So, in the last class we have seen how to remove the useless symbols from a context-free grammar, we have also seen how to remove the epsilon productions from a context-free grammar. But, if the language contains epsilon you must have a rule of the form $S$ goes to epsilon and you must make sure that $S$ does not appear on the right hand side of any production. The next result we will see is that we can remove the unit productions also.

What is a unit production?

(Refer Slide Time: 00:43)

A unit production is of the form $A$ goes to $B$. A non-terminal going into another non-terminal set a production is called a unit production.

So, you may want to get rid of these rules, say mentioned. If you introduce too many of this unit production in a compiler if you want to represent a programming language by a grammar and use a compiler for that, then if you have too many of this unit production. The compile time will be more so, you want to avoid these unit productions.
So, how to remove these unit productions? First of all you must see from which non terminals find all pairs A B. Such that A, from A you can go to B need not be in onestep but, in many steps. That is your starting with the grammar G is equal to N T P S and in this grammar you want to remove the unit production without affecting the language generative.

Now, you split P into two sets P 1 union P 2, where P 1 is set of unit productions and P 2 is the other set non unit. Now, you construct an equivalent grammar G dash is equal to N T P dash same non terminals same terminals. But, the production rules are different. What is P dash? P dash consists of all P 2, all P 1 will be removed but, instead you will add some more rules and what are these rules you are going to add.

Now, for every pair A B, such that from A you can go to B by unit productions, it may not be in onestep. If A goes to B is a unit production you can go in onestep or if you have A goes to c, c goes to B then you will go to B from A in twosteps. So, for every pair such that A a goes to B add rules A goes to beta if B goes to beta belongs to P 2. If you have a rule B goes to beta and P 2 then you add the rule A goes to beta to this.

So, P dash consists of P 2 plus all such rules added. Then you can show that L G is equal to L G dash, again a formal proof on the number of steps and so can be given in induction on the number of steps. But, informally you can see like this, you can see that suppose, you have a step from which you derive B. And in derivation from S you may have a sentential form say alpha Agamma, somewhere it get gamma. Then after some steps making use of unit productions only you get alpha B gammabecause, from A you are able to derive B.

And these steps are using only unit productions, then after some step you may be using some rule B goes to beta i. So, after in the next step may be alpha beta i gammathen etcetera. But, in the grammar G dash which we have constructed you have added the rule A goes to beta i, you have added the rule this. So, instead of going through all these steps start this is happening in G and in G dash what will happen is you are getting alpha Agamma. And instead of applying these unit productions and then applying B goes to beta i you straight away apply A goes to beta i. So, alphabeta i gamma you get in onestep, then the derivation in G dash can proceed.
So, the effect is the same instead of applying A derives B and then B derives beta i straight away you are using the rule A goes to beta i and deriving this in one step. So, by removing the unit productions and adding the rules of the form A goes to beta, the resultant language is not changed, the language generated is the same. So, it is advantageous to do this rather than use A unit productions but, one thing you must be careful, the number of rules here will be enormously increased depends on. Suppose, I have such pairs A B 5 6 or 7 pairs like that. The number of rules is going to increase very much but, if I have just one pair you can just remove it and so on.

So, the number of rules in P dash will be much more than the number of rules in P and when you use a parser to reduce a string you at every point you may be able, you may have to check, which rule is can be used for reduction. Then if the number of rules is too much every rule you must try.

So, that look up procedure will be more it may take more time. So, you have to see that there is a balance that in while removing the unit production you are not adding too many of other rules. But, any way this result tells you that you can rid of unit productions it is not necessary to use the unit productions. Let us take an example.

(Refer Slide Time: 08:34)

A simple example let me take (No audio from 08:25 to 28:31) S goes to a S b, S goes to A, A goes to c A d, A goes to c d. If I have this grammar, what is the language
generated? The language generated is first you will use this rule many times then from \( S \) you go to \( A \), then use this to generate equal number of \( c \)'s and \( d \)'s.

So, it will be a power \( n \), \( c \) power \( m \), \( d \) power \( m \), \( b \) power \( n \), \( m \) greater than or equal to 1 but, \( n \) can be 0 also. This will be the language generated and there is a unit production. Now, we want to get rid of this unit production. So, by this procedure you split it into \( P_1 \) and \( P_2 \), \( P_2 \) will consist of this three rules, \( P_1 \) will consist of this.

So, remove this you have to remove this but, \( P_2 \) you have to keep as it is. So, these three rules you keep, then you have a pair \( S \) \( A \). Then whenever there is a pair \( S \) \( A \), the right side hand you replace with the right hand side for every rule with \( A \) on the left hand side. So, what are the rules you will be adding now, the rules you will be adding will be \( S \) goes to \( A \) \( d \), \( S \) goes to \( d \). So, instead of applying \( S \) goes to \( A \), then \( A \) goes to \( c \) \( A \) \( d \) you can straight away apply this rule, instead of applying \( S \) goes to \( A \), \( A \) goes to \( c \) \( d \) you can straight away use this rule.

Now, look at the grammar without this rule, but, adding these two rules you will see that any number of equal number \( a \)'s and \( b \)'s can be generated using this rule. Then you can go from \( S \) to this or \( S \) to this, you see that at least 1 \( c \) and 1 \( d \) must be generated, use this many times then use this 1 \( c \) \( d \) will be generated, use this many times as many times you want. And then use this once and then this you will get \( c \) square \( d \) square. If you want to get more use this many times for a power \( n \) \( b \) power \( n \).

Then \( c \) cube \( d \) cube if you want use this once, use this once, use this once then any number of \( c \) power \( m \), \( b \) power \( m \) you can generate \( d \) power \( m \) you can generate with proper combination of these three rules. Just 1 \( c \) \( d \) alone you want means you use this rule after applying this rule you need not to apply this rule at all because, \( n \) can be 0 also just \( c \) \( d \) also belongs to the language. You can start with \( S \) and derive \( c \) \( d \) there \( n \) can take the value 0 1 2 etcetera \( m \) takes a value 1 2 3 that should be at least 1 \( c \) and 1 \( d \). So, this is the way you get rid of the unit productions.
So, we have seen how to remove the useless symbols that is form every symbol should occur in a derivation like, every symbol X must occur in a derivation like this, from S you should be able to reach x and then from X you should be able to derive a terminal string. So, for that we have seen that how to apply two Lemmas Lemma 1 and Lemma 2 and you have to apply them in that particular order. The first make sure that from every non-terminal a terminal string is derived with this again one make sure that every symbol is reachable from S.
So, now, you have this result every CF L without epsilon can be generated by a CF G without epsilon productions, unit productions and useless symbols. So, in which order you go about because, we are without loss of generality, we are assuming that the language S 0 contain epsilon. So, the first step will be remove the epsilon productions because, when you remove the epsilon productions you may introduce unit productions.

So, first remove the epsilon productions after removing them remove the unit productions, then after doing this remove the useless symbols use Lemma 1 and Lemma 2 in that order. So, ultimately you will end up with a grammar which does not have epsilon productions, unit productions and useless symbols. Such a grammar is called a reduced grammar. (No audio from 14:30 to 14:39) So, we are interested in finding out the reduce to grammar for a language. Now, one assumption we have made is that it has it does not have epsilon.

(Refer Slide Time: 15:07)

So, we have assumed epsilon does not belong to the language with which we started. Suppose, what do we do suppose epsilon belongs to L, then how do we go about it. In this case L minus epsilon can be generated by a reduced grammar. (No audio from 15:42 to 15:49) L minus epsilon will be reduced by a grammar.

Now, we have to include epsilon for that what do we do? So, this is suppose, this is generate by G is equal to N T P S then you have G dash where you add on more symbol S 1 same terminals P union say P 1 I will put comma S the new symbol will be the start symbol you
addonemore symbol $S_1$ till set of non terminals and make it the start symbol. And all these productions will be there but, you may add a few more productions and what is the set of productions $P_1$ you are going to add. First of all you want to include epsilon in the language. So, you have a rule $S_1$ goes to epsilon. So, if you want to derive epsilon you will use this rule alone $S_1$ goes to epsilon and then derive epsilon no other steps involved. Other rules will not have epsilon on the right hand side. But, starting from $S_1$ you also have if $S$ goes to alpha belongs to $P$ add $S_1$ goes to alpha $P_1$. If $S$ goes to alpha belongs to $P$ the earlier start symbol going into alpha, then you also add $S_1$ goes to alpha.

So, $P_1$ will consist of rules of the form $S_1$ goes to epsilon and $S_1$ goes to alpha, where $S$ goes to alpha belongs to $P$. This makes sure that the start symbol does not occur on the right hand side. Now, alpha cannot contain alpha belongs to $P$ the earlier grammar the earlier grammar did not have $S_1$. So, alpha cannot contain $S_1$. So, $S_1$ will not occur on the right hand side of any production and if you want to derive epsilon you was you have to use this rule and this rule only and you will able to drive epsilon.

(Refer Slide Time: 18:54)

Now, if you have derivation in $G$ of the form $S$ goes to some alpha then something etcetera. In $G$ dash you will have $S$ dash goes to alpha I am sorry $S_1$ goes to alpha same derivation you can have the first step alone will be changed. So, whenever a
string is derivable here it is derivable here and whenever a string is derivable here, it is derivable here, apart from that you are also deriving epsilon.

So, L minus epsilon is generate by a reduced grammar L is generated by G dash and in G dash you have the property that, the start symbol does not occur on the right side of any production. (No audio from 20:06 to 20:14) The reason for this is if you have a derivation S 1 goes to this some alpha, alpha 1 or alpha 2 whatever it is. If we have a derivation the successive sentential forms will be non decreasing in length.

Length of alpha 1 will be more or equal to alpha length of alpha 2 will be more or equal to alpha and so on. It will not decrease but, if you have S 1 on the right hand side the start symbol. Because, of this rule any time you will be able to apply this rule, the success sentential form can reduce in length you want to avoid that, in anyway derivation you want to make sure that the successive sentential forms are non decreasing in length. And that is why you want to avoid having S 1 on the right hand side of any production since you are adding this rule S 1 goes to epsilon that is a reason.

So, whenever we have this problem with epsilon, epsilon has slight whenever you want to have the empty word there will, in the proofs and all there will be a slight problem. So, you want to avoid writing a very lengthy proof. So, you make sure that the grammar generator has this format. So, the next thing is we shall study two normal forms.

(Refer Slide Time: 21:54)
What are the normal forms? One is called this Chomsky normal form. It says that every C F L without epsilon can be generated by a C F G with rules of the form A goes to B C or A goes to a, where A B C are non terminals and a is a terminal. So, without loss of generality, we can assume that the language, S not contain epsilon and we know how to accommodate for epsilon now. So, if a C F L without epsilon, then it can be generated by the grammar with rules or of the form A goes to B C and A goes to a. That is on the left hand side as usual for any C F G you how only a single non terminal on the right hand side you have two non terminals or a single terminal on the right hand side. You can just have two non terminals or a single terminal.

So, every C F G you can bring in to this form this is called Chomsky normal form. How can you bring it? Let us, take an example by considering an example it is much more easy.

(Refer Slide Time: 24:22)

Let us, take the grammar. This you know that the language generated is a power n b power n greater than or equal to 1. Now, we want to bring this to Chomsky normal form. How do we go about doing this, first step will be for every terminal symbol introduce a new non terminal. So, for small a introduce A non terminal it should not. For small a you use A and small b use capital B. Now, the rules will become in
this example the rules will become $S$ goes to $A$ $S$ $B$, $S$ goes to $AB$, $A$ goes to $a$, $B$ goes to $b$ (No audio from 26:11 to 26:22).

Now, we can see that, this rule is in Chomsky normal form, this rule is also in Chomsky normal form, this rule is also in Chomsky normal form. But, this is not in a Chomsky normal form. So, you have to do something about it.

(Refer Slide Time: 26:41)

So, the second step is if $A$ goes to $B_1 B_2 B_m$ is a rule note that now you have made all the symbols on the right hand side as non terminals.

So, $B_1 B_2 B_m$ are all non terminals now. So, this rule you can split like this $A$ 1 goes to $B_1 D_1$ $D_1$ goes to $B_2 D_2$ like that upto $D_{n-2}$ goes to $B_{n-1} B_n$. The previous one will be $D_{n-3}$ goes to $B_{n-2} D_{n-2}$ $D_{n-2}$ goes to $B_{n-1} B_n$.

So, at the end of the first step after we have replaced every terminal with a non terminal, the rules will be either in this form Chomsky normal form, terminal rules or rules will be of the form on the left hand side you have a non terminal.

And the right hand side you have a string of non terminals. So, when you have this when you have a string of non terminals on the right hand side you would not have only two of them. So, you have to split this rule by introducing new non terminals $D_1 D_2 D_3$ etcetera up to $D_{n-2}$ and split this rule. But, you must be careful that if you have two such rules when you want to split this rules you are introducing the non
terminals D 1 D2 etcetera. Another rule is there then you must introduce some other non terminal E 1 E 2 etcetera, you should not use the same non terminals that because they will mix up and then create problem.

So, in this example which we have considered these two are in Chomsky normal form, this is no problem it is in Chomsky normal form. The first rule alone is in not in Chomsky normal form. So, you introduce a new non terminal say D, S goes to D and then D goes to S B you introduce a new non terminal and have S goes to AD and D goes to S B.

So, instead of applying like this the same result will be achieved in two steps by applying AD first and then D goes to S B when you apply it will be A S B. So, without any problem, any grammar you can convert into Chomsky normal form. Let us take one more example.

(Refer Slide Time: 30:14)

S goes to a S a, S goes to b S b, S goes to c. What is the language generated? The language generated consists of strings of the form w c w R. Where w belongs to a’s and b’s. Now, convert this into Chomsky normal form, convert this grammar into shortened form is CNF, Chomsky normal form. The first step is introduce a new non terminal for every terminal symbol, if you do that you will get S goes to A S A, S goes to B S B, S goes to c, A goes to a B goes to b.
These are in Chomsky normal form no problem. So, the other two rules you have to convert into Chomsky normal form. So, when you want to convert this, what do you do? You introduce a new non terminal. And instead of this you will have S goes to say AD introduce a new non terminal D and have it as S goes to AD, D goes to SA.

Now, when you want to convert this rule, introduce another non terminal E not the same. You should not use the same D. S goes to B E, E goes to S B. So, the effect of applying this rule is achieved in two steps by applying this first and then this. The effect of applying this rule is achieved by applying this and this. Now, all these four rules are in Chomsky normal form. So, every C F G can be converted into Chomsky normal form. (No audio from 32:19 to 32:31)

(Refer Slide Time: 32:39)

There is another normal form called Greibach normal form. There are several normal forms for C F G but, these are the two which are mainly used. The Chomsky normal form is very useful in proving some results about context free grammars. Greibach normal form is useful for proving the equivalence within push down automata and context free grammars and also sometimes in parsing purposes if the grammar is in Greibach normal form it is easy to parse.

What is Greibach normal form? Every CFL without ε can be generated by a CFG with rules of the form:

A → a B₁... Bₘ  A, B₁,..., Bₘ ∈ N
A → a

Where A B₁ etcetera are non terminals and a is a terminal. That is on the left hand side you have a single non
terminal on the right hand side you have a single terminal or a single terminal followed by a string of non terminals you may have one non terminal, two non terminal, three non terminals any number you can have.

So, the rules are of the form such that set on the left hand side you have a single non terminal, on the right hand side you can have a single terminal or a single terminal followed by a string of non terminals. So, you can bring any C F G to this form. Let us see how we can do that.

(Refer Slide Time: 35:12)

Again take the same simple examples if S goes to a Sb, S goes to ab this we have already considered. The first step is for every terminal symbol; introduce a new non terminal symbol. So, in let us, let me illustrate after every step, how for this example. There are five steps in the conversion but, this example may not use all the five steps. So, let me see how we can do this. So, S goes to A S B, S goes to A B. Then you have A goes to a this is step is a same as the previous conversion to Chomsky normal form. Now, you find that because of this the language generator is not affected, the language generated is still going to be the same first you will generate capital a power n, b power n convert all this capital A’s to small a’s and capital B’s to small b’s. Now, these two are already in Greibach normal form.
The second step here will be (No audio from 37:07 to 37:18) introduce an order among non terminals by renaming them. (No audio from 37:41 to 37:50) So, the grammar which we consider earlier (No audio from 37:55 to 38:01) you can call them as A 1 A 2 A 3. So, make S as A 1, A as A 2, B as A 3 there are non terminals here.

So, make S (No audio from 38:20 to 38:28) S as A 1, A as A 2, B as A 3 you are renaming them. So, now the rules will become A 1 goes to A 2 A 1 A 3, A 1 goes to A 2 A 3, A 2 goes A, A 3 goes to b. So, you are having an order among the non terminals. This is also very simple it is just the language generator will not be affected because just you are renaming the non terminals and calling them as A 1 A 2 A 3.
The third step is involves the use of two Lemmas, we will see what are the two Lemmas.

(No audio from 39:25 to 39:31) Lemma 1 is to define an A production to be a production with variable A on the left, an A production is a production with variable A on the left. Let G be a CFG. Let \( A \rightarrow \alpha_1 B \alpha_2 \) be a production in P and \( B \rightarrow \beta_1 \beta_2 \beta_r \) be the set of all productions. Let \( G_1 = (N, T, P, S) \) be obtained from G by deleting the production \( A \rightarrow \alpha_1 B \alpha_2 \) for some reason you want to get rid of it. We will see why we need this Lemma later.

Now, when you want to get rid of this rule, what should you do? A production in P and with B on the left here, B is a non-terminal with B on the left hand side you have a set of production B goes to \( \beta_1 \), B goes to \( \beta_2 \), B goes to \( \beta_r \). A set of productions you have with B on the left hand side, with B goes to \( \beta_1 \) \( \beta_2 \) \( \beta_r \) be the set of B productions. Then from G you construct G' such that G' has the same non-terminals same terminals but, the productions have same.

Let G' be obtained from G by deleting the production alpha 1 B alpha 2 for some reason you want to get rid of it, we will see why we need this Lemma later. So, you delete this rule but, when you delete this rule.
The effect must be obtained by some other rules. So, when you delete this rule what you do is you add the rules \( A \) goes to \( \alpha_1 \beta_1 \alpha_2 \alpha_1 \beta_2 \alpha_2 \alpha_1 \beta_r \). So, \( r \) rules you add, then the language generated does not change. So, in a sense what happens is.

(No audio from 41:31 to 41:58)

In this grammar I have a derivation.

(Refer Slide Time: 42:06)
Somewhere I have alpha A beta or probably alpha A gamma you can write. And the next step I use the rule alpha 1 I will put A goes to B beta I use this rule.

So, A goes to alpha B beta I apply and the next gamma will be there. And the next step the beta will be replaced by some beta i, alpha 1 alphabeta ibeta gamma, then the derivation proceeds. Now, this rule you want to remove, now when you remove this you are adding the rules of the form A goes to alpha beta 1 beta, A goes to alpha beta 2 beta and so on, A goes to alpha betar betas such rules you are adding.

So, at this stage instead of using this rule and then replacing B by beta i straight away you can use the rule A goes to alphabeta i beta and write it as alpha 1 alpha beta i betagamma. In the original grammar G you may have a derivation which a is replaced by alpha B beta first and then B replaced by beta i. By in the new grammar G dash you do not have this production A goes to alpha B beta.

So, but, instead of you have a added these productions, so, instead of getting it in two steps like this straight away you apply the rule A goes to alpha beta i beta and get this. Send the derivation will proceed as before. So, the effect is the same so, the language generated will not be effective this is when you want to get rid of a particular rule for some reason. Then there is another Lemma this also, we will be making use in steps three and four of that conversion to Greibach normal form.

(Refer Slide Time: 45:13)
So, let us see what is, let $d$ is equal to $n \ T \ P \ S$ be a C F G and then this is a context free grammar. Then let $A$ goes to $A \alpha_1, A \alpha_2, A \alpha_r$ be the set of a productions for which $A$ is the left most symbol of the right hand side. Such rules are called left recursive rules. A rule of the form $A$ goes to $A \alpha \beta$ is called a recursive rule. A rule of the form $A$ goes to $A \alpha$ where $A$ is the first symbol left most symbol this is called a left recursive rule. A rule of the form $A$ goes to $\beta_1, A$ goes $\beta_2, A$ goes to $\beta_S$ these are remaining $A$ productions they are not left recursive rules.

(Refer Slide Time: 45:48)

Sum $A$ going into $A \alpha A \beta$ is called arecursive rule. A rule of the form $A$ goes to $A \alpha$ where $A$ is the first symbol left most symbol this is called a left recursive rule. A rule of the form $A$ goes to $\alpha \beta$ where $A$ is the right most symbol that is called a right recursive rule. (No audio from 46:23 to 46:32) In many cases you may want to avoid left recursion, when we learn about parsing, we will see that we have sometimes we will want to avoid left recursion. But, when you want to avoid left recursion you will be introducing right recursion but, that is ok.

So, that is what we are going to do here, we want to avoid the left recursive rules, we do not want have left recursion. Then let us see what the set $A$ goes to $A \alpha_1 A \alpha_2 A \alpha_r$ be the set of $A$ productions for which $A$ is the left most symbol of the right hand side. Then there are other rules $A$ goes to $\beta_1 A$ goes $\beta_2 A$ goes to $\beta_S$ these are remaining $A$ productions they are not left recursive rules.

Then you introduce you want to replace all the $A$ rules by a set like this you have a new grammar, where you introduce a new non-terminal $Z$ and instead of having the set of $A$ productions. So, the new grammar let $G_1$ be equal to $N$ union $Z, T \ P \ S$ be the C F G
form by adding the variable $Z$ to $N$ and replacing all the $A$ productions by the productions $A$ goes to $\beta_1$, $A$ goes to $\beta_i Z$, where $i$ varies from $1$ to $S$ and $Z$ goes to $\alpha_i Z$, where $i$ varies from $1$ to $r$.

So, the earlier with $A$ you have $r$ left-recursive rules and $S$ other rules, which are not left recursive. But, even you want to replace you are introducing a new non terminal $Z$ and you are having $2S$ rules like this and $2r$ rules like this. So, actually what you are trying to do is the $r$ plus $S$ rules you are replacing with $2r$ plus $2S$ rules. Now, how do we justify this?

(Refer Slide Time: 49:16)

So, we you are having $A$ goes to $A\alpha_1 A\alpha_2 A\alpha_r$, $r$ rules with which are left recursive. And $A$ goes to $\beta_1 \beta_2 \beta_S$, $s$ other rules. So, $r$ plus $S$ rules you are having. Now suppose, you are applying starting from sum $A$ then you have $A\alpha_1$ and from this you are deriving something. AI would put $A\alpha_i 1$. So, 1 of the rules I am applying here then from this again another left recursive rule I am applying $A\alpha_i 2$ and from this you may derive something. Then again from this you have $A\alpha_i 3$ and from this you may drive something. You can proceed like this until you have $A\alpha_i n$. And then here you use the rule an non recursive rule $\beta_j$ from this again you can derive something.

So, from this $A$ the string derived is $\beta_j \alpha_i n, \alpha_i n$ minus 1, $\alpha_i 1$ the string derived is this. Now, the same string you must be able to derive with the replaced rules.
What are the rules which you have replaced $A$ goes to $\beta i \beta j$ $i$, can put $A$ goes to $Z$, where $i$ varies from 1 to $S$ then $Z$ goes to $\alpha i$. $Z$ goes to $\alpha i Z$, where $i$ varies from 1 to $r$.

Now, how do I get the same effect with these rules start from $a$ then use the rule $\beta j Z$. Then you use the rule $\alpha i n Z$ you have such a rule. From this again you can drive something then from this $\alpha i n$ minus 1 $Z$, then again use the rule $\alpha i n$ minus 2 $Z$ and so on. Until you have $Z$ goes to $\alpha i 2 Z$, then $Z$ will go to $\alpha i 1$.

So, the string generated here will be $\beta j \alpha i n \alpha i n$ minus 1 etcetera up to $\alpha i 1$ and from this $\alpha$’s again something else can be derived. So, the string generated is $\beta j \alpha i n \alpha i n$ minus 1 $\alpha i 1$. Here you have the same effect by using this rule first $\alpha i n$ goes to $\beta j$ is that so $\beta j$ is generated, then you can use a rule of the form $Z$ goes to $\alpha i n Z$ then $Z$ goes to $\alpha i$ n minus 1 is that and so on.

So, you will get the same string $\beta j \alpha i n$ etcetera. So, the effect of using such rules you can obtain from this also, the language generated does not change it remains the same. But, instead of $r$ plus $S$ rules now, we have 2 $r$ plus 2 $S$ rules and what have we achieved by doing this, we have avoided left recursion left recursion is avoided.

But, what have we done for that, we have introduced right recursion $Z$ occurring as a last symbol introduces right recursion. So, these two Lemmas, we have to use again and again in
the step 3 and 4 of the conversion to Greibach normal form there are 5 steps there. We shall consider the examples and the procedure continue with that in the next class. These are not the only two normal forms; there are other normal forms as well. But, these are the main normal forms which are used to especially Greibach normal form is essentially improving the equivalence between push down automata and context free grammars.