F)$. First suppose that q_0 is not a fixed state. Then $L = L(a)$ for right linear grammar $G = (Q, \Sigma, P, q_0)$, where P consists of production $P \to aq$ whenever $\delta(P, a) = q$ and also $P \to a$ whenever $\delta(P, a)$ is a final state. Then clearly $\delta(P, w) = q$ if and only if $P \overset{*}{\Rightarrow} wq$. If wa is accepted by M, let $\delta(q_0, w) = P$, implying $q_0 \overset{*}{\Rightarrow} wP$. Also $\delta(P, a)$ is final, so $P \to a$ is a production. Thus $q_0 \overset{*}{\Rightarrow} wa$. Conversely, let $q_0 \overset{*}{\Rightarrow} x$. Then $x = wa$ and $q_0 \overset{*}{\Rightarrow} wP \Rightarrow wa$ for some state (variable) P. Then $\delta(q_0, w) = P$, and $\delta(P, a)$ is final. Then x is in $L(M)$. Hence $L(M) = L(G) = L$.

Now let q_0 be in F, so ϵ is in L. We note that the grammar G defined above generates $L - \{\epsilon\}$. We may modify G by adding a new start symbol S with productions $S \to q_0/\epsilon$. The resulting grammar is still right-linear and generates L.

To produce a left-linear grammar for L, start with an NFA for L^R and then reverse the right sides of all productions of the resulting right-linear grammar.

7.5.2. Conversion of Regular Grammar into Finite Automata

Given a regular grammar G, a finite automata accepting $L(G)$ can be obtained as follows :

(*a*) The number of states in the automata will be equal to the number of non-terminals plus one. Each state in automata represents each non-terminal in the regulr grammar. The additional state will be the final state of the

automata. The state corresponding to the start symbol of the grammar will be the initial state of automata. If $L(G)$ contains ϵ that is start symbol in grammar derives to ϵ, then make start state also as final state.

(*b*) The transitions for automata are obtained as follows

(*i*) For every production $A \to aB$ make $\delta(A, a) = B$ that is make an are labelled '*a*' from A to B.

(*ii*) For every production $A \to a$ make $\delta(A, a)$ = final state.

(*iii*) For every production $A \to \epsilon$, make $\delta(A, \epsilon) = A$ and A will be final state.

Example 7.26. Consider the following grammar

$$S \to 0A/1B/0/1$$
$$A \to 0S/1B/1$$
$$B \to 0A/1S$$

Find the automation for this grammar.

Solution. The automation for this grammar will have four states : S, A, B and one additional say C, which will be the final state.

Step 1. For production $S \to 0A$ and $S \to 1B$

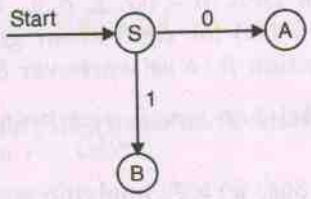


Fig. 7.3.

Step 2. For production $S \to 0/1$

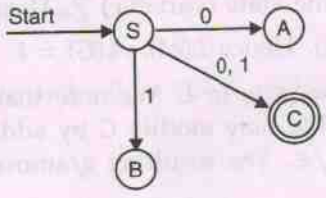


Fig. 7.4.

Step 3. For production $A \to 0S$

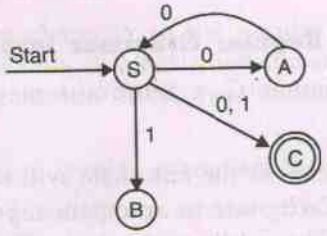


Fig. 7.5.

Step 4. For the production $A \rightarrow 1B$

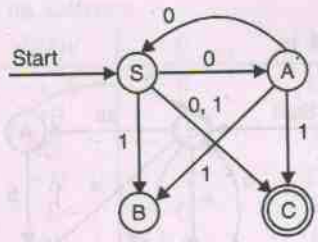


Fig. 7.6.

Step 5. For the production $B \rightarrow 0A$ and $B \rightarrow 1S$

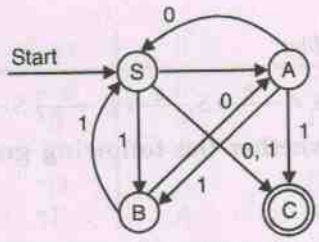


Fig. 7.7.

There is no need to show all the steps every time and FA can be generated directly.

Example 7.27. Give the automation for the following grammar :

$$S \rightarrow 0S/1A/1$$
$$A \rightarrow 0A/1A/0/1.$$

Solution. The automation for the above grammar will be

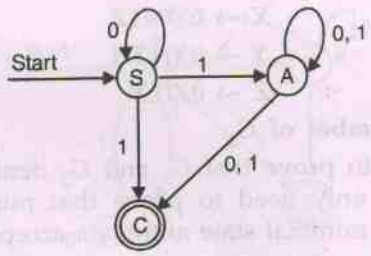


Fig. 7.8.

Example 7.28. Consider the regular grammar $G = (V_n, V_\epsilon, P, S)$, where $V_n = \{A, B, S\}$, $V_t = \{a, b\}$ and P is defined as

$$S \rightarrow abA$$
$$S \rightarrow B$$
$$S \rightarrow baB$$
$$S \rightarrow \epsilon$$
$$A \rightarrow bS$$
$$B \rightarrow aS$$
$$A \rightarrow b$$

Construct a NFA M such that $L(M) = L(G)$. Trace the transitions of M to accept the string *abba*.

Solution. The NFA M is

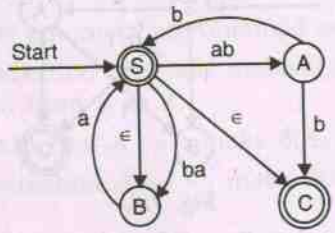


Fig. 7.9.

Here C is final state.

Given string B W = *abba*

$$S \xrightarrow{ab} A \xrightarrow{b} S \xrightarrow{\epsilon} B \xrightarrow{a} S \xrightarrow{\epsilon} C$$

Example 7.29. Find whether the following grammars denote the same language.

Grammar G_1 is

$$A \rightarrow 0B/1E$$
$$B \rightarrow 0A/1F/\epsilon$$
$$C \rightarrow 0C/1A$$
$$D \rightarrow 0A/1D/\epsilon$$
$$E \rightarrow 0C/1A$$
$$F \rightarrow 0A/1B/\epsilon$$

where A is the start symbol for G_1.

Grammar G_2 is :

$$X \rightarrow 0Y/0/1Z$$
$$Y \rightarrow 0X/1Y/1$$
$$Z \rightarrow 0Z/1X$$

where X is the start symbol of G_2.

Solution. In order to prove that G_1 and G_2 denote the same language, *i.e.*, $L(G_1) = L(G_2)$, we only need to prove that minimum state automata accepting $L(G_1)$ and the minimal state automata accepting $L(G_2)$ are identical.

The finite state automata for grammar G_1 is given below :

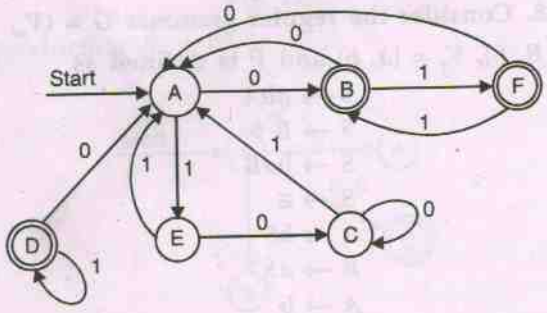


Fig. 7.10.

This automation shown in Fig. 7.10 is deterministic. To minimise this, we write it in tabular forms as follows :

State	0	1
→ A	B	E
*B	A	F
C	C	A
*D	A	D
E	C	A
*F	A	B

Here, two sets of states (final and non-final) are {A, C, E} and {B, D, F}.

Since the output C and E are same, hence replacing every occurrence of E by C, we get

State	0	1
→ A	B	C
*B	A	F
C	C	A
*D	A	D
*F	A	B

and now the two sets become {A, C} and {B, D, F}.

Now since in set {B, D, F} each state enters in a final state 'A' at input 0 and enters in a final state at input 1. Hence they can be reduced as follows :

State	0	1
→ A	B	C
*B	A	B
C	C	A

Similarly automation for grammar G_2 is

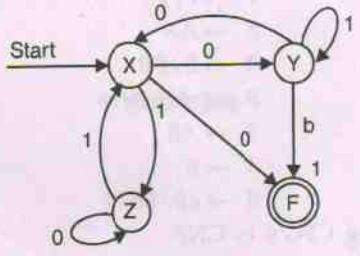


Fig. 7.11.

Above automata is not DFA, so convert it to DFA as follows :

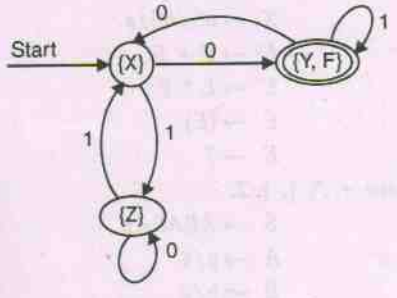


Fig. 7.12.

To minimize it, write it in tabular form

State	0	1
$\rightarrow \{X\}$	$\{Y, F\}$	$\{Z\}$
$*\{Y, F\}$	$\{X\}$	$\{Y, F\}$
$\{Z\}$	$\{Z\}$	$\{X\}$

It is already minimised.

It is clear that these two automation for G_1 and G_2 are similar, hence G_1 and G_2 represent the same language.

EXERCISE

1. Each of the following CFG's has a production using the symbol ϵ and yet ϵ is not a word in the language. Using the method, discussed in this chapter, show that there are other CFGs for these languages that do not use ϵ-productions.

(*i*)
$$S \rightarrow aX/bX$$
$$X \rightarrow a/b/\epsilon$$

(*ii*)
$$S \rightarrow aX/bS/a/b$$
$$X \rightarrow aX/a/\epsilon$$

(*iii*)
$$S \rightarrow aS/bX$$
$$X \rightarrow aX/\epsilon$$

(*iv*)
$$S \rightarrow XaX/bX$$
$$X \rightarrow XaX/XbX/\epsilon$$

2. Each of the following CFG's has unit productions. Using the algorthim presented in this chapter, find CFG's for these same languages that do not have unit productions.

(*i*)
$$S \rightarrow aX/Yb$$
$$X \rightarrow S$$
$$Y \rightarrow bY/b$$

(*ii*)
$$S \rightarrow AA$$
$$A \rightarrow B/BB$$
$$B \rightarrow abB/b/bb$$

(*iii*)
$$S \rightarrow AB$$
$$A \rightarrow B$$
$$B \rightarrow aB/Bb/\epsilon$$

3. Convert the following CFG's to CNF :

(*i*)
$$S \rightarrow SS/a$$

(*ii*)
$$S \rightarrow aSa/SSa/a$$

(*iii*)
$$S \rightarrow aXX$$
$$X \rightarrow aS/bS/a$$

(*iv*)
$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow (E)$$
$$E \rightarrow 7$$

The terminal here are +, *, (,), 7.

(*v*)
$$S \rightarrow ABABAB$$
$$A \rightarrow a/\epsilon$$
$$B \rightarrow b/\epsilon$$

Note that $\in$ is the word in this language, but when converted into CNF, the grammar will no longer generate it.

(vi) $S \rightarrow SaS/SaSbS/SbSaS/\in$

(vii) $S \rightarrow AS/SB$

$A \rightarrow BS/SA$

$B \rightarrow SS$

4. Convert the following CFG's with unit production into CNF :

(i) $S \rightarrow X$

$X \rightarrow Y$

$Y \rightarrow Z$

$Z \rightarrow aa$

(ii) $S \rightarrow SS/A$

$A \rightarrow SS/AS/a$

5. Construct a DFA that accepts the language generated by the grammar

$S \rightarrow abA$

$A \rightarrow baB$

$B \rightarrow aA/bb.$

6. Find the regular grammar that generates the language $L(aa^* (ab + a)^*)$.

7. Construct the left linear grammar and right-linear grammar for the language

$L = \{a^n b^m : n \geq 2, m \geq 3\}$.

8. Find a regular grammar that generates the language on $\Sigma = \{a, b\}$ consisting of all strings with no more than a's.

9. Find a regular grammar for the language

$L = \{a^n b^m : n + m \text{ is even}\}$.

10. Find regular grammars for the following languages on $\{a, b\}$.

(a) $L = \{w : n_a(w) \text{ and } n_b(w) \text{ are both even}\}$

(b) $L = \{w : (n_a(w) - n_b(w)) \bmod 3 = 1\}$.

*****

Chapter 8

Pushdown Automata

8.1. INTRODUCTION

Inside This Chapter

The finite automata that we have studied earlier are not capable to recognise the context free languages such as $\{W\ C\ W^R / W \in \Sigma^*\}$. Because by reading a string in this language from left to right, automata must "remember" the string before it encounters symbol 'C' and then compare it to the string after 'C'. But our finite automata is not capable to remember anything.

From above example it becomes clear that the finite automata can be extended by addition of an auxiliary storage to accept context free languages.

TIPS

The pushdown automata is essentially a finite automata with control of both an input tape and a stack to store what it has read. A stack is a "first in last out" list, that is, symbols can be entered or removed only at the top of the list. When a new symbol is entered at the top, the symbol previously at the top becomes second and so on.

A familiar example of the stack is the set of plates that we see in cafeteria. We can remove plates only from the top of this set (stack) not from the middle. Similarly we can add plates only at the top of the stack not in the middle.

We shall now formalise the concept of a PDA. The PDA will have 3 things : an input tape, a finite control, and a stack. The device will be non-deterministic, having some finite number of moves in each situation.

We can also see pictorial representation of pushdown automata.

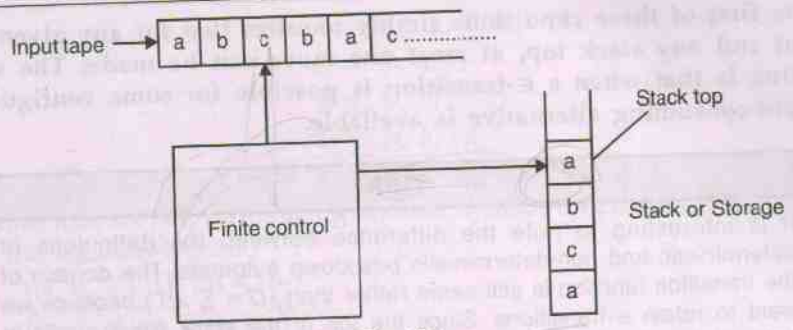


Fig. 8.1.

Here is input tape, from which finite control (which works like finite control of finite automata), reads the input, and same time it reads the symbol from stack top. Now it depends on finite control, that what is the next state and what will happen with stack top.

8.2. Definition of Pushdown Automata

Definition A pushdown automata is a system, which is mathematically defined as follows :

$$P = (Q, \Sigma, \Gamma, \delta, s, F)$$

where Q : non-empty finite set of states

Σ : non-empty finite set of input symbols

Γ : is finite set of pushdown symbols

s : is the initial state, $s \in Q$

F : is the set of final states and $F \subseteq Q$

δ : it is a transition function or transition relation which maps.

$$\delta (Q \times \Sigma^* \times \Gamma^*) \rightarrow (Q \times \Gamma^*)$$

As we have already seen in case of automation, that it may be nondeterministic or deterministic, pushdown automata also behaves in similar fashion.

"If it follows all the properties of non-determinision then it is said to be non-deterministic pushdown automata".

Clearly for the "NPDA" transition relation must behave as follows :

$$(Q \times (\Sigma \cup \{\in\} \times \Gamma^*) \rightarrow \text{finite subset of } (Q \times \Gamma^*).$$

TIPS

From the above relation it becomes very clear that $\in$-transitions are allowed in case of non-deterministic pushdown automata. Similar to the non-deterministic finite automata, non-deterministic pushdown automata also have the power of guess.

Now let us analyse the case of deterministic pushdown automata.

"A pushdown automata $P = (Q, \Sigma, \Gamma, \delta, S, F)$ is said to be deterministic if it is an automation as defined in definition of PDA, subject to the restrictions that for every $q \in Q$, $a \in (\Sigma \cup \{\in\})$ and $b \in \Gamma$.

(*a*) $\delta(q, a, b)$ contains at most one element,

(*b*) if $\delta(q, \in, b)$ is not empty, then $\delta(q, c, b)$ must be empty for every $c \in \Sigma$.

The first of these conditions simply requires that for any given input symbol and any stack top, at most one move can be made. The second condition is that when a $\in$-transition is possible for some configuration, no input-consuming alternative is available.

TIPS

It is interesting to note the difference between the definitions of deterministic and non-deterministic pushdown automata. The domain of the transition function is still same rather than $(Q \times \Sigma \times \Gamma)$ because we want to retain $\in$-transitions. Since the top of the stack plays a role in determining the next move, the presence of $\in$-transition does not automatically simply non-determinism. **Also some transition of DPDA may be to the empty set, that is undefined, so there may be dead configurations.** This is not going to affect the definition; the only criterian for determinism is that all the times at most one possible move exists.

8.2.1. Explanation for the Moves of Pushdown Automata

Let us see some moves of PDA in the form of mapping, that

$$(Q \times \Sigma^* \times \Gamma^*) \rightarrow (Q \times \Gamma^*).$$

(*a*) The interpretation of $((q, a, z), (p, y)) \in \delta$ (where $p, q, \in Q$, a is an alphabet, z and y in Γ^*) is that PDA whenever is in state q, with z on the top of the stack may read 'a' from the input tape, replace z by y on the top of the stack and enter state P.

(*b*) To push a symbol on the stack, just add it on the top of the stack. This can be achieved by the transition $((q, a, \in), (p, a))$.

(*c*) Similarly to pop a symbol is to remove it from the top of the stack. The transition $((q, a, z), (p, \in))$ pops z from the stack.

(*d*) PDA can also behave as do nothing machine, just read the input from the input and don't make any change in the state and symbol at the stack.

Let PDA in state P and it has any symbol say x on the stack top, now it read a input 'a' from the input tape then write a move for PDA which behaves as do nothing machine.

$$((P, a, z), (p, z))$$

Clearly PDA just read input 'a', and without making any change, the state or at stack top, waits for another input from input tape.

Now I think we are able to handle some practical problems :

Example 8.1. Design a PDA which accepts the language

$L = \{w \in \{a, b\}^* / w$ **has the equal number of** a**'s and** b**'s}.**

Solution. To solve any problem, it becomes very important that we should be able to analyse the language very clearly.

Here Language $L = \{w \in \{a, b\}^* / w$ has equal number of a's and b's$\}$ is having a set of string in which every string is having equal number of a's and b's, there is no concern of the position of a's and b's, the machine can read input 'a' first or input b's first, a after b, b after a, a followed by a or b followed by b and of course null string is also accepted.

Now let us assume that PDA be P.

$$P = (Q, \Sigma, \Gamma, \delta, S, F)$$

where $Q = \{s, q, f\}$

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

$\Gamma = \{a, b, c\}$

$F = \{f\}$ δ is follows.

(Note that we are introducing another symbol other than Σ that is 'c' to make the solution logical.)

1. $((s, \in, \in), (q, c))$
2. $((q, a, c), (q, ac))$
3. $((q, a, a), (q, aa))$
4. $((q, a, b), (q, \in))$
5. $((q, b, c), (q, bc))$
6. $((q, b, b), (q, bb))$
7. $((q, b, a), (q, \in))$
8. $((q, \in, c), (f, \in))$

Here we are using symbol c to mark the bottom of the stack. PDA intially pushes 'c' in to the stack (using transition 1). Now if PDA reads symbol 'a' from the input tape and if stack is empty or has 'a' on the stack top then push 'a' in to the stack (by using transition 2 and 3). Now if it reads 'a' from input and 'b' is on the top of the stack then it simply pops the symbol 'b' and pushes nothing. (This is achieved by transition 4) same reason holds true for symbol 'b'. When it has read all the input and finds empty symbol 'c' on the top of the stack, it enters in state f, and pop the symbol c. **So final state is defined as, PDA is in state f and stack is empty.**

Take a string *abab* for example. Now if we want to check whether it contains equal number of a's and b's, we simply store a's or b's whatever we read. Now when we read next a, we delete a corresponding 'b' from our store. The same process follows for b's. If at the end we have nothing to read and nothing in store then we can say that the string contains equal number of a's and b's. Following table shows the operation of PDA on the string *abbbaaba*.

S. No.	State	Unread input	Stock	Transition used
1	s	*abbbaaba*	$\in$	-
2	q	*abbbaaba*	c	1
3	q	*bbbaaba*	ac	2
4	q	*bbaaba*	c	7
5	q	*baaba*	bc	5
6	q	*aaba*	bbc	6
7	q	*aba*	bc	4
8	q	*ba*	c	4
9	q	*a*	bc	5
10	q	$\in$	c	4
11	f	$\in$	$\in$	8

This problem can also be solved without introducing the symbol 'c' at the stack bottom, let us see another solution.

Let PDA be P_1

$$P_1 = (Q, \Sigma, \Gamma, \delta, S, F)$$

where

$$Q = \{s, q, f\}$$
$$\Sigma = \{a, b\}$$
$$\Gamma = \{a, b\}$$
$$F = \{f\}$$

and transition relation δ is defined as follows :

1. $((s, a, \in), (q, a))$
2. $((s, b, \in), (q, b))$
3. $((q, a, a), (q, aa))$
4. $((q, b, b), (q, bb))$
5. $((q, a, b), (q, \in))$
6. $((q, b, a), (q, \in))$
7. $((q, \in, \in), (f, \in))$

It will also work.

Example 8.2. Design PDA for the language $L = \{wcw^r / w \in \{a, b\}^*\}$.

Solution. By the analysis of the language it becomes clear that $ababcbaba \in L$, but $abc\ ab \notin L$, and $cbc \notin L$.

Let pushdown automata be P

$$P = (Q, \Sigma, \Gamma, \delta, S, F)$$

where

$$Q = \{s, f\}$$
$$\Sigma = \{a, b, c\}$$
$$\Gamma = \{a, b\}$$

$F = \{f\}$ and transition relation δ is defined as follows :

(1) $((s, a, \in), (s, a))$
(2) $((s, b, \in), (s, b))$
(3) $((s, c, \in), (f, \in))$
(4) $((f, a, a), (f, \in))$
(5) $((f, b, b), (f, \in))$.

This automation operates in the following way. As it reads the first half of its input, if remains its in intial state and uses transition 1 and 2 to transfer symbols from the input string on to the pushdown store. **Note that these transitions are applicable regardless of the current content of the stack.** Since the "string" to be matched on the top of the pushdown store is the empty string, when machine sees *a* '*c*' in the input string, it switches from state *s* to state *f* without operating on the stack. Thereafter only transition 4 and 5 are operative; these permit the removal of the top symbol on the stack, provided that it is the same as the next input symbol. If the input symbol does not match the top symbol on the stack, no further operation is possible. If the automation reaches in this way the configuration $(f, \in, \in)$. Final state, end of input, empty stack then the input was indeed of the form $wc\ w^R$, and the automation accepts on the other hand if automation or if the input is exhausted before the stack is emptied, then it does not accept.

The operation of P can be seen by following table, we have taken the string *abacaba* for example

S.No.	State	Unread input	Stack	Transition used
1	s	abacaba	∈	–
2	s	bababa	a	1
3	s	acaba	ba	2
4	s	caba	aba	1
5	f	aba	aba	3
6	f	ba	ba	4
7	f	a	a	5
8	f	∈	∈	4

Example 8.3. Construct a push down automata to accept the following language

$$L = \{ww^R : w \in \{a, b\}^*\}.$$

Solution. By the analysis of given language it becomes very clear that the strings accepted by the machine are the same as those accepted by the machine of the previous example, except that the symbol c that marked the center of the string is missing. **Therefore, the machine must "gess" when it has reached the middle of the input string and change from state 's' to state 'f' in a non-deterministic fashion.**

Let PDA be $P = (Q, \Sigma, \Gamma, \delta, S, F)$

where $Q = \{s, f, f'\}$

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

$F = \{f'\}$

and transition relation δ is defined as follows :

(1) $((s, a, \in), (S, a))$

(2) $((s, b, \in), (S, b))$

(3) $((s, \in, \in), (f, \in)) \longrightarrow$ (middle of the input string)

(4) $((f, a, a), (f, \in))$

(5) $((f, b, b), (f, \in))$

(6) $((f, \in, \in), (f', \in))$

Clearly whenever the machine is in state 's' it can non-deterministically choose either to push the next input symbol on the stack or to switch to state f without consuming any input. Move 6 is for accepting the string.

Example 8.4. Design a PDA for the following language.

$$L = \{a^n b^n : n > 0\}.$$

Solution. By the analysis of the language it is clear that PDA for the given language will accept the set of strings in which every string starts with 'a' followed by any number of a's and followed by same number of b's, clearly 'a' should not be encountered whenever b is encountered strings like *ab, aabb, aaabbb* ... should be accepted and $\in$, *aa, bb, aabbb* and *bbaa* should be rejected.

Let push down automata be P

$$P = (Q, \Sigma, \Gamma, \delta, S, F)$$

where $Q = \{s, f, q\}$
$F = \{f\}$
$\Sigma = \{a, b\}$
$\Gamma = \{a\}$

and transition relation δ is defined as follows :

(1) $((s, a, \in), (S, a))$
(2) $((s, a, a), (S, aa))$
(3) $(s, b, a), (q, \in))$
(4) $((q, b, a), (q, \in))$
(5) $((q, \in, \in), (f, \in))$

Initially machine is at state s, and it can read input a then pushes all a's encountered before any 'b' to the stack (by using transition relation on 1 and 2). When first 'b' is encountered, corresponding 'a' is poped up from the stack and now state becomes q, and this process can be repeat with the help of transition relation number '4'. Finally when there is no input an stack is empty PDA charges its state from q to final state 'f'.

Example 8.5. In Fig. 8.2 a DFA is given, write the regular expression for it and design a pushdown automata

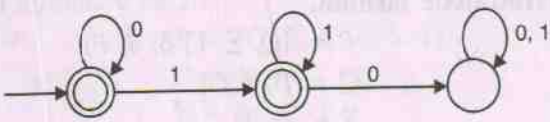


Fig. 8.2.

Solution. By the observation of DFA defined by the transition diagram in Fig. 8.2, we can write the regular expression for it, let it is r then

$$r = 0^* 1^*$$

Clearly all the strings which start with any number of 0's followed by any number of 1's and there is no '0' after 1 is encountered is the string of regular set.

Let pushdown automata be P

$P = (Q, \Sigma, \Gamma, \delta, s, F)$
$Q = \{s, p, r\}$
$\Sigma = \{0, 1\}$
$\Gamma = \{\in\}$
$F = \{s, P\}$

and the transition relations δ are defined as follows :

(1) $((s, \in, \in), (s, \in))$
(2) $(s, 0, \in), (s, \in))$
(3) $((s, 1, \in), (p, \in))$
(4) $(p, 1, \in), (p, \in))$
(5) $((p, 0, \in), (r, \in))$

Now let us see the explanation of transitions, initial state of machine is s,

it is also final state so $\in$ is accepted. Now any number of 0's can be accepted (by the help of transition relation 2). As soon as 1 is encountered state s is changed to p from s (by the help of transition 3). Now any number of 1 can be accepted by the machine with the help of transition number 4. Whenever '0' is encountered after reading 1, state is change to r from P and r state behaves like trap state and here process stopped and string is not accepted, since r is not final state.

Example 8.6. Design a pushdown automata for the following language, $L = \{a^n\ b^{2n} : n > 0\}$.

Solution. Language $L = \{a^n\ b^{2n} : n > 0\}$ is the set of strings, in which every string starts with a, ends with b, no 'a' comes after b, no b comes before a and number of b's are double than number of a's. For example, strings like *aabbbb, abb, aaabbbbbbb* will be accepted by PDA and strings like *a, b, ab, ba, aab, aaabbb* will be rejected.

Let PDA be $P = (Q, \Sigma, \Gamma, \delta, S, F)$

where $Q = (p, q, f, r)$

$S = \{P\}$

$F = \{f\}$

and transition relations are defined as follows :

(1) $((p, a, \in), (p, a))$

(2) $((p, a, a), (p, a))$

(3) $((p, b, a), (q, b))$

(4) $((q, b, ba), (r, \in))$

(5) $((r, b, a), (q, ba))$ → Here these two transition 4 and 5 reprsent the looping.

(6) $((r, \in, \in), (f, \in))$

Now let us see explanation for above discussed transition relation. Here first we pushed all the a's by the help of moves 1 and 2. Now as first b is encountered, then it is pushed on the stack by the help of move number 3 and when next 'b' is encountered then we pop the ba from the stack and change the state from q to r. It should be clear that moves 4 and 5 represent loop that is move 4 and 5 works on the value of 'n'. Finally when input becomes empty and stack also becomes empty then state r changes to final state that is f and string is accepted.

Example 8.7. Construct a PDA for the regular expression

$$r = 0^*\ 1^+.$$

Solution. Regular expression is $r = 0^*\ 1^+$, let us write the language for this regular expression, let us say it is L,

$$L = \{0^m\ 1^n / m \geq 0,\ n \geq 1\}$$

Clearly regular expression r contains the set of string in which every string starts with any number of 0's but certainly having atleast one 1, and it must be last symbol of the string. As 1 encountered then 0's should not come.

Let PDA be P

$P = (Q, \Sigma, \Gamma, \delta, S, F)$

where $Q = \{q_0, f\}$

$$S = \{q_0\}$$
$$F = \{f\}$$
$$\Sigma = \{0, 1\}$$

and P is defined as

(1) $((q_0, 0, \in), (q_0, \in))$
(2) $((q_0, 1, \in), (f, \in))$
(3) $((f, 1, \in), (f, \in))$

It is very important to understand that in above three moves we are not using the stack. Initially machine is at state q_0 when it reads 0's then it does nothing, but as soon as 1 is encountered then state changes to final state, with the help of move number '2'. Now after it, machine can accept any number of time 1's in the string by the help of move '3'.

Example 8.8. Construct the pushdown automata for the following language :

$$L = \{a^n\ b^{n+1} / n = 1, 2, 3, \ldots\}.$$

Solution. Language L is the set of strings in which number of b's are exactly one greater number of a's and every string starts with 'a' and ends with 'b'. No a is encountered after b.

Let PDA be P

$$P = (Q, \Sigma, \Gamma, \delta, S, F)$$

where

$$Q = \{p, q, q_1, f\}$$
$$\Sigma = \{a, b\}$$
$$S = \{p\}$$
$$F = \{f\}$$

and transitions relations are defined as follows :

(1) $((p, a, \in), (p, a))$
(2) $((p, a, a), (p, aa))$
(3) $((p, b, a), (q, \in))$
(4) $((q, b, a), (q, \in))$
(5) $((q, b, \in), (q_1, \in))$
(6) $(q_1, \in, \in), (f, \in))$.

Let us analyse the solution, first of all by the help of move number '1' and '2' we are pushing all the a's on the stack then by the help of move '3' and '4' we pop all a's from the stack when machine starts reading b's from the input tape. When machine reads last b (at this time stack is empty), the change the state to q_1, and now at state q_1, input is null string and stack is empty, then machine goes to final state, so string is accepted.

Example 8.9. Design a pushdown automata for the following language

$$L = \{a^n\ b^m / n > m \geq 0\}.$$

Solution. As usual, let us first analyse the language. Here language is the set of strings in which every string starts with 'a' (there is at least one a symbol in the strings) and if b is there then number of a's are always greater than number of b's and there should be no 'a' when any b is encountered

clearly strings *aaabb*, *a*, *aab* are in language but *ab*, *abb*, *b*, *aaabbb* ... are not accepted.

Let PDA for the language is P

where
$$P = (Q, \Sigma, \Gamma, \delta, S, F)$$
$$Q = (q_0, q_1, q_2, q_3, f)$$
$$S = \{q_0\}$$
$$F = \{f\}.$$
$$\Sigma = \{a, b\}$$

and here transition relations are defined as follows :

(1) $((q_0, a, \in), (q_1, a))$
(2) $((q_1, a, a), (q_1, a))$
(3) $((q_1, b, a), (q_2, \in))$
(4) $((q_2, b, a), (q_3, \in))$
(5) $((q_3, b, a), (q_2, \in))$
(6) $((q_2, \in, a), (q_2, \in))$
(7) $((q_2, \in, \in), (f, \in))$
(8) $((q_1, \in, a), (q_1, \in))$
(9) $((q_1, \in, \in), (f, \in))$

Now let us analyse the move of the pushdown machine. Since single '*a*' is in the language so moves '1' and '9' make the acceptance of '*a*' possible. Move 1 and 2 help us to push all the *a*'s on the stack, now as soon as '*b*' is encounttered an '*a*' is poped up and state is changed to q_2, move 4 and 5 are showing loop for poping *a*'s when *b*'s are encountered, move '6' helps us to empty stack from the '*a*' since there is no input on the input tape, finally move '7' makes the string acceptable.

8.3. Pushdown Automata and Context-Free Grammars

It should be clear now that PDA can recognise any language for which there exists a CFG. "That is class of language accepted by pushdown automata is exactly the class of context-free languages".

8.3.1. Construction of PDA Equivalent of a CFG

Let $G = (V_n, V_t, P, S)$ be a context-free grammar; we must construct a pushdown automata P such that $L(P) = L(G)$. The machine we construct has only two states, p and q, and remains permanently in state q after its first move. Also, p uses V_n, the set of non-terminals and V_t the set of terminals, as its stack alphabet. We let $P = (Q, \Sigma, \Gamma, \delta, S, F)$

where
$$Q = (p, q]$$
$$\Sigma = V_t$$
$$\Gamma = V_n \cup V_t$$

that is the set of terminals and non-terminals.

$$S = p$$

and transition relation δ is defined as follows :

(1) $((p, \in, \in), (q, S))$ (as S is starting non-terminal of CFG)

(2) $((q, \in, A), (q, x))$ for each rule $A \to x$ in CFG.

(3) $((q, a, a), (q, \in))$ for each $a \in V_t$.

The pushdown automata P begins by pushing S, the starts symbol of grammar G, on its initially empty pushdown store, and entering state q (transition 1). On each subsequent step, it either replaces the top most symbol A on the stack, provided that it is a non-terminal, by the right hand side x of some rule $A \to x$ in grammar (transition of type 2), or POPS the top most symbol from the stack, provide that it is a terminal symbol that matches the next input symbol (transition of type 3).

Example 8.10. Design a PDA for the following CFG,

$$G = (V_n, V_t, P, S) \text{ with}$$

$V_n = \{S\}$, $V_t = \{(,)\}$ and P is define δ as follows :

$$S \to \in$$

$$S \to SS$$

$$S \to (S).$$

Solution. Let us assume corresponding PDA will be

$$P = (Q, \Sigma, \Gamma, \delta, S, F)$$

where $Q = \{p, q\}$

$\Sigma = \{(,)\}$

$\Gamma = (S, (,)\}$ that is terminals and non-terminals in grammar

$S = p$

$F = \{q\}$, and δ is as follows.

(1) $((p, \in, \in), (q, S))$

(2) $((q, \in S), (q, \in))$ because $S \to \in$ is a rule of CFG

(3) $((q, \in S), (q, SS))$

(4) $((q, \in, S), (q, (S))$

(5) $((q, (, ( ), (q, \in))$

(6) $((q,),)), (q, \in))$ for each $a \in V_t$

Let us apply these transition relations on the string

$$w = ( ) ( )$$

S. No.	State	Unread input	Stack	Transition used
1	P	( ) ( )	$\in$	–
2	q	( ) ( )	S	(1)
3	q	( ) ( )	SS	(3)
4	q	( ) ( )	(S) SS	(4)
5	q	) ( )	$S)$ S	(5)
6	q	) ( )	) S	(2)
7	q	( )	S	(6)
8	q	( )	(S)	(4)
9	q	)	$S)$	(5)
10	q	)	)	(2)
11	q	$\in$	$\in$	(6)

Example 8.11. Design PDA for the grammar $G = (V_n, V_t, P, S)$ where

$$V_n = \{S\}$$
$$V_t = \{a, b, c\}$$

and P is defined as

$$S \to aSa$$
$$S \to bSb$$
$$S \to c.$$

Solution. Let PDA be

$$P = (Q, \Sigma, \Gamma, \delta, S, F)$$

where

$$Q = \{p, q\}$$
$$\Sigma = \{a, b, c\}$$
$$\Gamma = \{S, a, b, c\}$$
$$S = \{p\}$$
$$F = \{q\}$$

and δ is defined as :

(1) $((p, \in, \in), (q, S))$
(2) $((q, \in, S), (q, aSa))$
(3) $((q, \in, S), (q, bSb))$
(4) $((q, \in, S), (q, c))$
(5) $((q, a, a), (q, \in))$
(6) $((q, b, b), (q, \in))$
(7) $((q, c, c), (q, \in))$

Let us apply this on string

$$w = abbcbba$$

S.No.	State	Unread input	Stack	Transform used
1	p	$abbcbba$	$\in$	–
2	q	$abbcbba$	S	(1)
3	q	$abbcbba$	aSa	(2)
4	q	$bbcbba$	Sa	(5)
5	q	$bbcbba$	$bSba$	(3)
6	q	$bcbba$	Sba	(6)
7	q	$bcbba$	$bSbba$	(3)
8	q	$cbba$	$Sbba$	(6)
9	q	$cbba$	$cbba$	(7)
10	q	bba	bba	(6)
11	q	ba	ba	(6)
12	q	a	a	(6)
13	q	$\in$	$\in$	(5)

8.3.2. Construction of CFG Equivalent of a PDA

Let $P = (Q, \Sigma, \Gamma, \delta, q_0)$ be a PDA that accepts a language L by empty stack. Then a CFG $G = (V_n, V_t, P, S)$ that generates L can be constructed using following rules :

Assumption : Here we are taking a assumption that stack bottom is indicated by a special symbol \$.

Let S be the start symbol of grammar.

(*i*) For every $q \in Q$, add a production $S \rightarrow [q_0\ \$\ q]$ in P. Thus if there are n states in a PDA P, then there will be 'n' new productions added in P.

(*ii*) For every $q, r \in Q$, $a \in \{\Sigma \cup \in\}$, $X \in \Gamma$, if $\delta(q, a, X) = (r, \in)$, then add a production

$$[q \times r] \rightarrow a \text{ in } P.$$

(*iii*) For every $q, r \in Q$, $a \in \{\Sigma \cup \in\}$, $X \in \Gamma$, and $x \geq 1$, if $\delta(q, a, X) = (r, X_1, X_2, \ldots X_k)$ where $X_1, X_2, \ldots, X_k \in \Gamma$, then for every choice of $q_1, q_2, \ldots, q_k \in Q$, add the production

$$[q\ X\ q_k] \rightarrow a[r\ X_1\ q_1]\ [q_1\ X_2\ q_2] \ldots [q_{k-1}\ X_k\ q_k] \text{ in } P.$$

The basic idea behind this construction is to recognize that the current string in the derivation will consist of two parts, the string of input symbols read by the PDA so far and a remaining portion corresponding to the stack contents. Next we introduce variables of the proposed grammar in the form $[P\ X\ q]$ where P and q are states of the PDA. For the variable $[P\ X\ q]$ to be replaced by a symbol 'a' (a may be $\in$ also), it may be the case there is a move in PDA that reads a, pops X from the stack, and takes machine from state P to q moves. For example, it may go from state P to state P_1, that reads a and replaced X on stack by $Y_1, Y_2, Y_3, \ldots Y_k$. Then after a sequence of moves it may Pop off Y_1 and end up in state say P_2. Similarly it may pop off Y_2 and go into state P_2 and through many such steps finally entering into state P_k and popping of symbol Y_k. All these intermediate move, in their cumulative effect, are equivalent to derive a portion of strings say w and reaching a state P_k. We can represent this fact by the production

$$[P\ X\ q_k] \rightarrow a[r\ X_1\ q_1]\ [q_1\ X_2] \ldots [q_{k-1}\ X_k\ q_k]$$

Example 8.12. Consider the following PDA and construct equivalent CFG.

1. $((q_0, a, \$) \rightarrow (q_0, a\$))$
2. $((q_0, b, \$) \rightarrow (q_0, b\$))$
3. $((q_0, a, a) \rightarrow (q_0, aa))$
4. $((q_0, b, b) \rightarrow (q_0, bb))$
5. $((q_0, a, b) \rightarrow (q_0, ab))$
6. $[(q_0, b, a) \rightarrow (q_0, ba))$
7. $((q_0, c, \$) \rightarrow (q_1, \$))$

8. $((q_0, c, a) \rightarrow (q_1, a))$
9. $((q_0, c, b) \rightarrow (q_1, b))$
10. $((q_1, a, a) \rightarrow (q_1, \in))$
11. $((q_1, b, b) \rightarrow (q_1, \in))$
12. $((q_1, \in, \$) \rightarrow (q_1, \in))$

Solution. CFG using above construction production rules is as follows :

$$S \rightarrow [q_0\ \$\ q_0]/[q_0\ \$\ q_1] \qquad \text{(from rule } (i)\text{)}$$

$$\left.\begin{aligned} [q_1\ a\ q_1] &\rightarrow a \\ [q_1\ b\ q_1] &\rightarrow b \\ [q_1\ \$\ q_1] &\rightarrow \in \end{aligned}\right\} \qquad \text{from rule } (ii)$$

Now we will get remaining productions using rule (*iii*).

$$\begin{aligned}
[q_0\ \$\ q_0] &\rightarrow a[q_0\ a\ q_0]\ [q_0\ \$\ q_0] \\
[q_0\ \$\ q_1] &\rightarrow a[q_0\ a\ q_0]\ [q_0\ \$\ q_1] \\
[q_0\ \$\ q_1] &\rightarrow a[q_0\ a\ q_1]\ [q_1\ \$\ q_1] \\
[q_0\ \$\ q_1] &\rightarrow b[q_0\ b\ q_0]\ [q_0\ \$\ q_0] \\
[q_0\ \$\ q_1] &\rightarrow b[q_0\ b\ q_0]\ [q_0\ \$\ q_1] \\
[q_0\ \$\ q_1] &\rightarrow b[q_0\ b\ q_1]\ [q_1\ \$\ q_1] \\
[q_0\ a\ q_0] &\rightarrow a[q_0\ a\ q_0]\ [q_0\ a\ q_0] \\
[q_0\ a\ q_1] &\rightarrow a[q_0\ a\ q_0]\ [q_0\ a\ q_1] \\
[q_0\ a\ q_1] &\rightarrow a[q_0\ a\ q_1]\ [q_1\ a\ q_1] \\
[q_0\ a\ q_0] &\rightarrow b[q_0\ b\ q_0]\ [q_0\ a\ q_0] \\
[q_0\ a\ q_1] &\rightarrow b[q_0\ b\ q_0]\ [q_0\ a\ q_1] \\
[q_0\ a\ q_1] &\rightarrow b[q_0\ b\ q_1]\ [q_1\ a\ q_1] \\
[q_0\ b\ q_0] &\rightarrow a[q_0\ a\ q_0]\ [q_0\ b\ q_0] \\
[q_0\ b\ q_1] &\rightarrow a[q_0\ a\ q_0]\ [q_0\ b\ q_1] \\
[q_0\ b\ q_1] &\rightarrow a[q_0\ a\ q_1]\ [q_1\ b\ q_1] \\
[q_0\ b\ q_0] &\rightarrow b[q_0\ b\ q_0]\ [q_0\ b\ q_0] \\
[q_0\ b\ q_1] &\rightarrow b[q_0\ b\ q_0]\ [q_0\ b\ q_1] \\
[q_0\ b\ q_1] &\rightarrow b[q_0\ b\ q_1]\ [q_1\ b\ q_1] \\
[q_0\ \$\ q_1] &\rightarrow c[q_1\ b\ q_1] \\
[q_0\ a\ q_1] &\rightarrow c[q_1\ a\ q_1] \\
[q_0\ b\ q_1] &\rightarrow c[q_1\ b\ q_1]
\end{aligned}$$

Example 8.13. Construct PDA for the language

$$L = \{a^3\ b^n\ c^n : n \geq 0\}.$$

Solution. By the observation we can say that language contains only that set of strings, in which every string contains exactly 3*a*'s in the starting of string (like *aaa* ...). If $n > 0$, then *a*'s are followed by *b*'s and *b*'s are followed by *c*'s and number of *b*'s and *c*'s are equal.

String like *aaa, aaabc, aaabbbccc* ... are accepted and strings like *a, aa, abc, aabc, aaabbbc, aaaccc, aaacb* are not accepted.

Let PDA be $P = (Q, \Sigma, \Gamma, \delta, S, F)$

where $Q = \{q_0, q_1, q_2, q_3, q_4, q_5 f\}$

$\Gamma = \{b\}$

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

$F = \{f\}$

$S = \{q_0\}$

and δ is defined as follows :

(1) $((q_0, a, \in), (q_1, \in))$
(2) $((q_2, a, \in), (q_2, \in))$
(3) $((q_2, a, \in), (q_3, \in))$
(4) $((q_3, \in, \in), (f, \in))$
(5) $((q_3, b, \in), (q_4, b))$
(6) $(q_4, b, b), (q_4, b))$
(7) $(q_4, c, b), (q_5, \in))$
(8) $((q_5, c, b), (q_5, \in))$
(9) $((q_5, \in, \in), (f, \in))$

Here first three moves read exactly three a's and machine moves to state q_3, now if $n = 0$ then q_3 change to final state and if $n > 0$ then we push all b's on stack with the help of moves 5 and 6. Now when c encountered then we POP all corresponding b's (by the help of moves 7 and 8). If b's and c's are equal then machine finally transit to final state.

EXERCISE

1. Let input alphabet be $\Sigma = \{a, b, c\}$ and L be the language of all words in which all the a's come before the b's and there are same number of a's as b's and arbitrarily many c's that can be in front, behind or among the a's and b's. Some words in L are *abc, caabcb, ccacaabcccbccbc.*

   (*i*) Show the language is not regular.

   (*ii*) Design a PDA for above given language.

2. Let L be some regular language in which all the words happen to have an even length. Let us define the new language second (L) to be set of all the words of L second, where second we mean the first and second letters have been interchanged, the third and fourth letters have been interchanged, and so on.

   If $L = \{ba, abba, babb, ...\}$

   Second $(L) = \{ab, baab, abbb, ...\}$

   Build a PDA that accepts second (L).

3. Construct a PDA accepting the following languages :

   (*i*) $\{0^n 1^m 0^n / m, n \geq 1\}$

   (*ii*) $\{0^n 1^{2n} / n \geq 1\}$

(iii) $\{0^m 1^m 2^n / m, n \geq 1\}$

(iv) $\{0^m 1^n / m > n \geq 1\}$.

4. Let given language $L = \{a^m b^n / n < m\}$

(i) Construct a PDA for the language

(ii) Write the CFG for the language and

(iii) Convert this CFG into pushdown automata.

5. Starting with the CFG for $\{a^n b^n\}$

$$S \rightarrow aSb/ab$$

(i) put this CFG into CNF form

(ii) Give the equivalent PDA for this CNF form.

6. For each of the CFGs, construct PDA that accepts the same language they generate.

(i) $S \rightarrow aSbb/abb$

(ii) $S \rightarrow SS/a/b$

(iii) $S \rightarrow XaaX$

$X \rightarrow aX/bX/\in$

(iv) $S \rightarrow aS/aSbS/a$

(v) $S \rightarrow XY$

$X \rightarrow aX/bX/a$

$Y \rightarrow Ya/Yb/a$

(vi) $S \rightarrow Xa/Yb$

$X \rightarrow Sb/b$

$Y \rightarrow Sa/a$

7. Construct the equivalent PDA for the following CFG's

(i) $S \rightarrow Saa/aSa/aaS$

(ii) $S \rightarrow (S)\ (S)/a$

(iii) $S \rightarrow XaY/YbX$

$X \rightarrow YY/aY/b$

$Y \rightarrow b/bb$

8. Construct npda's that accepts the following languages :

(i) $L_1 = L(aaa^*b)$

(ii) $L_2 = L(aab^*aba^*)$

9. Find the npda for the language :

$$L = \{ab(ab)^n b(ba)^n : n \geq 0\}$$

10. Find the PDA for the following languages :

(i) $L = \{w : n_a(w) < n_b(w)\}$

(ii) $L = \{w : n_a(w) = 2n_b(w)\}$

(iii) $L = \{w : n_a(w) + n_b(w) = n_c(w)\}$.

11. Design a PDA which convert infix to prefix.

12. Design a PDA which convert prefix to postfix.

*****