Theory of Computation

Prof. Kamala Krithivasan

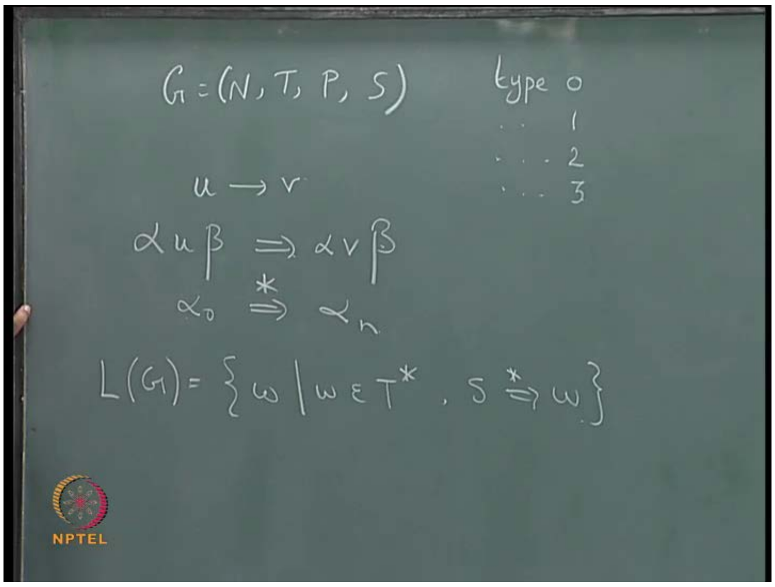
Department of Computer Science and Engineering

Indian Institute of Technology, Madras

Lecture No. # 02

Grammars and Languages Generated

(Refer Slide Time: 00:16)



$G = (N, T, P, S)$

type 0

. . . 1

. . . 2

. . . 3

$u \to v$

$\alpha u \beta \Rightarrow \alpha v \beta$

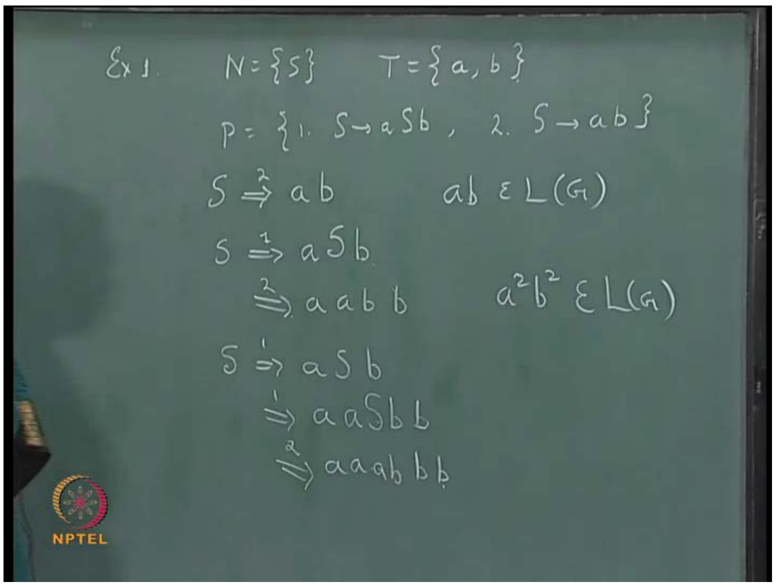
$\alpha_0 \overset{*}{\Rightarrow} \alpha_n$

$L(G) = \{ w \mid w \in T^*, S \overset{*}{\Rightarrow} w \}$

We were considering the definition of a grammar. The grammar consists of four components G is equal to N T P S, where N is a set of non terminals, T is the set of terminals, P is a set of production rules and S is the start symbol. We consider four types of grammar type 0, type 1, type 2 and type 3. If the rule is of the form u goes to v, then the application of a role means, if you have a sentential form alpha u beta then you replace u with v then you get alpha v beta. This is means from alpha u beta you directly derive alpha v beta and then if you derive something say from alpha naught in alpha n from alpha naught in n steps, you denote it by alpha naught derives alpha n and the language generated by the grammar is denoted by L G and it is given by w, w belongs to

T star, it is a set of terminal string derivable from the start symbol. This is a way the language generated by the grammar is defined.

(Refer Slide Time: 01:45)



Ex 1. $N = \{S\}$ $T = \{a, b\}$

$P = \{1.\ S \to aSb,\ 2.\ S \to ab\}$

$S \overset{2}{\Rightarrow} ab$ $ab \in L(G)$

$S \overset{1}{\Rightarrow} aSb$

$\overset{2}{\Rightarrow} aabb$ $a^2b^2 \in L(G)$

$S \overset{1}{\Rightarrow} aSb$

$\overset{1}{\Rightarrow} aaSbb$

$\overset{2}{\Rightarrow} aaabbb$

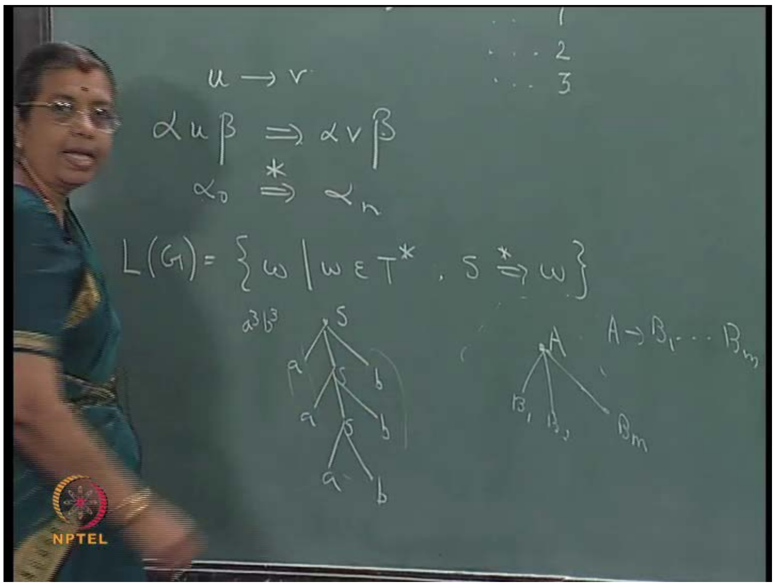
Let us consider some examples, consider example one, the non terminal has the grammar a non terminal S only one non terminal, the terminal symbols are a and b and you have the set of productions, I will number them because the first rule is S goes to a S b, the second rule is S goes to a b and S of course, is the start symbol, there is only one non terminal in this example and that happens to be the start symbol.

Now, what is the language generated by this grammar? Let us derive some strings and see. The any terminal string derivable from the start symbol belongs to the language generated by the grammar. So, let us see what sort of terminal strings are derivable in this grammar. So, look at this, using the second rule starting from the start symbol using rule two alone you will get a b. So, a b is a string which belongs to the language generated. So, I can say a b belongs to L of b. In the other hand you can say S you can apply rule one and then get a S b and then from this S you can derive using rule two, S will be replaced by a b, this b will occur here, this a will occur here. So, a a, b b, is a terminal string which is generated by the grammar. So, you have a squared b squared belonging to L G, a a, b b, can represent as a squared b squared.

And then look at this use rule one first you get a S b, then again use rule again rule one again. So, you will get a, this S is replaced by a S b, then this b will come here. So, you have a a S b b b and then use rule two you will get a, a, a, b, S is replaced by a b then you get b b this is again another terminal string derivable and it belongs to the language generated. So, you have a cubed b cubed belonging to L G.

Now, you see the pattern whenever you apply rule one. One a is generated, one b is generated, in between you have the S and finally, when you replace that S by a b another a b is generated. So, at any step you have equal number of a's and equal number of b's and not only that first a's occur first and then b's occur next. So, the language generated consists of strings of the form L of G is a power n, b power n, n greater than or equal to one, strings of the form a power n, b power n, number of a's followed by an equal number of b's will be generated by this grammar. And to generate a power n, b power n you will be applying rule one n minus one times and rule two once and in general the derivation tree take a cubed b cubed.

(Refer Slide Time: 05:59)

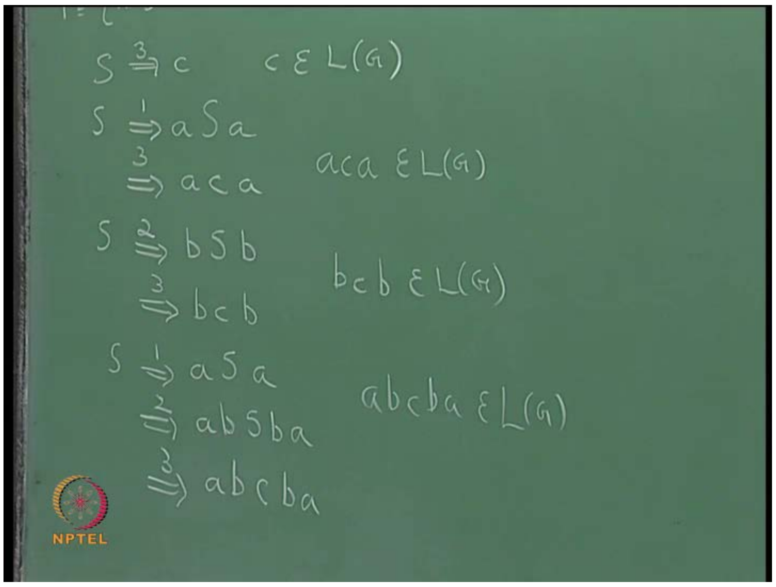


$u \to v$
$\alpha u \beta \Rightarrow \alpha v \beta$
$\alpha_0 \overset{*}{\Rightarrow} \alpha_n$
$L(G) = \{ w \mid w \in T^*, S \overset{*}{\Rightarrow} w \}$
a^3b^3
S a S b a S b a b
A $A \to B_1 \cdots B_m$
B_1 B_2 B_m
1
2
3
NPTEL

The derivation tree will be like this S a, S b, a S b, b, the derivation tree, in a derivation tree in a context free grammar are type two, this is a type two grammar, because see that on the left hand side you have single non terminal and on the right hand side you have a string.

So, this is type two or context free grammar. In general the rules when you apply an internal node, suppose you have a rule A goes to B 1, B 2, B m the internal node labeled A there will be a tree and somewhere the internal node will be labeled A and it will have sons B 1, B 2, B m, the derivation tree it will be represented like this. The leaves will have terminal symbols and if you read out the leaves from left to right you get the string generator. From left to right if you read out the labels of the leaves a a a, b b b that is the string generator, that is called the result of the tree, it is called the result of the tree. Other terminal string generated by the tree.

(Refer Slide Time: 07:37)



$S \overset{3}{\Rightarrow} c \quad c \in L(G)$

$S \overset{1}{\Rightarrow} aSa$

$\overset{3}{\Rightarrow} aca \quad aca \in L(G)$

$S \overset{2}{\Rightarrow} bSb$

$\overset{3}{\Rightarrow} bcb \quad bcb \in L(G)$

$S \overset{1}{\Rightarrow} aSa$

$\overset{2}{\Rightarrow} abSba \quad abcba \in L(G)$

$\overset{3}{\Rightarrow} abcba$

So, this is one example. Consider, some more examples, consider another example. Here again there is only one non terminal S which happens to be the start symbol, there are three terminal symbols a, b, c and the production rules are S goes to a S a, second rule is S goes to b S b, third rule is s goes to c, there are three rules, now. S goes to a S a, S goes to b S b and S goes to c.

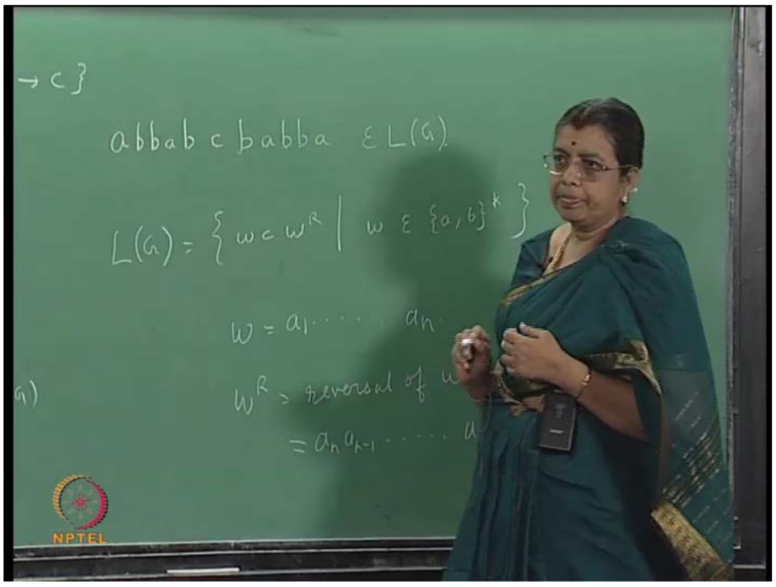
Now, what is the language generated by this grammar? Let us see how strings are derived in this grammar? Starting from S you apply a rule three you get c right. So, c belongs to

L of G and then starting from S apply rule one you will get a S a, then you apply rule three you get a c a. So, a c a will belong to L of G.

Now, starting from S if you apply rule two you will get b S b and then, if you apply rule three you will get b c b. So, b c b will belong to the language. Now, starting from S you apply rule one then, apply rule two then, apply rule three you get a b c b a. So, a b c b a will belong to the language.

Now, what sort of strings can be derived in this grammar? So, first you can apply rule one or rule two. When you generate using rule one, one a will be generated this side, another a will be generated this side, then you can use rule one again or rule two, one will, one b will be generated here, one b will be generated this side or one a will be generated and one a will be generated. Finally, you have to end up the derivation with S goes to c. Once you apply that S goes to c you cannot proceed further you have to stop the derivation at that stage, but before applying rule three, rule one and two you can apply and apply again and again in any order. Either apply one or two and two and one or one one two any order you can apply, but whenever you apply rule one, one a will be generated this side one a will be generated this side. When you apply rule two one b will be generated this side one b will be generated that side.

(Refer Slide Time: 11:26)

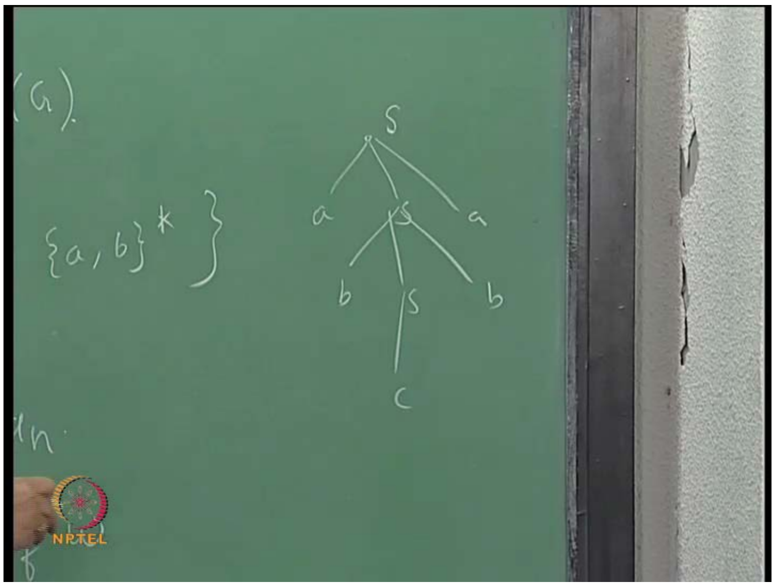


So, you can see that any string generated for example, a string of the form a b b, a b some string some sequence of a's and b's followed by c, then this b is generated this side also you must get a b.

So, this side what will be the string generated? b a, b, b, a, the reverse of this reversal of this will be generated this side. So, such a string will belong to the language generated. So, L G consists of strings of the form w c w r where w belongs to a, b star. w is a string of a's and b's, a string of a's and b's then c then the reversal of the first portion will be generated.

Now, by w if w is a string of the form a1, a 2, a n, w r is the reversal of the W, reversal of W and that is nothing, but a n, a n minus 1, a 2 a 1 this is called the reversal of this string.

(Refer Slide Time: 13:21)

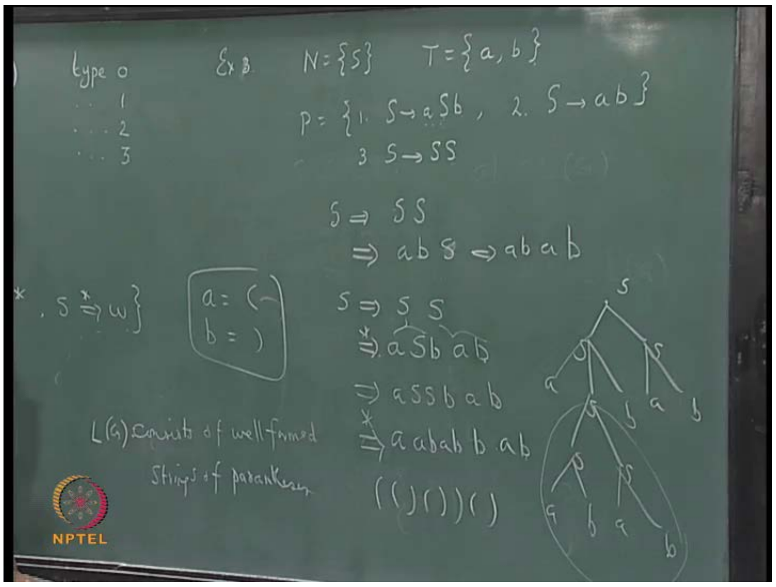


So, the language generated by this grammar has the strings will be of the form w c w r and if you take a derivation tree s a s a, b s b c, something like that or a b c b a. So, you can see that when S is replaced, one a is here one a is here one again then this axis expanded b S b one b is here one b is here then S is replaced by c, the result is you have to read the leaves from left to right a b c b a .

So, this is another example. Both the examples, which we considered here are context free or type two and one more thing on the right hand side you see that there is only one non terminal. So, actually it is a slightly restricted version of a context free grammar. In the definition of context free grammar or type two grammar on the right hand side you can have any string, but in these two examples which we have considered. On the right hand side you are having along with terminal symbols only one non terminal. So, it is slightly restricted version of a context free grammar such a grammar is called a linear grammar, linear grammar and the language generated is a linear language.

The condition for a linear grammar is r h s has at most one non terminal, you can have just terminals alone. So, after considering these two, we will consider one more example of a context free grammar, slightly more general than this, then some type three grammars we will consider. These do not belong to type three, the reason is type three the right hand side should be a terminal symbol followed by a non terminal. Here you are having a terminal symbol followed by a non terminal, then another terminal symbol. So, these two will not belong to type three class, they are slightly higher type two.

(Refer Slide Time: 16:30)


type 0
1
2
3
Ex 3. $N = \{S\}$ $T = \{a, b\}$
$P = \{1.\ S \to aSb,\ 2.\ S \to ab\}$
$3\ S \to SS$
$S \Rightarrow SS$
$\Rightarrow abS \Rightarrow abab$
$S \overset{*}{\Rightarrow} w\}$
$a = ($
$b =)$
$S \Rightarrow SS$
$\overset{*}{\Rightarrow} aSbab$
$\Rightarrow aSSbab$
$\overset{*}{\Rightarrow} aababbab$
L(G) consists of well formed strings of parantheses
(()())()
S
S
S
a
b
a
b
S
S
a
b
a
b

Let us consider one more one which is a variation of this, but you see that lot of different strings can be generated. There is only one non terminal S example three there is only one non terminal S, two terminal symbols a and b, there are two rules S goes to a S b, S goes to b then there is one more rule S goes to S S, one more rule is there this S goes to S S. With this additional rule what is the language generated, what types of strings are generated by the grammar? Start from with these two rules alone you see that you can generate strings of the form a power n, b power n. You are having one additional rule S goes to S S.

Now, this rule sees that there are two non terminals on the right hand side. So, this is not a linear grammar, it is higher than that you say type two grammar because left hand side there is only one non terminal right hand side you have a string. So, it is type two, but it is not linear. Now, let us derive some strings and see S goes to S S because all a power n, b power n can be derived by the first two rules alone, S goes to S S .Then from the first S you derive a b from the second S also you derive a b. So, a b a b can be derived which cannot be derived with the first two rules alone. Similarly you can see that a b, a b, a b can be derived a b, a b, a b, a b four a b's can be derived and so on. More than that you can derive something like this S goes to S S, S goes to a S b a b this S is replaced by a S b, this is replaced by a b.

Now, this S is replaced by S S. Now, each one of this S S you can replace by a b. So, a a b, a b b, a such a string can be generated by the grammar. How do you describe the language generated by this? That let us consider the derivation three for this it will be S goes to S S, this goes to a b, then this goes to a S b, this goes to S S, then a b a b, the derivation tree for this derivation is that.

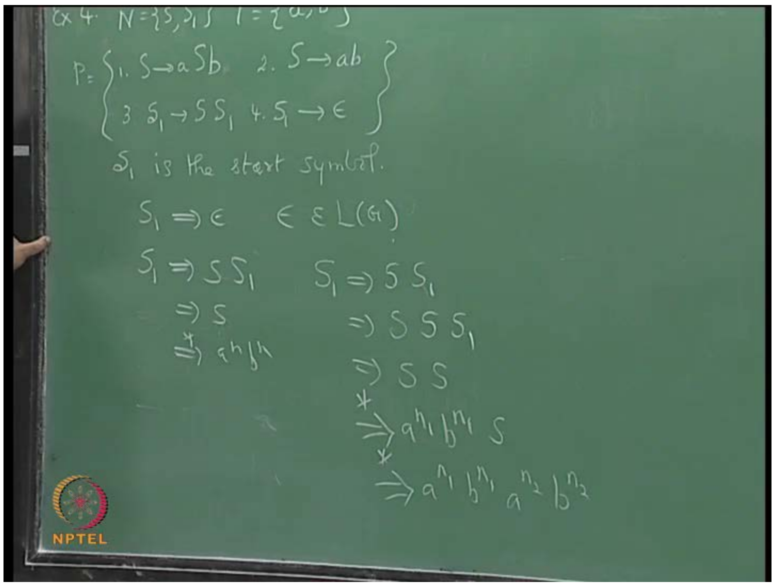
So, please note that from going to from this sentential form to this sentential form two steps are involved that is why I have put star here. From this to this only one step is involved S b written as S S going from this to this two steps are involved S is replaced by a b, S is replaced by a b. The derivation is sequential here at any step you are using only one rule the order is immaterial, it has to follow up our product, but which one you replace first is immaterial, but if you use a leftmost derivation you must replace the leftmost non terminal.

So, the derivation tree looks like that. Now, this is the father of these and these three are the sons this is an ancestor of this, this is a descendant, all those terminologies and this if you look at this. This is a sub tree. So, these definitions sub tree, father, sons, ancestor, descendant, whichever terminology you use for terminology trees they are also used here and the internal nodes have non terminals as their labels, leaves have terminals and if you read out the labels of the leaves from left to right you get the string generated and that is called the result of the tree.

Now, how can you describe precisely what is L G here, what is L G here? You must realize that when you use rule three you are generating only S S. Then, when you use rule one or rule two you generate one a and one b, whether you use rule one or two one a will be generated, one will be b will be generated. So, the resultant string will always have equal number of a's and equal number of b's something more also. You cannot generate b a can you generate b a with this, you cannot generate b a with this. So, the resultant string generated in the grammar will have equal number of a's and b's. For example, here you have four a's and four b's not only that there is some more feature. What is that feature? If you look at a as a left parenthesis and we look at b as the right parenthesis, what does that mean, it is a well balanced string of parenthesis. If you look at a as a left parenthesis and if you look at b as the right parenthesis, this string will be right.

So, any well formed string of parenthesis will be generated by this grammar. So, the language generated is L G consists of well formed strings of parenthesis. If you look at a as the left parenthesis and b as the right parenthesis. This particular grammar and this particular language has certain special features. We will learn about that as we move there is a difference between this and that, such linear grammar. Not only that, I will again give one more example, there is a basic difference between this and other type. So, this is called in Dyck set. This particular language well formed strings of parenthesis is called the Dyck set.

(Refer Slide Time: 25:24)



Ex 4. $N = \{S, S_1\}$ $T = \{a, b\}$

$$P = \left\{ \begin{array}{ll} 1.\ S \to aSb & 2.\ S \to ab \\ 3\ S_1 \to S S_1 & 4.\ S_1 \to \epsilon \end{array} \right\}$$

S_1 is the start symbol.

$S_1 \Rightarrow \epsilon$ $\quad \epsilon \in L(G)$

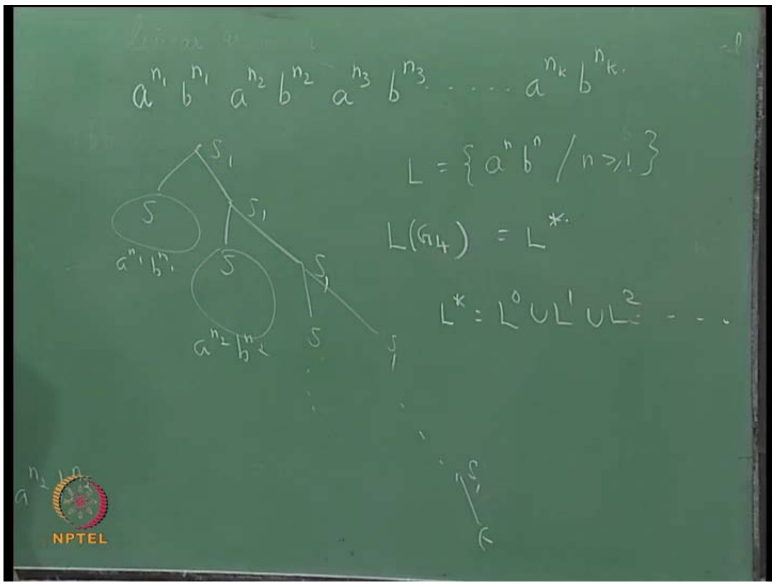
$$\begin{array}{ll} S_1 \Rightarrow S S_1 & S_1 \Rightarrow S S_1 \\ \Rightarrow S & \Rightarrow S S S_1 \\ \overset{*}{\Rightarrow} a^n b^n & \Rightarrow S S \\ & \overset{*}{\Rightarrow} a^{n_1} b^{n_1} S \\ & \overset{*}{\Rightarrow} a^{n_1} b^{n_1} a^{n_2} b^{n_2} \end{array}$$

Now, I will modify this grammar again a little bit. Now, you tell me what is the language generated? There are only two terminal symbols. The rules are a S goes to a S b, S goes to a b and two more rules some S 1 goes to S S 1, S 1 goes to there are two non terminals. The rules are like this, four rules are there, S 1 goes to S S 1, S 1 goes to epsilon, S goes to a S b, S goes to a b.

Now, S 1 is the start symbol. What set of strings can be derived in this grammar. Now, you can have S 1 derives epsilon, you can use this alone. So, epsilon belongs to L G, in the previous three example the language did not generate the empty string. Empty string cannot be generated in the previous three examples, which we consider whereas, in this example you find that epsilon is generated by this. Then I can use S S 1. Now, replace this S 1 by epsilon. So, you get S then from this S you can generate a power n, b power n or start with S 1, 1 S I can have and again from this S 1 another S I can generate, then I can make this S 1 go to epsilon.

So, now two S S I have, from one of the S S I generate a power n 1 b power n 1, by the usual method, by many steps, then from the other S I can generate a power n 1, b power n 1, a power n 2, b power n 2, from the second S I am generating a power n 2, b power n 2, this sort of a thing I can do any number of times.

(Refer Slide Time: 29:05)


$a^{n_1} b^{n_1} a^{n_2} b^{n_2} a^{n_3} b^{n_3} \ldots a^{n_k} b^{n_k}$
$L = \{a^n b^n / n \geq 1\}$
$L(G_4) = L^*$
$L^* = L^0 \cup L^1 \cup L^2 \ldots$
S_1
S
$a^{n_1} b^{n_1}$
$a^{n_2} b^{n_2}$
ϵ

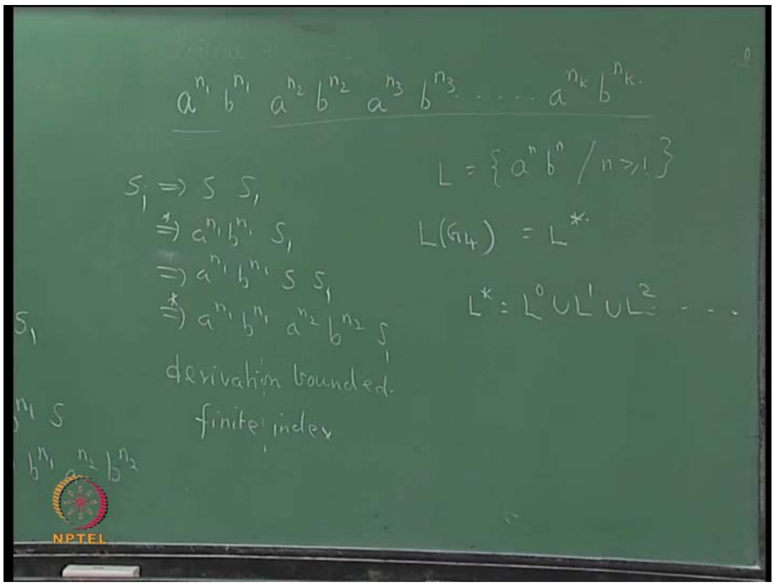
So, in general you can derive strings of the form a power n 1, b power n 1, a power n 2, b power n 2, a power n 3, b power n 3, a power n k, b power n k, strings of the form a power n 1, b power n 1, a power n 2, b power n 2, like that can be generated by this grammar. The derivation tree will be like this, from S you S 1 you generate S S 1, from this you can generate a power n 1, b power n 1, from this one again another S S 1, from this you can generate a power n 2, b power n 2, then from this again another S S 1. Like that finally, the S 1 goes to epsilon, the derivation tree will be of this form.

This is also I mean equal number of a's, equal number of b's. What is the basic difference between that and this? There is no nesting there, then there is nesting here see within this left parenthesis and right parenthesis you are having this that sort of a nesting is not possible here. You have a power n 1, b power n 1, a power n 2, b power n 2, like that, but no nesting is there, see the difference. What is the language generated by this grammar? It has strings of the form can you express it in a simplified form suppose, I denote by L, a power n, b power n, n greater than or equal to one the first example, which we considered, what is the language generated by the fourth grammar, I will put it as L G 4 what is the connection between this and this, you have studied set operations on sigma star. So, that is L star, the Kleene closure of L.

What is L star by our definition? L star will be L 0 union, L 1 union, L 2 etcetera. What is L zero? L zero is just epsilon, epsilon can be generated, L 1 is just a power L itself, L2 will be, L 2 will have strings of the form a power n 1, b power n 1, a power n 2, b power n 2, L k will have strings of the form a power n 1, b power n 1 etcetera.

So, the language generated is one star here, this is not same as this here please note that there is nesting.

(Refer Slide Time: 32:50)



$a^{n_1} b^{n_1} a^{n_2} b^{n_2} a^{n_3} b^{n_3} \dots a^{n_k} b^{n_k}$

$L = \{a^n b^n / n \geq 1\}$

$S_1 \Rightarrow S S_1$

$\overset{*}{\Rightarrow} a^{n_1} b^{n_1} S_1$

$\Rightarrow a^{n_1} b^{n_1} S S_1$

$\overset{*}{\Rightarrow} a^{n_1} b^{n_1} a^{n_2} b^{n_2} S_1$

$L(G_4) = L^*$

$L^* = L^0 \cup L^1 \cup L^2 \dots$

derivation bounded

finite index

S_1

$^{n_1} S$

$b^{n_1} a^{n_2} b^{n_2}$

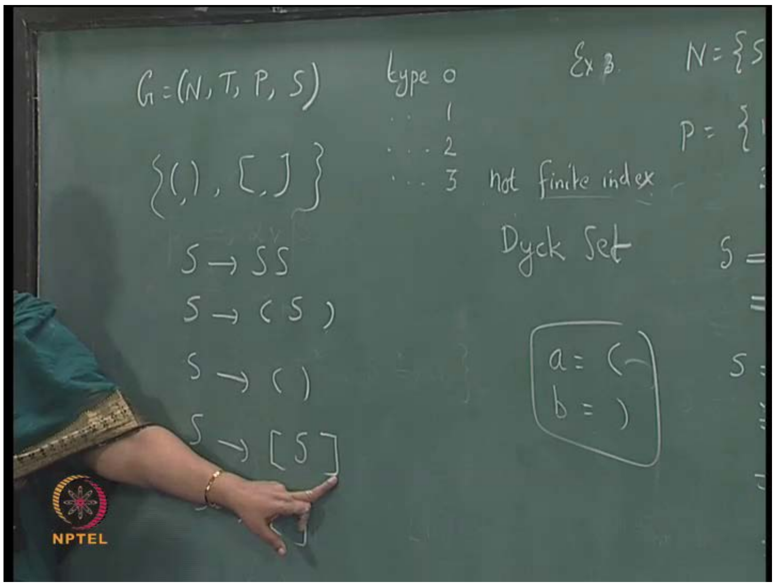
So, there is basically a sort of a difference between this grammar and this grammar. I will slightly put it more express it, suppose I want to derive a power n 1 b power. The string I want to derive this string, how do I go about doing this? Suppose I use a leftmost derivation then afterwards from this I can derive a power n 1, b power n 1.Then I use S S 1, then from this S a power n 2, b power n 2, S 1 and so on. So, you will find that the any string you can derive in such a way using a leftmost derivation. In any sentential form you need to have at most two non terminals, maximum number of non terminals which will be occurring in any sentential form is two, use the leftmost derivation. In this example, I have derived in such a way you have three here, but you could have done with two, first derive the a power n, b power for a S then expand the S S 1, you can proceed like that.

So, you can have a derivation in which every step you will have one or two non terminals only, not more than that. Whereas, in this grammar as the length of the string increases and also as the nesting increases the number of S S in any sentential form will keep on increasing. For example, there may be a string which you cannot derive with four non terminals you may you may require five, the level of nesting is five you may require five S S at one stage similarly, there may be a string, there is a string why there may be there is a string which will require at least ten non terminals in one step. As the nesting and the length of the string increases the number of non terminals in any sentential form you cannot bind it, it is not bound.

So, this is called derivation bounded, and this sort of a thing also it is called the language of finite index. This grammar and this language Dyck set, it is not a finite index. This is the major result. Actually, we will consider more about it a later. So, it belongs to a different class. Now, having done that so far, what we have done is, we have taken grammars and then tried to find the language generated by the grammar. Let us start the simple exercise, the real way you will tackle is you have to find the grammar when the language is given, that is what is required in compilers, you try to define a new programming language or a new programming package or something like that, then you want to write the compiler for that. So, you have to capture the whole feature of that language using a grammar.

So, given the language you will try to find the grammar that is the real problem which you have to tackle.

(Refer Slide Time: 37:14)

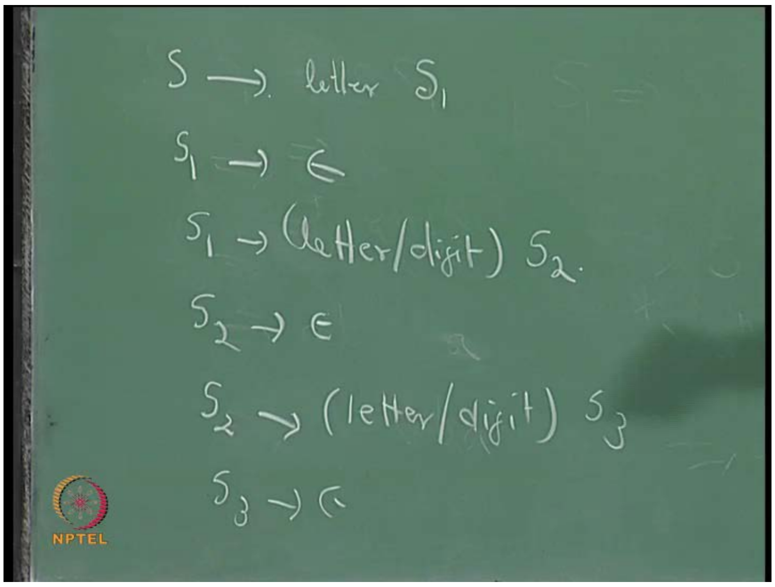


So, let me just change let us consider one or two examples for that, for example, this well formed string of parenthesis we have considered with this. Suppose, I have to find all well formed strings using two types of parenthesis I want to have, still it has to be well formed. What grammar will generate this? You must have this S goes to S S for nesting purpose that it should be there then, you when you generate one left parenthesis you must generate the right parenthesis in between you all you must allow for nesting. So, S goes to this, this, this should be there and terminate also you should have this and similarly, for the other one also you should have.

So, S goes to this and S goes to. So, with this you can generate square parenthesis, square brackets and here ordinary parenthesis you can generate and with the S you can generate S goes to S S and have any sort of combination, but whenever, a left parenthesis, a left square bracket is generated, a right bracket will be generated matching one will be generated.

So, it will always generate only well formed strings of parenthesis. If you want to have another one also you must have two more rules that is all. So, this is actually, this is called Dyck set on two letters, this is called Dyck set on four letters. Generally, you have Dyck set on two n letters, n types of parenthesis you have Dyck set on two n letters, two n symbols.

(Refer Slide Time: 39:33)



$S \rightarrow \text{letter } S_1$

$S_1 \rightarrow \epsilon$

$S_1 \rightarrow (\text{letter/digit})\ S_2$

$S_2 \rightarrow \epsilon$

$S_2 \rightarrow (\text{letter/digit})\ S_3$

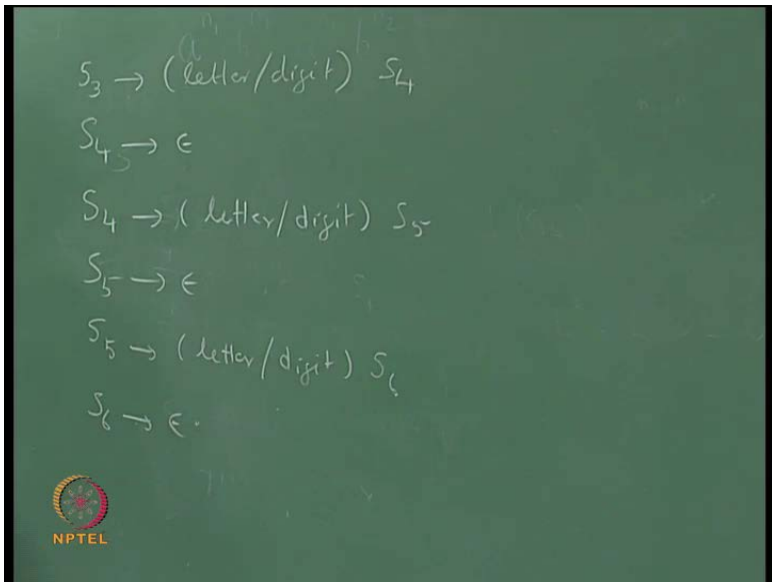
$S_3 \rightarrow \epsilon$

Consider a grammar which will generate Fortran Identifiers. What is the structure of a Fortran Identifiers? It can have up to six symbols, first one will be a letter, second to six it can be digit or a symbol, letter.

So, we will use letters , I would mean I use capital only of course, you can change it, digit denotes 0 to 9. So, how can you generate all Fortran Identifiers you can have non terminal null I will write later S goes to letter S. You can have a rule like this letter should take all possibilities A to Z, that means, twenty six rules will be there and then S 1 goes to epsilon you can have, which would mean you just have a identifier a, c, x like that.

Now, you can have an identifier with two letters, second one can be a digit also, you should allow for that. So, what do you do? You allow S 1 goes to letter or digit S2. So, after generating one letter you can generate another letter r i digit. So, there will be actually I am writing like this it represents thirty six rules, then you can stop at that stage S 2 goes to epsilon or you can continue, you can have up to six letters up to six symbols. The first one will be a letter, second to fifth, sixth one, it can be a letter or a digit.

(Refer Slide Time: 42:48)



$S_3 \rightarrow (\text{letter}/\text{digit})\ S_4$

$S_4 \rightarrow \epsilon$

$S_4 \rightarrow (\text{letter}/\text{digit})\ S_5$

$S_5 \rightarrow \epsilon$

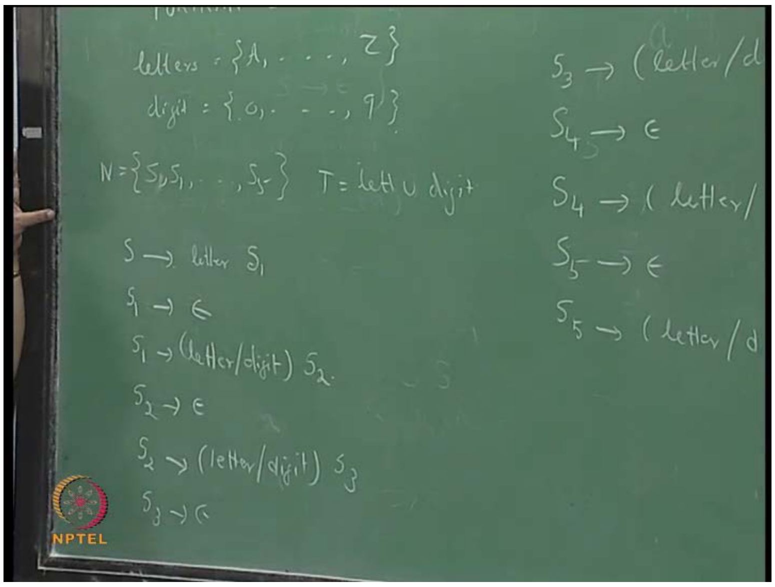
$S_5 \rightarrow (\text{letter}/\text{digit})\ S_6$

$S_6 \rightarrow \epsilon.$

So, S 2 goes to epsilon will terminate with the second symbol. If you want to use one more symbol S 2 goes to letter digit S 3 and then S 3 will go to epsilon, this will generate up to three symbols, this will generate up to three symbols. If you want to generate the fourth symbol S 3 goes to letter digit S 4, S 4 goes to epsilon. So, this is 1 symbol 2, 3 up to 4 symbols you can generate like that. To generate the fifth one, then to generate the sixth one or I can combine this actually this represent twenty six rules, this represent thirty six rules you must remember that.

So, you can have a grammar which will generate all Fortran Identifiers. We can express it as a regular expression, this we shall learn in due course and what type of a grammar is this? Look at this and tell me what type of a grammar is this? You can see that the left hand side is a single non terminal, the right hand side has one symbol followed by a non terminal. It can be a digit or a letter followed by a non terminal or S goes to epsilon. This is the definition of type three grammar which we have considered.

(Refer Slide Time: 44:37)



letters = $\{A, \ldots, Z\}$

digit = $\{0, \ldots, 9\}$

$N = \{S, S_1, \ldots, S_5\}$ $T = \text{letter} \cup \text{digit}$

$S \to \text{letter } S_1$

$S_1 \to \epsilon$

$S_1 \to (\text{letter}/\text{digit})\, S_2$

$S_2 \to \epsilon$

$S_2 \to (\text{letter}/\text{digit})\, S_3$

$S_3 \to \epsilon$

$S_3 \to (\text{letter}/\text{d}$

$S_4 \to \epsilon$

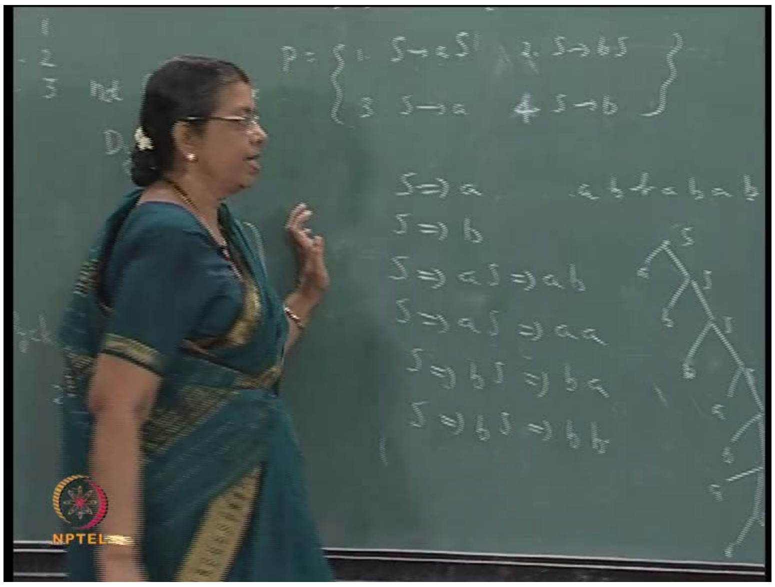
$S_4 \to (\text{letter}/$

$S_5 \to \epsilon$

$S_5 \to (\text{letter}/\text{d}$

So, this is the type three grammars or a regular grammar. So, the non terminals we have used here are S S, S 1 2, S 5, terminal symbols are letters, union, digit.

(Refer Slide Time: 45:06)



$P = \{ 1.\ S \to aS \quad 2.\ S \to bS \quad 3\ S \to a \quad 4\ S \to b \}$

$S \Rightarrow a$

$S \Rightarrow b$

$S \Rightarrow aS \Rightarrow ab$

$S \Rightarrow aS \Rightarrow aa$

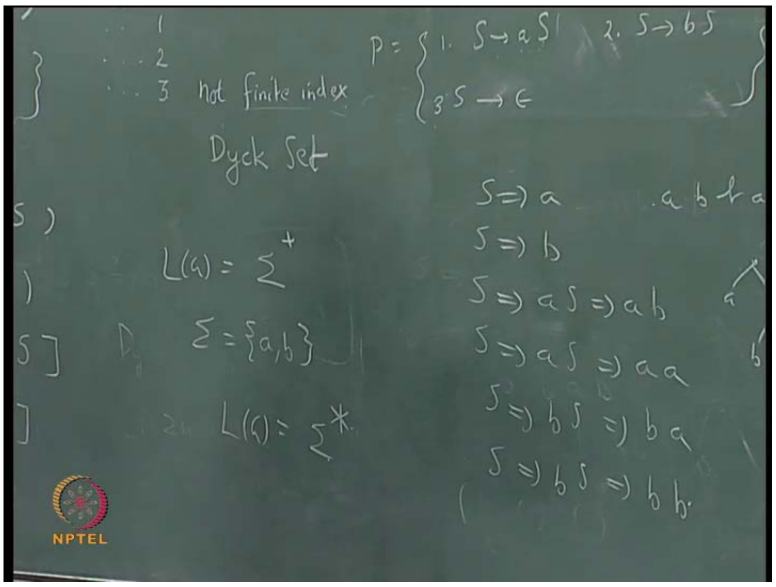
$S \Rightarrow bS \Rightarrow ba$

$S \Rightarrow bS \Rightarrow bb$

Now, going back to type three look at this grammar. Let us consider one more example, the grammar has four rules S goes to a S, S goes to b S, there is only one non terminal S and two terminal symbols a and b. There are four rules, S goes to a S, S goes to b S, S goes to a, S goes to b. What is the language generated? What type of a grammar is this? On the left hand side you are having a single non terminal, on the right hand side you are having a terminal symbol followed by a non terminal or just a terminal symbol.

So, this is a type three grammar and what sort of strings will be generated in this grammar? For example, I can generate a alone, I can generate b alone, I can generate a b, I can generate a a, I can generate b a, I can generate b b, all four strings of length two are can be generated. In fact, you can see that any string of a's and b's can be generated by this grammar, any sequence. Suppose, I have something like a, b b, a b, a b, something like that if I want to generate, whenever, I want to generate a I use rule one. Then rule two, rule two, rule one, rule two, rule one then ending up I have to use rule four for b rule three for a the derivation tree will be like this for this S goes to a ,S b, S b, S it has special structure you can see that a S, a b b, a, b, S, a b b, a, b, a, S, the derivation tree will look like this.

(Refer Slide Time: 47:35)



So, any string of a's and b's will be generated. So, in this case L G is, L G will be a sigma is a b, any string of a b's, but epsilon cannot be generated by this. Now, instead of these two rules I just have S goes to epsilon then, I can generate epsilon also I can generate any string of a's and b's. So, if I replace the two rules by S goes to epsilon the language generated will be sigma star, a sigma is a b, any string of a's and b's can be generated. Type three is the simplest form of a grammar, regular grammar and it is used to describe there like a identifiers, integers, try to generate integers other things using type three grammar, decimals and so on. We can do that. So, the idea of type three grammars is very much used in lexical analysis where you have to recognize the tokens. Tokens are recognized as key words or identifiers or operators or equal to operator and so on, and they are put in symbol tables or constant table and so on. There will be a pointer and so on, we will spend some time on that later on.